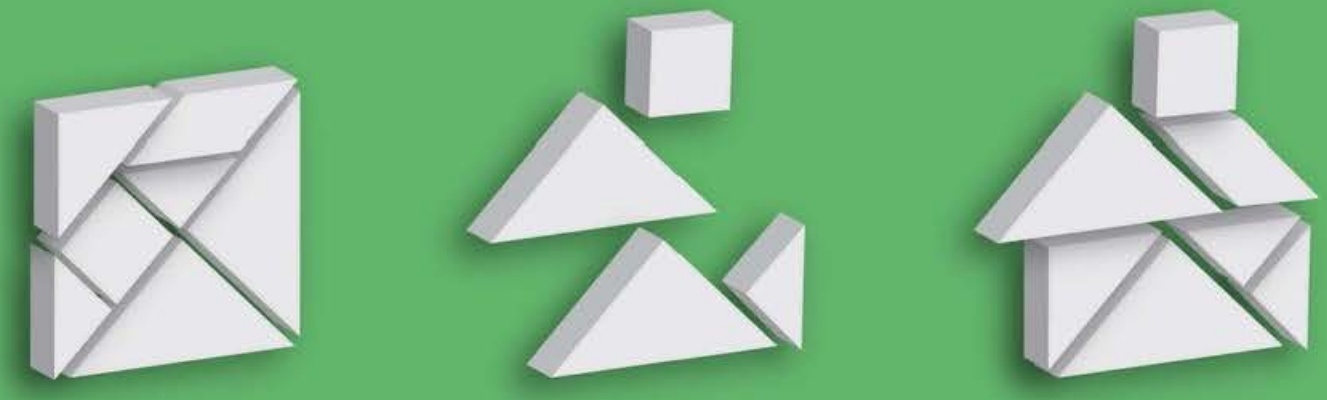
Валерий Лаптев

C++

20 уроков для начинающих





Материалы
на www.bhv.ru

bhv®

Валерий Лаптев

C++

20 уроков для начинающих

Санкт-Петербург
«БХВ-Петербург»
2025

УДК 004.438C++
ББК 32.973.26-018.1
Л24

Лаптев В. В.

Л24 C++. 20 уроков для начинающих. — СПб.: БХВ-Петербург, 2025. — 368 с.: ил.

ISBN 978-5-9775-1961-8

Даны основы программирования на C++ по стандартам C++17 и C++20, базовые структуры данных и методы работы с ними. С позиций разработки ПО рассмотрено программирование функций: обычных и рекурсивных, constexpr-функций, лямбда-выражений и лямбда-функций, перегрузка и шаблоны функций. Раскрыты более сложные вопросы C++: указатели и динамическая память, структуры и классы, конструкторы и деструкторы, методы и перегрузка операций, работа с текстовыми файлами, устройство чисел и битовые операции.

Рассмотрены важные для начинающих темы: интегрированная среда Code::Blocks, русификация ввода и вывода, ошибки при написании кода, неопределенное поведение, тестирование и написание тестового кода. В книге 150 примеров программ и более 90 заданий для самостоятельной работы. Материал проверен на практике при обучении студентов и школьников и подготовке их к экзаменам.

Электронное приложение-архив на сайте издательства содержит исходные коды всех примеров.

Для начинающих программистов

УДК 004.438C++
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Людмила Гауль*
Компьютерная верстка	*Натальи Смирновой*
Корректор	*Людмила Минина*
Оформление обложки	*Зои Канторович*

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-5-9775-1961-8

© ООО "БХВ", 2025
© Оформление. ООО "БХВ-Петербург", 2025

Оглавление

Предисловие **11**

Что вы найдете в книге 12
Используемое ПО 13
Благодарности 14
Как связаться с автором 14

ЧАСТЬ I. Знакомство с C++ **15**

Урок 1. Начало **16**

Процесс создания программы 17
Программы-инструменты 19
    Редактор кода 19
    Компилятор и линкер 19
    Консоль или терминал 20
Интегрированная среда 22
Онлайн-компиляторы 23
Задачи и упражнения 24

Урок 2. Здравствуй, C++ **25**

Первая программа 25
    Директива *#include* 26
    Using-директива 26
    Функция *main* 27
        Заголовок функции 27
        Тело функции 28
        Оператор вывода 28
        Оператор *return* 29
Пишем комментарии 29
Копнем глубже 31
    Формат кода 31
    Директивы 32
    Алфавит и зарезервированные слова 32
Модифицируем первую программу 33
    Специальные символы 34
    Необрабатываемые строковые литералы 35
    Снова рефакторинг 36

Русские буквы в консоли ... 37
Вывод по-русски в системе Windows ... 38
Немного арифметики ... 38
Задачи и упражнения ... 40

Урок 3. Погружаемся в C++ ... 42

Ввод данных ... 42
Работаем со строками ... 45
Снова таблица умножения ... 49
Вычисления ... 50
Вещественные числа ... 50
Как писать вещественные константы ... 51
Тестируем программу ... 52
Вывод данных ... 55
Манипуляторы ... 55
Снова вычисляем ... 56
Задачи и упражнения ... 58

Урок 4. Встроенные типы данных ... 60

Арифметические типы данных ... 60
Вещественные типы ... 60
Целые типы ... 63
Операции, выражения и присваивания ... 65
Арифметические операции ... 66
Присваивание ... 67
Инкремент и декремент ... 68
Преобразование типов ... 69
Объявление переменных и констант ... 71
Объявления *auto* ... 73
Синонимы стандартных целых типов ... 73
Объявление синонимов ... 74
Порешаем задачи ... 75
Задача 1. Математика должна быть экономной ... 75
Задача 2. Стоимость учебников ... 76
Задача 3. Число-палиндром ... 77
Задачи и упражнения ... 78

Урок 5. Выбор есть всегда ... 80

Логический тип ... 80
Логические операции и выражения ... 81
Проверяем входные данные ... 83
Условная операция ... 83
Операция *запятая* ... 84
Решаем задачу ... 85

Условный оператор ... 85
Решаем задачи ... 88
Задача 1. Решить уравнение $ax + b = 0$ ... 88
Задача 2. Високосный год ... 89
Булевские, целые и условия ... 91
Ныряем глубже ... 92
Задача 3. Кирпич и отверстие ... 92
Никогда так не делайте ... 93
Снова про ввод данных ... 94
Задачи и упражнения ... 96

Урок 6. Повторение — мать учения ... 97

Повторяем работу ... 97
Операторы цикла ... 99
Оператор *while* ... 99
Опять таблица умножения ... 101
Оператор *do while* ... 103
Оператор *for* со счетчиком ... 104
Решаем задачи ... 105
Полная таблица умножения ... 105
Возведение в степень ... 106
Вариации на тему циклов ... 108
Операторная область видимости ... 110
Операторы *break* и *continue* ... 111
Задачи и упражнения ... 112

Урок 7. Функции ... 114

Объявление и вызов функции ... 115
Как написать функцию ... 116
Определение функции ... 118
Осваиваем функции ... 119
Математические функции с числом π ... 120
Функция-предикат ... 123
Список параметров и вызов функции ... 124
Передача параметров по значению ... 125
Передача параметров по ссылке ... 126
Константные параметры функций ... 127
Область видимости и время жизни ... 128
Сокрытие имен ... 129
Статические переменные ... 131
Автоматизация тестирования функций ... 133
Стандартные средства тестирования ... 134
Задачи и упражнения ... 135

Урок 8. С пользой для дела ... 137

Тренажер по таблице умножения ... 137
Случайные числа ... 138
Инициализация датчика случайных чисел ... 138
Получение случайных чисел ... 142
Разработка программы ... 143
Оператор-переключатель ... 146
Система обучения арифметике ... 148
Пишем калькулятор ... 149
Пишем еще один тренажер ... 150
Задачи и упражнения ... 152

Часть II. Работа с данными в C++ ... 153

Урок 9. Массивы ... 154

Зачем нужны массивы ... 154
Встроенные массивы ... 156
Объявление массивов ... 156
Синонимы типов для массивов ... 158
Доступ к элементам ... 158
Размер массива ... 159
Операции с массивами ... 159
Новые массивы ... 161
Размер и тип массива ... 161
Новый цикл *for* ... 162
Массив *array* ... 163
Объявление и инициализация ... 163
Операции с массивом *array* ... 164
Объекты типа *initalizer_list* ... 165
Решаем задачи ... 165
Поиск минимума ... 166
Подсчет максимумов ... 167
Поиск заданного числа ... 168
Задачи и упражнения ... 170

Урок 10. Создаем новые типы ... 171

Структуры (классы) ... 171
Зачем нужны структуры (классы) ... 171
Определение структуры (класса) ... 172
Объявление и инициализация объектов ... 174
Операции с объектами структурного типа ... 176
Размеры структур ... 177
Структуры, массивы, объекты и функции ... 178
Массивы и структуры ... 179
Структуры, объекты и функции ... 180

Классы и структуры ... 182
Конструкторы ... 184
Методы ... 186
Задачи и упражнения ... 187

Урок 11. Знакомство с указателями ... 188

Что такое указатели ... 188
Две самые важные операции ... 190
Объявление указателей ... 191
Типизированные указатели ... 192
Бестиповые указатели и преобразование типов ... 193
nullptr ... 194
Константность ... 195
Размеры указателей ... 195
Указатели и ссылки ... 196
Встроенные массивы и указатели ... 196
Операции с указателями ... 197
Указатели, итераторы и контейнеры ... 200
Функция *distance* ... 203
Итераторы ввода/вывода ... 204
Задачи и упражнения ... 205

Урок 12. Массивы и функции ... 206

Массивы и функции ... 206
Передача массива по ссылке ... 207
Перегрузка функций ... 209
Про массив *array* ... 211
Шаблоны функций ... 213
Стандартные алгоритмы ... 216
Заполнение массива ... 216
Заполнение случайными числами ... 218
Поиск, подсчет, замена ... 219
Числовые алгоритмы ... 220
Задачи и упражнения ... 220

Урок 13. Строки, строки, строки ... 222

Строки как массивы символов ... 222
Итераторы для строки ... 223
Стандартные средства ... 224
Строка — массив переменного размера ... 224
Встроенные символьные массивы ... 225
Размер встроенного символьного массива ... 226
Обработка символьных массивов ... 227
Универсальный параметр *string_view* ... 229

Размеры строк STL ... 230
Числа — прописью ... 231
Тестирование программы ... 234
Длинные-длинные числа ... 235
Символы — это числа ... 235
Функция сложения ... 236
Задачи и упражнения ... 237

Урок 14. Снова о функциях ... 238

Рекурсивные функции ... 238
Осваиваем рекурсию ... 241
Формы рекурсивных функций ... 243
Выполнение рекурсивных функций ... 244
Рекурсивные функции и массивы ... 245
Функции времени компиляции ... 250
Задачи и упражнения ... 253

Урок 15. Печальная действительность ... 254

Проблемы с целыми числами ... 254
На ноль делить нельзя ... 257
Вещественные числа тоже не подарок ... 258
0.1 + 0.2 != 0.3 ... 259
«Страшный зверь» по имени UB ... 261
Как не наступать на «грабли» ... 263
Исключения ... 263
Генерация исключений ... 263
Перехват и обработка исключений ... 266
Стандартные исключения ... 268
Обработка «плавающих граблей» ... 269
Функции *isnan*, *isinf*, *isfinite* ... 270
Задачи и упражнения ... 270

Часть III. Вглубь C++ ... 271

Урок 16. Как все устроено ... 272

Числа, числа и числа ... 272
Числа как многочлены ... 274
Перевод из десятичной в другую систему ... 275
Перевод в десятичную систему ... 277
Стандартные средства ... 278
Перевод дробных чисел ... 279
Битовое устройство беззнаковых целых ... 281
Битовые операции ... 282
Как писать двоичные литералы ... 283
Исключающее *ИЛИ* ... 284
Сдвиги ... 285

Битовое устройство целых со знаком ... 286
Сдвиг вправо знакового целого числа ... 288
Дополнительный код ... 288
Задачи и упражнения ... 289

Урок 17. Опять функции ... 290

Параметры по умолчанию ... 290
Функции как параметры ... 291
Функциональные объекты ... 293
Указатель на функцию ... 293
Лямбды ... 294
Лямбда-функции ... 296
Захват ... 298
Перегрузка операций ... 299
Задачи и упражнения ... 300

Урок 18. Снова указатели ... 301

Память и объекты ... 301
Динамическая память для встроенных типов ... 302
Динамическая память для реализованных типов ... 303
Деструктор ... 304
Динамические массивы ... 305
Размеры массивов ... 307
Стандартный динамический массив *vector* ... 309
Объявление и инициализация вектора ... 309
Размеры векторов ... 310
Операции с вектором ... 311
Снова длинные числа ... 312
Многомерные массивы ... 313
Динамические многомерные массивы ... 316
Задачи и упражнения ... 317

Урок 19. Учимся писать и читать ... 318

Потоки ... 318
Стандартные потоки ... 319
Ввод встроенных типов ... 319
Ввод строк с пробелами ... 321
Пишем новый калькулятор ... 322
Строковые потоки ... 323
Файлы ... 325
Файлы и потоки ... 326
Читаем и пишем ... 326
ЕГЭ и файлы ... 329
Задачи и упражнения ... 331

Урок 20. Большие программы ... 332

Проект «Арифметика» ... 333
Разделение на модули ... 333
Вывод инструкции ... 335
Первая сборка ... 336
Межмодульное взаимодействие ... 336
Связывание ... 337
Пространства имен ... 339
Добавим тренажеры ... 339
Препроцессор и модули ... 344
Задачи и упражнения ... 345

Подведение итогов ... 346

Приложения ... 347

Приложение 1. Список литературы ... 348

Приложение 2. Перечень ресурсов Интернета ... 349

Приложение 3. Установка Code::Blocks ... 350

Загрузка установщика в Windows ... 350
Установка Code::Blocks в Windows ... 352
Установка Code::Blocks в «Альт Образование» ... 355
Установка Code::Blocks в MOC ... 355

Приложение 4. Интегрированная среда Code::Blocks ... 356

Запуск IDE и выход ... 356
Выключение ненужных панелей инструментов ... 357
Создание и сохранение исходного кода ... 358
Создание файла ... 358
Создание проекта ... 360
Установка режимов компиляции ... 362
Компиляция и сборка ... 363
Сообщения об ошибках ... 363
Выполнение программы ... 364

Предметный указатель ... 365

Предисловие

Здравствуйте, читатель! Если вы держите в руках эту книгу, значит, вас по какой-то причине заинтересовал C++. Но, возможно, вы сомневаетесь, что вам по силам освоить этот язык программирования, ведь он такой сложный. Однако не так страшен C++, как его «малюют». Наглядное подтверждение тому — автор этой книги.

Когда-то очень давно мне в руки впервые попала книга Б. Страуструпа о языке C++. Когда я ее открыл и попытался освоить C++, я ничего не понял. Но с тех пор «много воды утекло». Я прочитал много хороших книг по C++, мне довелось программировать на C++ и я разобрался, что он не сложнее других языков программирования. Вот уже 25 лет я обучаю студентов языку C++, 20 лет назад я написал свою первую книгу[1] по C++ для начинающих программистов. Потом были написаны другие книги по C++. И вот вы держите в руках еще одну.

Эта книга представляет собой введение в практическое программирование на C++ с самого начала. Все написанное в ней проверено временем и практикой обучения студентов и школьников, при подготовке к ЕГЭ по информатике и при начальном обучении олимпиадному программированию.

Книгу может читать любой начинающий, которому надо научиться писать программы на C++. Это и первокурсник, который изучает программирование в вузе, и просто новичок, который решил научиться программировать, и школьник 7–11 класса, участвующий в олимпиадах по программированию. Если читатель имеет опыт программирования на любом языке (пусть даже и очень небольшой), это только поможет ему освоить C++.

Как известно, официальным языком программирования на Всероссийской олимпиаде школьников по информатике является C/C++. В книге описаны некоторые методы и приемы олимпиадного программирования, где можно применять стандартные средства современного C++. Не менее одной трети задач ЕГЭ по информатике решаются с помощью программирования, и этому тоже уделено внимание.

Основное содержание знакомит читателя с реальным программированием на современном C++. На сегодняшний день нужно ориентироваться на стандарты C++17 и C++20. Новый стандарт C++23 был утвержден только в конце 2023 года и в полной мере в промышленных компиляторах еще не реализован. Кроме того, книга написана с учетом скорого перехода России на Linux.

---

[1] Лаптев В.В. C++. Экспресс-курс. — СПб.: БХВ-Петербург, 2004. — 512 с.

До недавнего времени основной операционной системой в российских вузах и школах была Windows. Однако с 30 сентября 2023 года корпорация прекратила поддержку своего софта для российских компаний. Понятно, что теперь для России переход на Linux — только вопрос времени. Это касается и российской системы образования в целом.

Что вы найдете в книге

В первой части (*уроки 1–8*) излагаются основы программирования на языке C++.

«*Урок 1. Начало*» рассказывает о том, что такое программирование, какие программы используются для создания программ, что такое IDE и как с ней работать.

В «*Уроке 2. Здравствуй, C++*» детально разбирается программа «Hello World», комментарии, формат кода. Первая программа несколько раз модифицируется, изучаются возможности вывода в консоль. Завершается урок написанием нескольких простых программ.

«*Урок 3. Погружаемся в C++*» знакомит читателя с основными понятиями программирования и особенностями C++ (переменная, тип данных, строгая типизация, вещественные числа, строки, новые возможности ввода/вывода и т. п.). Рассматриваются простые вычислительные задачи и объясняется процесс тестирования программы.

В «*Уроке 4. Встроенные типы данных*» подробно рассказывается об арифметических типах данных в C++ и операциях над ними. Рассматриваются возможности объявления переменных и констант, преобразование типов, решается несколько задач.

В «*Уроке 5. Выбор есть всегда*» описываются особенности условного оператора, логические выражения и связанные с ними операции. Написаны несколько программ для изучения этого материала на практике.

«*Урок 6. Повторение — мать учения*» — это подробное описание операторов цикла и сопутствующих операторов. Реализуются программы, на примере которых объясняются различные нюансы написания циклов.

В «*Уроке 7. Функции*» рассказывается об основных средствах C++ для написания функций. Некоторые программы из предыдущих уроков переписаны в виде функций. Излагаются особенности тестирования функций.

И наконец, в завершающем «*Уроке 8. С пользой для дела*» описывается разработка реально полезных программ — тренажеров. В связи с реализацией этих программ объясняется оператор-переключатель.

Во второй части (*уроки 9–15*) изучаются структуры данных C++ и методы работы с ними.

В «*Уроке 9. Массивы*» рассматриваются встроенные и array-массивы стандартной библиотеки. Описываются новые возможности C++ для работы с массивами и приводятся типовые алгоритмы поиска по массивам.

В «*Уроке 10. Создаем новые типы*» изучаются структуры и классы, объявление и инициализация объектов с помощью конструкторов, их сочетание с массивами.

Название «*Урок 11. Знакомство с указателями*» говорит само за себя. Описывается связь указателей и массивов, объясняется понятие итератора, рассматриваются сходство и различие указателей и ссылок.

«*Урок 12. Массивы и функции*» продолжает тему массивов, указателей и функций. Описываются перегрузка и шаблоны функций. Рассматриваются стандартные алгоритмы для массивов.

В «*Уроке 13. Строки, строки, строки*» рассматриваются различные виды строк, включенных в C++ и стандартную библиотеку. Реализуется операция сложения длинных чисел на строках.

«*Урок 14. Снова о функциях*» включает материал о рекурсивных функциях и функциях периода компиляции.

В «*Уроке 15. Печальная действительность*» описываются разнообразные проблемы, с которыми сталкивается программист в C++. Рассматриваются способы решения этих проблем с помощью исключений.

В третьей части (*уроки 16–20*) изучаются более глубокие вопросы C++: как устроены числа, работа с динамической памятью, новые возможности функций и работа с текстовыми файлами.

В «*Уроке 16. Как все устроено*» рассказывается о битовом устройстве чисел и битовых операциях. Разъясняется, почему случаются все те неприятности, о которых написано в предыдущем уроке.

В «*Уроке 17. Опять функции*» рассматриваются функции как параметры, лямбда-выражения и перегрузка операций.

«*Урок 18. Снова указатели*» описывает работу с динамическими массивами, с вектором (vector) и снова реализуются длинные числа. Объясняется понятие многомерного массива.

«*Урок 19. Учимся читать и писать*» рассказывает о средствах ввода строк с пробелами. Здесь же описывается работа с текстовыми файлами. Рассматривается организация дополнительных наборов данных в задачах ЕГЭ по информатике.

В завершающем «*Уроке 20. Большие программы*» описана разработка проекта многофункционального тренажера по нескольким темам арифметики для учеников начальных классов.

В конце каждого урока приведены задания и упражнения, которые нужно выполнить для закрепления материала.

В *приложениях 1–4* приводится список литературы, список интернет-источников, процедура установки среды Code::Blocks в Windows и работа в этой среде.

Используемое ПО

Все программы, которые вы обнаружите на страницах книги, реально написаны на стандартном C++ с использованием интегрированной среды Code::Blocks 20.03.

Это свободно распространяемая среда программирования, она устанавливается и работает в любых операционных системах.

Как я упомянул выше, для России переход на Linux неизбежен. У нас существует несколько дистрибутивов Linux. Можно выделить два из них, которые предназначены для образования: «Альт Образование» от компании Базальт СПО и РОСА «ХРОМ» для образования от НТЦ ИТ РОСА. На базе РОСА недавно создан школьный дистрибутив МОС, который установлен во всех московских школах.

В этих операционных системах предустановлена интегрированная среда Code::Blocks 20.03. Кроме того, во всех системах Linux установлены компиляторы C/C++ из коллекции компиляторов GNU GCC. На момент написания книги в системе «Альт Образование» установлена версия не ниже 10.1, а в МОС — версия 12.1. Эти версии поддерживают стандарт C++20 почти в полном объеме (кроме некоторых свойств, которые новичкам не понадобятся), тогда как пакет MinGW (Minimal GNU for Windows), установленный вместе с Code::Blocks в Windows, включает компилятор версии 8.1, поддерживающий только стандарт C++17.

Поэтому все программы в книге написаны в стандарте C++17, но некоторые (о чем оговаривается в каждом случае) реализованы в стандарте C++20. Эти программы проверялись и работали в системах «Альт» и МОС.

Все операционные системы, естественно, 64-битные.

Благодарности

Прежде всего хотел бы от души поблагодарить сотрудников издательства «БХВ-Петербург», без самоотверженного труда которых эта книга не могла появиться на свет.

Не могу не поблагодарить моих прекрасных учителей, которые учили меня в Образовательном центре «Сириус» олимпиадному программированию для школьников. Отдельную большую благодарность хочу выразить Михаилу Густокашину, Елене Андреевой и Инессе Шуйковой, чьими материалами я пользовался при написании этой книги.

Большое спасибо моей внучке Анне Павловне, которая пошла во 2-й класс и начала учить таблицу умножения и тем самым вдохновила меня на создание программ-тренажеров по арифметике для нее.

Как связаться с автором

Проще всего это сделать по электронной почте: **WLaptew@yandex.ru** или **vlaptevds@gmail.com**. Напишите мне, и я обязательно отвечу.

- ЧАСТЬ I -

Знакомство с C++

УРОК 1. НАЧАЛО
УРОК 2. ЗДРАВСТВУЙ, C++
УРОК 3. ПОГРУЖАЕМСЯ В C++
УРОК 4. ВСТРОЕННЫЕ ТИПЫ ДАННЫХ
УРОК 5. ВЫБОР ЕСТЬ ВСЕГДА
УРОК 6. ПОВТОРЕНИЕ — МАТЬ УЧЕНИЯ
УРОК 7. ФУНКЦИИ
УРОК 8. С ПОЛЬЗОЙ ДЛЯ ДЕЛА

- Урок 1 -

Начало

Меня часто спрашивают начинающие: что такое программа? Как научиться программировать? Ответ очень простой: программа — это план действий. Планы мы составляем почти каждый день, но практически никогда их не записываем. Примерами записанных планов являются «план модернизации», «программа развития», «дорожная карта» и т. п. Составлять и записывать планы — это и есть программирование.

План можно записать на своем родном языке: мы с вами пишем на русском языке, англичане — на английском, китайцы — на китайском. План можно нарисовать, тогда он называется схемой. В программировании план обычно называют *алгоритмом.*

План составляется не просто так, а всегда для решения какой-нибудь задачи. Задачу надо решить, поэтому план (алгоритм) кто-то должен выполнить. Этот «кто-то» является *исполнителем.* До появления компьютеров исполнителем планов был человек. Компьютер — это тоже исполнитель написанных планов (алгоритмов). Но язык, на котором пишутся алгоритмы для компьютера, — искусственный, специально созданный для этой цели, и он называется *язык программирования.* Алгоритм, записанный на этом языке, называется *исходным кодом* программы.

Язык программирования — это «иностранный» язык для человека. Опытный программист уже выучил этот «иностранный» язык и пишет программы сразу на нем. А начинающий так делать не может — ему проще написать алгоритм на своем родном языке, а затем перевести его на язык программирования.

История из жизни

Тридцать лет назад я учил программированию на Паскале своего первого дипломника. Недавно мы с ним встретились, и он мне рассказал, как я его научил программировать на первом курсе. С его слов, он просто ничего не понимал, т. к. у них в сельской школе, где он учился, не было информатики. Он сильно переживал, что он такой «тупой» и никак не может ничего понять. Так продолжалось до тех пор, пока я ему однажды не сказал: ты напиши сначала по-русски план, что нужно сделать. Он так и сделал. И в этот момент его осенило: написанный план можно перевести на Паскаль! С этого момента проблем с программированием у него не было. В дипломном проекте он разработал программу, переводящую код на Java в код на C++. Это был один из лучших дипломных проектов выпуска 1999 года.

Алгоритм записывают на естественном языке, но не произвольным образом, а по некоторым правилам. Такой язык обычно называют *псевдокодом*. Мы будем применять псевдокод для записи алгоритмов.

Алгоритм потом нужно перевести в исходный код на языке C++.

А теперь рассмотрим процесс создания программы после того, как алгоритм переведен в исходный код (этот процесс изображен на рис. 1.1).

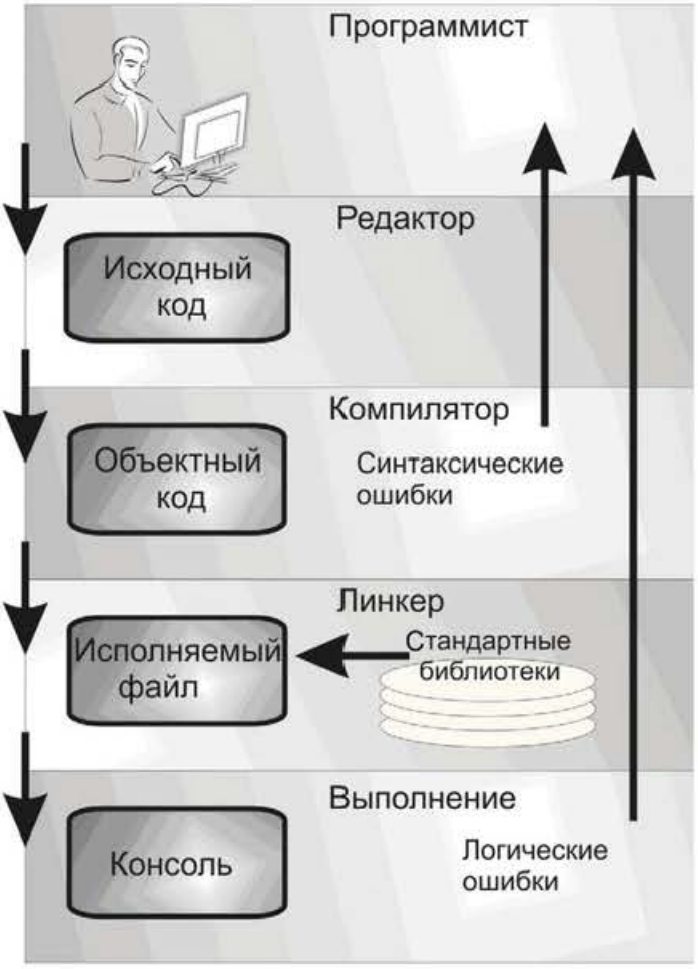


Рис. 1.1. Процесс создания программы

Процесс создания программы

Алгоритм — это набор действий, которые надо выполнить в определенном порядке. Действия часто называют шагами и говорят, что алгоритм содержит такие-то шаги. Напишем на псевдокоде алгоритм создания программы (см. рис. 1.1):

1. Подготовка исходного кода на языке программирования (например, C++).
2. Компиляция исходного кода (в объектный код).
3. Если есть ошибки компиляции, то вернуться к шагу 1 (исправить ошибки).
4. Иначе (если ошибок компиляции нет) выполняется сборка программы.
5. Собранная программа запускается для тестирования.

6. Если тестирование показывает неправильные результаты, то вернуться к шагу 1 (исправить ошибки).
7. Ура, заработала!

Этот алгоритм исполняет программист. Эти шесть шагов повторяются при разработке каждой программы, даже самой маленькой. Работа на каждом шаге описана ниже.

Первые два шага — это действия, которые надо выполнить. А вот на третьем шаге происходит проверка результатов второго шага, и дальнейшие действия зависят от этой проверки. Третий шаг алгоритма называется *ветвлением*.

При отсутствии ошибок на третьем шаге автоматически выполняется сборка, потом программист выполняет шаг 4: запускает программу для тестирования. И снова дальнейшие действия зависят от результатов предыдущего шага.

Однако программисты, как и все люди, делают ошибки. В программировании любая ошибка — это ошибка: программа либо работает, либо не работает. Поэтому приходится возвращаться к первому шагу и исправлять исходный код. Сколько раз придется это делать — заранее не известно. Это зависит и от задачи, и от квалификации программиста. Такое повторение действий тоже пишут на псевдокоде с помощью слова «повторить» (или «повторять»):

1. Повторять

   1.1. Подготовка исходного кода на языке программирования.

   1.2. Компиляция исходного кода (в объектный код).

2. Пока есть ошибки компиляции.

   Нумеровать шаги можно как угодно — лишь бы было понятно. Повторяемые шаги обычно сдвигаются немного правее, как в этом примере.

   Шаг 2 — это программист проверяет результаты компиляции. Если есть ошибки, то первые два шага повторяются. Если ошибок нет, то повторение завершается и выполняется шаг сборки:

3. Сборка программы.

   Потом надо запустить тестирование:

4. Запустить программу для тестирования.

И снова при наличии ошибок надо начинать сначала — это тоже повторение. Но надо повторять с первого шага. Напишем повторение после тестирования немного по-другому:

```
Делать, пока есть ошибки тестирования          // повторение с условием
   Повторять                                   // повторение
        Подготовка исходного кода на языке программирования
        Компиляция исходного кода (в объектный код)
   Пока есть ошибки компиляции                 // условие
```

```
   Сборка программы
   Запустить программу для тестирования
Конец делать                                    // просто обозначен конец
Ура, заработала!
```

Можно писать алгоритмы и без нумерации шагов.

Программы-инструменты

Программист, который пишет программу на языке программирования C++, на каждом шаге должен использовать программы, чтобы с их помощью создать свою. Эти программы являются системными программами и обычно называются *инструментами* для программиста. Все программы-инструменты работают в операционной системе (ОС), которая установлена на компьютере. В этой книге все инструменты будем запускать в ОС Windows и Linux («Альт» и МОС). Рассмотрим самые основные программы-инструменты.

Редактор кода

Первая такая программа — редактор кода. Это основной рабочий инструмент программиста. В редакторе набирается (и исправляется при необходимости) исходный код программы на языке C++.

Сначала в текстовом редакторе нужно создать файл (с исходным кодом) или открыть уже существующий файл. С текстом (исходным кодом) программист работает, добавляя новые строки кода или изменяя существующие.

Набранный код нужно сохранить на диске в виде текстового файла. Обычно текстовый файл имеет имя и расширение. Имя завершается точкой, и после нее пишется расширение (обычно из трех букв). Имя файла с кодом на языке C++ задает программист, а расширение обычно бывает cpp или cxx.

Операции в любом текстовом редакторе в общем-то стандартные, поэтому, научившись работать в одном, программист без труда освоит и другой.

Компилятор и линкер

На следующем шаге текстовый файл с программой на C++ «скармливается» компилятору, который анализирует код с целью обнаружить ошибки. Ведь даже на своем родном языке мы делаем ошибки, а тут иностранный, да еще и специальный. Компилятор — это большая и сложная программа. Писали ее много высококвалифицированных программистов. Компиляторов C++ и в мире Windows, и в мире Linux довольно много, но мы будем пользоваться самым широко распространенным, который входит в состав GNU Compiler Collection (GCC).

Компилятор анализирует код, который написал программист. Если что-то написано неправильно, компилятор об этом сообщает на английском языке. Поэтому любой программист обязан разбираться в английском хотя бы на начальном уровне.

Сообщения компилятора делятся на два типа: предупреждения (warning) и ошибки (error). Ошибка — это показатель, что в программе что-то написано совсем неправильно. Например, неверно написано какое-то слово C++ (например, `while`). Это грубые ошибки, которые надо исправить в исходном коде, иначе дальше двигаться невозможно.

Предупреждение — это не ошибка в исходном коде. Однако программист обязан обращать внимание на предупреждения, т. к. они иногда самым существенным образом могут влиять на результаты работы программы. Но выяснится это только при тестировании программы.

Если компилятор не обнаружил ошибок, то далее он преобразует исходный код программы в объектный файл. Объектные файлы имеют расширение obj или o. Имя объектного файла обычно совпадает с именем файла с исходным кодом. Объектный файл частенько называют объектным кодом.

> ***НА ЗАМЕТКУ***
>
> Некоторые конструкции языка C++ могут быть выполнены компилятором без создания объектного кода. Про такие конструкции говорят, что вычисления выполняются *на этапе* компиляции.

Это еще не работающая программа, это заготовка.

Исполняемую программу собирает линкер, который обычно автоматически запускается после создания компилятором объектного файла. По-русски линкер (от англ. *linker*) называется компоновщик. Компоновщик присоединяет к объектному файлу стандартные части из библиотек, которые нужны для работы в данной ОС (Windows или Linux). Объектный код преобразуется в исполняемый вид, разный в разных операционных системах.

Результат сборки — исполняемая программа, которую можно запускать в операционной системе принятым в ней способом. Файл исполняемой программы в системе Linux расширения обычно не имеет, а в системе Windows у таких файлов расширение exe.

Во время сборки тоже могут возникнуть ошибки, но в наших небольших учебных программах этого не случится.

В профессиональном программировании используется еще много разных программ, но нам они не понадобятся.

Консоль или терминал

Собранную программу нужно обязательно проверить, чтобы убедиться в корректности ее работы. Программы, которые мы будем писать, являются *консольными* программами. Это означает, что они обладают консольным *интерфейсом*, в отличие от *оконных* программ, к которым мы все привыкли как пользователи. До появления дисплеев все программы были консольными.

Поэтому проверка программ будет выполняться в консоли (Windows) или в терминале (Linux). Заметим, что программисты обычно говорят «вывод в консоль» (или «на консоль») и «ввод с консоли». В системах Linux вместо слова «консоль» используется слово «терминал». Фактически это одно и то же, но выглядят консоль и терминал немного по-разному.

Термин «консоль» появился давно, когда компьютеры были большими (такие компьютеры сейчас называются мэйнфреймами). В те времена человек с компьютером «общался» с помощью обычной пишущей машинки, которая и называлась «консоль оператора». Клавиатура пишущей машинки была устройством ввода, а бумажная лента в ней — устройством вывода. Потом, когда компьютеры стали поменьше (их стали называть мини-компьютерами), пишущая машинка была заменена на дисплей и клавиатуру, и они вместе стали называться «терминал оператора».

На нашем персональном компьютере такое устройство отсутствует. Вернее, сам персональный компьютер фактически является терминалом: клавиатура есть, а вывод осуществляется на экран. Но мы работаем в оконной системе, которая входит в состав установленной операционной системы, поэтому вывод на экран моделируется с помощью окна.

Если у вас на компьютере установлена операционная система Windows, то это окно (консоль) выглядит так, как показано на рис. 1.2.

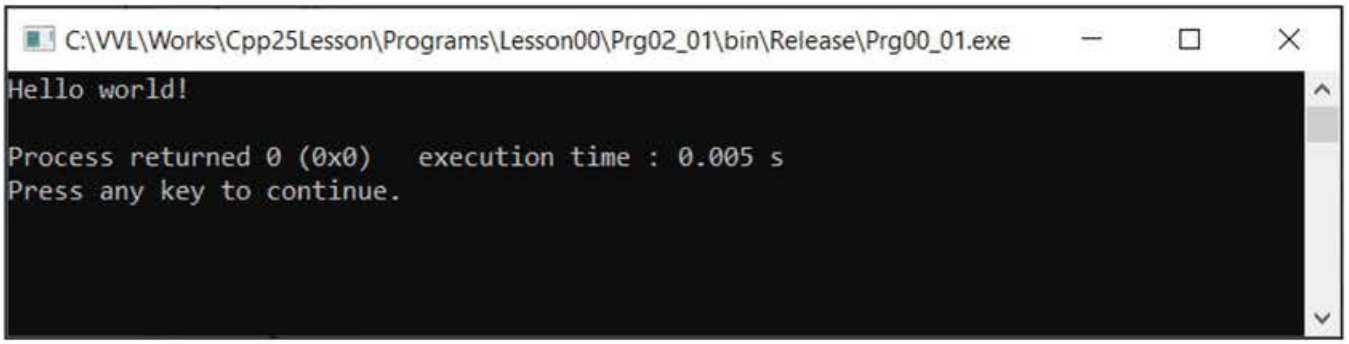


Рис. 1.2. Консоль в Windows

Это окно, как и все другие окна, можно разворачивать на весь экран, перемещать по экрану, сворачивать, изменять размеры и просто закрыть.

В системе «Альт», которая является клоном Linux, вид окна терминала зависит от установленной оконной системы. На моем компьютере окно терминала выглядит так, как показано на рис. 1.3.

Обратите внимание, что окно терминала обладает довольно развитым меню, под которым еще расположен ряд кнопок. Возможности терминала намного шире, чем возможности консоли. В операционной системе МОС установлена по умолчанию оконная система KDE, и в ней окно терминала выглядит немного по-другому. В дальнейшем я буду употреблять термин «консоль».

В консольном окне будут появляться все символы, которые мы будем набирать на клавиатуре компьютера. В консоль также будут выводиться все сообщения наших программ и все результаты выполнения.

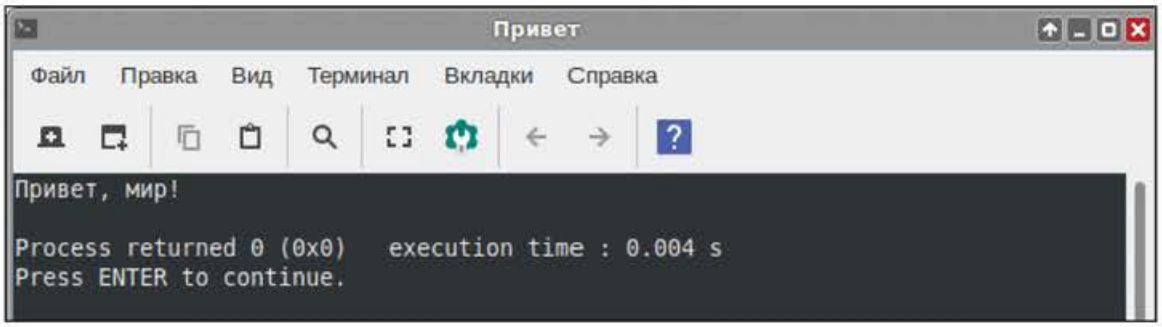


Рис. 1.3. Окно терминала в системе «Альт»

Получение программой данных из внешней среды (с клавиатуры, из файла, из сети) называется *вводом* данных. Слово «вывод» в программировании всегда означает вывод чего-то из программы во внешнюю среду (в файл, в сеть, в консольное окно).

Проверка программы состоит в том, что нужно вводить некоторые данные, набирая их на клавиатуре, а программа будет выводить в консоль то, что мы в ней запрограммируем для вывода. Если сравнить результаты программы с тем, что *должно* быть выведено, можно увидеть, правильно ли работает программа. Конечно, нужно заранее подготовить такие данные, по которым легко вычислить результаты вручную. Такой процесс проверки и называется *тестированием*.

Интегрированная среда

Все эти программы-инструменты (и многие другие) объединяются в единую систему — *интегрированную среду разработки*, или просто интегрированную среду. По-английски она называется Integrated Development Environment или кратко — IDE. Самая известная в России интегрированная среда появилась почти одновременно с персональными компьютерами — это Турбо Паскаль (Turbo Pascal), которой в 2023 году исполнилось уже 40 лет.

Мы в этой книге будем пользоваться средой Code::Blocks. Установка этой IDE в Windows описана в *приложении 3*. Если на вашем компьютере стоит операционная система «Альт» или МОС, то в них среда Code::Blocks уже есть.

Запускать IDE нужно так же, как запускаются программы в ОС, установленной на вашем компьютере. Рабочее окно среды выглядит так, как показано на рис. 1.4.

Как обычно, сверху строка меню. Под ней — панели инструментов (кнопки). Слева — окно проекта, справа — окно редактора кода. Внизу под ним — окно сообщений компилятора.

В интегрированной среде мы будем выполнять весь процесс разработки программы, о котором говорили выше (см. рис. 1.1): создавать и исправлять исходный код, компилировать и собирать исполняемый файл, запускать программу для тестирования, наблюдать и анализировать результаты работы в консольном окне (см. рис. 1.2 и 1.3).

Таким образом, учиться программировать в интегрированной среде намного комфортнее, чем по отдельности самому запускать программы-инструменты.

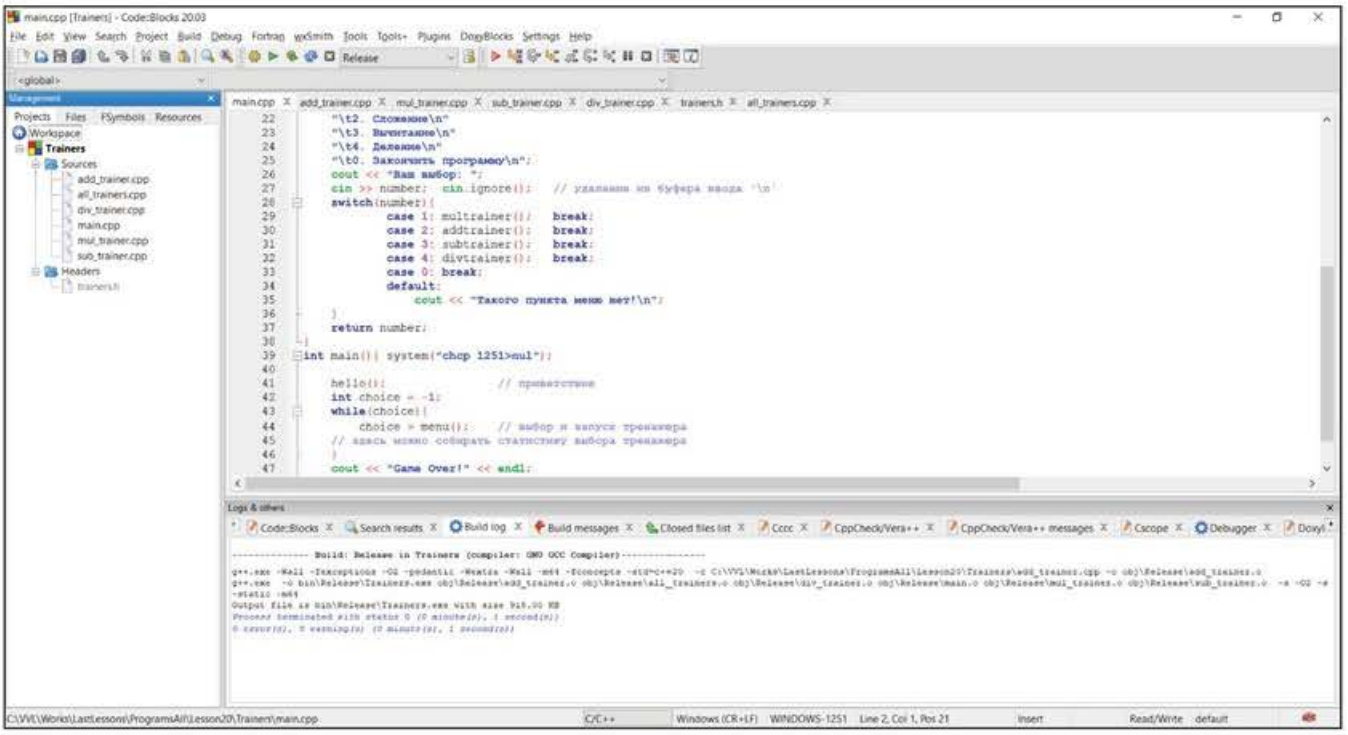


Рис. 1.4. Общий вид среды Code::Blocks

Онлайн-компиляторы

Начать учиться писать программы можно и без установки IDE. В Интернете доступны онлайн-компиляторы, с которыми можно работать непосредственно в браузере. Наберем в адресной строке адрес **`https://gcc.godbolt.org`**. Откроется страница, показанная на рис. 1.5.

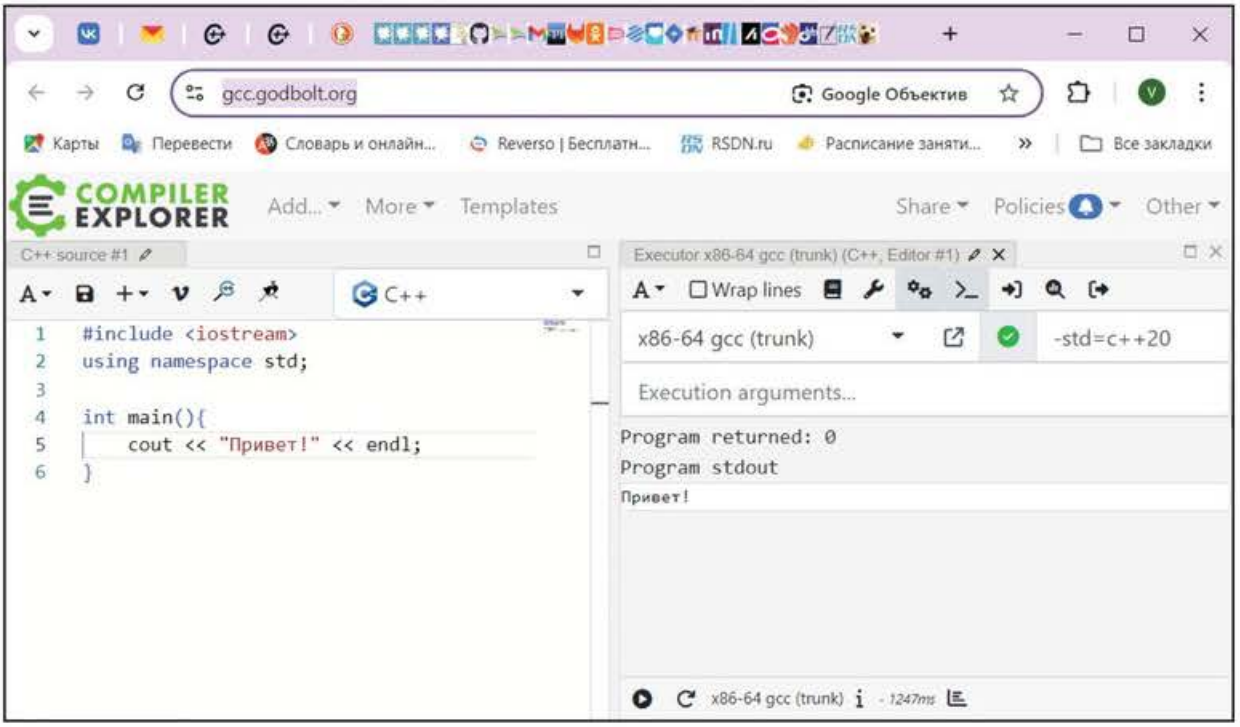


Рис. 1.5. Онлайн-компилятор

Слева на странице находится редактор кода, справа — консоль вывода. Над редактором и консолью — кнопки для установки режимов работы редактора, консоли и компилятора. Под консолью размещена информационная строка компилятора.

Подобных систем в Интернете достаточно много, можно найти именно ту, которая именно вам больше по душе (см. *приложение 2*).

Онлайн-компилятор удобен тем, что можно быстро проверить какое-нибудь свойство языка, не запуская интегрированную среду и не создавая там файл или проект. Профессиональные программисты пользуются этим постоянно.

Задачи и упражнения

1. Запустите интегрированную среду Code::Blocks.
2. Создайте проект `First` (см. *приложение 4*).
3. Напишите в редакторе код:

```
#include <iostream>
using namespace std;
int main()
{
    cout << "2 * 2 = " << 2*2 << endl;
    return 0;
}
```

4. Откомпилируйте проект.
5. Запустите программу на выполнение.

- Урок 2 -

Здравствуй, C++

С момента издания книги «Язык программирования C», написанной Брайаном Керниганом и Деннисом Ритчи, стало традицией начинать любую книгу по программированию с программы «Hello World». Более того, очень многие среды разработки (и не только для C++) при создании проекта сразу включают в проект простейшую программу «Hello World». Среда разработки Code::Blocks — не исключение и умеет делать то же самое.

Первая программа

Создадим проект консольного приложения с именем `Prg02_01`. Если все сделано правильно, то слева в окне проектов мы увидим дерево проекта и в нем — файл с именем `main.cpp`. Откроем этот файл в редакторе Code::Blocks и увидим исходный код на языке C++ программы «Hello World», который показан в листинге 2.1.

Листинг 2.1. Программа «Hello World»

```
#include <iostream>
using namespace std;
int main()
{
    cout << "Hello world!" << endl;
    return 0;
}
```

Программа состоит всего из семи строк, две из которых — это фигурные скобки. Тем не менее понимание даже такой маленькой программы требует знания довольно существенной части основ языка C++.

Запустим компиляцию и выполнение программы (нажмем клавишу <F9>). Так как программа создана средой, то ошибок в ней быть не может. Программа компилируется и запускается — открывается консольное окно (см. *урок 1*, рис. 1.2 и 1.3), в котором мы увидим результат работы программы.

Начнем погружение в язык C++ с детального разбора каждой строки нашей первой программы.

Директива #include

Как повелось со времен создания языка C, многие совершенно необходимые средства разработки (например, средства ввода/вывода или средства для работы со строками) перестали включать в язык программирования в явном виде. Вместо этого такие средства реализуются в виде наборов внешних программ и предоставляются вместе с компилятором языка.

Вся совокупность таких программ составляет *стандартную библиотеку* C++, кратко именуемую STL (Standard Template Library). Чтобы использовать в программе те или иные стандартные средства, их *подключают* с помощью *директивы* `#include`.

Первая строка программы имеет вид:

```
#include <iostream>
```

Английское слово include буквально переводится как «содержать в себе», «включать». Второе слово — это аргумент директивы, который задан в угловых скобках (< и >). Эти скобки — признак *стандартного заголовка*. А имя `iostream` задает используемую в программе часть стандартной библиотеки. В данном случае директива подключает средства *потокового ввода/вывода* (io — сокращение слов input (ввод), output (вывод), stream — поток). Это обеспечивает возможность вывести строку «Hello world!» в консольное окно.

Если убрать строку `#include <iostream>` или написать ее неправильно, то программа просто не скомпилируется.

Using-директива

Вторая строка программы называется *using-директивой*. В принципе эта строка не является обязательной, но среда ее генерирует. Using-директива тоже имеет отношение к стандартной библиотеке. Все имена, которые объявлены в стандартной библиотеке, входят в *стандартное пространство имен* `std`. Все стандартные имена надо писать с префиксом `std::`. Наличие using-директивы позволяет писать их без префикса.

Выясним, какие имена в программе являются стандартными и написаны без префикса. Так как using-директива в принципе не обязательна, то давайте удалим ее из кода и попробуем выполнить программу. Программа не будет выполнена, т. к. возникли ошибки компиляции:

```
error: 'cout' was not declared in this scope
error: 'endl' was not declared in this scope
```

Из этих сообщений становится понятно, что стандартными именами в нашей программе являются имена `cout` (читается си аут) и `endl` (читается эндл), написанные в пятой строке. При отсутствии using-директивы их требуется написать с префиксом. Тогда пятая строка программы будет выглядеть так:

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

Если после такого изменения снова запустить программу на выполнение, то программа выполнится, и мы увидим результат в консольном окне (см. *урок 1*, рис. 1.2 и 1.3).

Стандартных имен в реальных программах бывает очень много, и каждое должно быть написано с префиксом (а то и с несколькими). Это довольно сильно «засоряет» код, его становится труднее читать и понимать. Поэтому программисты обычно стараются как-то «очистить» код от «лишних» символов. Написав в программе using-директиву, мы получаем возможность писать все системные имена без префиксов.

Функция *main*

Функция — это часть программы, имеющая имя. По этому имени функцию можно *вызвать*, т. е. заставить выполнить свою работу. Когда вызванная функция заканчивает работу, она обычно (не всегда) должна возвратить некоторый результат в ту точку, откуда ее вызвали.

При запуске C++-программы первой всегда вызывается функция с именем `main`. Упрощенно можно считать, что эту функцию вызывает операционная система. При вызове функции `main` возвращаемым результатом является некоторое целое число, которое сигнализирует операционной системе, насколько успешно прошла работа.

Слово main переводится с английского как «главный», поэтому и функция с таким именем является *главной функцией* в программе на C++. Каждая программа на языке C++ должна содержать функцию с именем `main`, причем единственную.

В языке C++ имя функции задается в заголовке функции, а действия, которые функция должна выполнить, записываются после нее в теле функции.

Заголовок функции

В третьей строке программы написан *заголовок* главной функции:

```
int main()
```

Заголовок имеет *стандартный* (определенный в стандарте C++) вид:

- сначала указывается *тип* возвращаемого значения, который для главной функции представляет стандартный *целый* тип `int` (сокращение слова integer — целое);
- `main` — обязательное имя главной функции и другое имя писать нельзя;
- «пустые» скобки () после имени `main` показывают, что параметры функции отсутствуют.

Параметры — это объекты программы, которые нужно передать функции для выполнения задачи. Если скобки пустые, значит, никаких дополнительных внешних объектов функции для работы не нужно.

Заголовок функции находится перед телом функции (последовательность выполняемых действий в фигурных скобках), поэтому в нем прописано «обещание» вернуть из функции значение указанного типа. А сам возврат записывается в теле функции.

Тело функции

После заголовка следующие четыре строки вместе с фигурными скобками определяют *тело* функции `main`. В языке C++ (и почти во всех других языках программирования) конструкция в фигурных скобках называется *блоком*. Блоки в C++ используются не только для написания тела функции `main`.

Последовательность действий, которые выполняются во время работы главной функции, записываются на языке C++ в виде набора *операторов,* каждый из которых завершается символом «;» (точка с запятой). Операторы, записанные в теле, выполняются последовательно, с первого и до последнего.

> ***Примечание***
>
> Термин «оператор» используется в русскоязычной компьютерной литературе с момента появления языков программирования. В англоязычной компьютерной литературе для того же понятия используется термин statement, который современные переводчики часто переводят как «инструкция» или «команда». Мы будем придерживаться русской терминологии.

В нашей программе в теле функции в пятой и шестой строках написаны всего два оператора — оператор *вывода*:

```
cout << "Hello world!" << endl;
```

и оператор *возврата* значения:

```
return 0;
```

Эти операторы выполняются в том порядке, как записаны: сначала происходит вывод строки «Hello world!» в консольное окно, а после него выполняется оператор возврата, и программа заканчивает работу.

Оператор вывода

Напомню, что *вывод* — это процесс пересылки данных из программы во внешнюю среду (в данном случае в консоль). Оператор вывода имеет довольно сложную структуру. В нем заданы операция вывода, что выводить, куда выводить и что делать после вывода.

Первым в операторе записано имя `cout`, которое определяет, куда нужно выводить. Имя `cout` — это имя системного объекта, `который` определен в стандартной библиотеке потокового ввода/вывода `iostream`, подключенной директивой `#include`. Этот стандартный объект представляет в программе *выходной поток* данных, который связан с консольным окном.

> ***На заметку***
>
> В стандартной библиотеке потокового ввода/вывода C++ определены и другие объекты, по умолчанию связанные с консольным окном: `cerr`, `clog`.

Как писал создатель языка C++ Бьерн Страуструп в своем учебнике, имя `cout` является аббревиатурой от **c**haracter **out**put stream, что переводится как «поток вывода символов».

Далее в операторе написан двойной знак меньше (<<) — это и есть *операция вывода в поток*. Справа от нее записана *строковая константа* (которую также называют строковым *литералом*) — она и выводится. Строковая константа выводится точно в таком виде, как написана, только без кавычек. Внутри кавычек можно писать почти любые символы, которые есть на клавиатуре (кроме кавычек и символа «\» — обратной косой черты). В том числе без ограничений можно использовать русские буквы.

> ***Примечание***
>
> Вывод (и ввод) русских букв в разных операционных системах имеет свои особенности. Мы более подробно рассмотрим эту проблему далее.

Последняя часть оператора (перед символом «;») содержит код:

```
<< endl
```

Слово endl является сокращением двух английских слов end line, что переводится как «конец строки». Имя `endl` является именем системного манипулятора. *Манипулятор* — это реализованная в стандартной библиотеке функция, которая что-то делает в выходном потоке. Манипулятор `endl` завершает вывод строки и переводит курсор в консоли на следующую строку.

Оператор *return*

Английское слово return переводится как «возврат» или «вернуть». После него записано число `0`. Такие числа в программе называются целыми *константами* (или *литералами*). Это и есть то самое возвращаемое значение, тип `int` которого был «обещан» в заголовке главной функции. Значение возвращается функцией в точку вызова. Упрощенно можно считать, что это число передается операционной системе, и средствами ОС его можно получить и обработать.

Обычно значение `0` означает *нормальное* завершение программы. Это только соглашение, которого стараются придерживаться все программисты в мире. Вы, конечно, можете написать программу, в которой в качестве нормального возвращаемого значения используете значение `1000` или `-1`, но постарайтесь придерживаться общепринятого соглашения, потому что соглашения (поверьте моему опыту) облегчают программисту жизнь.

Кстати, оператор `return` можно не писать в главной функции, но среда его сгенерировала, поскольку в соответствии с принятым соглашением функция `main` должна завершаться оператором возврата.

Пишем комментарии

Мы разобрали каждую строку программы, которую сгенерировала нам среда. Но все это легко забыть, если не записать. И записывать нужно не в тетрадь, а непосредственно в исходном коде. Краткое пояснение по каждой строке — это «шпаргалка», которая поможет запомнить (и вспомнить через какое-то время) все изложенное выше.

Программисты с самых первых дней компьютерной эры поняли, что в работе с большими и сложными проектами подобные «шпаргалки» необходимы как воздух. Поэтому во всех языках программирования без исключения имеются средства, которые позволяют писать заметки в коде, чтобы не забывать, что и в каком порядке делает программа. Эти заметки называются *комментариями*.

> ***История из жизни***
>
> Начинающие программисты по неопытности никогда не пишут комментариев в программах. Как-то раз ко мне пришел студент, чтобы проконсультироваться по задаче, программированием которой он тогда занимался. Он принес код и показал мне место в коде, где ему требовалась консультация. Комментариев в программе не было совсем. Я спросил его, почему он написал код именно так, а не иначе. Он не смог вспомнить, т. к. это было «давно» — две недели назад. Тогда я показал ему программу, написанную мною 7 лет назад. В моем коде было много комментариев, и я легко объяснил ему все нюансы программы через 7 лет после ее написания. Пишите комментарии всегда!

В комментариях можно писать любые символы, доступные на клавиатуре, в том числе и русские буквы.

В C++ есть *многострочный* комментарий, который появился еще в языке C.

Такой комментарий начинается с символов `/*` (косая черта и звездочка) и *заканчивается* символами `*/` (звездочка и косая черта). Эти символы были выбраны разработчиками C не просто так: на цифровой клавиатуре они расположены рядом и их быстро можно набрать даже вслепую.

Начало и конец комментария обычно располагаются на разных строках:

```
/*      Это
        многострочный
        комментарий
*/
```

Но можно написать начало и конец и на одной строке:

```
/* Первая программа на C++ */
```

Получился однострочный комментарий. Однако для такого комментария в C++ существует отдельная конструкция. Однострочный комментарий всегда начинается с символов `//` (двойная косая черта). Понятно, что две косые тоже можно вслепую набрать на цифровой клавиатуре. Начало строчного комментария может быть в любом месте строки, он не обязательно должен начинаться с начала строки. Такой комментарий продолжается до конца строки (а в конце мы обычно нажимаем клавишу <Enter>).

Прокомментируем строки нашей программы. Код с комментариями показан в листинге 2.2.

Листинг 2.2. Программа «Hello World» с комментариями

```
/* Программа "Hello World" на C++
   Сгенерирована средой Code::Blocks
*/
```

```
#include <iostream>                          // директива препроцессора
using namespace std;                         // объявление пространства имен
// программы на C++ начинают выполнение с функции main
int main()                                   // заголовок главной функции
/* тело главной функции – блок */
{                                            // начало блока
    cout << "Hello world!" << endl;          // вывод строки
    return 0;                                // нормальное завершение
}                                            // конец блока
```

Я написал комментарии почти в таком же стиле, как пишу в своих настоящих программах на C++. В строчных комментариях я после двух косых обычно набираю пробел и всегда выравниваю комментарии по вертикали — они так лучше читаются.

Комментарии бывают очень полезны, когда нужно удалить из программы строку или несколько строк. Можно их просто стереть, однако бывает и так, что стертые строки могут потом снова понадобиться. Или по ошибке вообще стерта не та строка. Комментарий позволяет «выключить» участок кода, не удаляя его, чтобы потом легко было вернуть. Но злоупотреблять этим не следует, т. к. когда закомментированных строк много, это очень затрудняет чтение и понимание программы.

Копнем глубже

Займемся более детальным исследованием сгенерированной программы: поэкспериментируем с формой записи программы и отдельных операторов.

Формат кода

Программа сгенерирована средой в типичном формате (стиле), который наиболее распространен при написании кода на языке C++. Но C++ позволяет писать программу практически в любом формате. Например, допустимо написать главную функцию в таком виде:

```
int main(){ cout << "Hello world!" << endl; return 0; }
```

Однако, если операторов много, то, конечно, их лучше писать «в столбик» — такой код гораздо лучше читается.

В C++ разрешается писать операторы на нескольких строках. Например, можно написать оператор вывода таким образом:

```
cout
<< "Hello world!" << endl;
```

Когда вы будете писать код своими руками, то можете делать так, как вам удобнее и понятнее. Программисты обычно как-то структурируют код — так его проще читать (листинг 2.3).

Листинг 2.3. Программа «Hello World» в свободном формате

```
#include <iostream>
using namespace std;
int main(){                                    // блок в стиле языка Go
    cout << "Hello world!" << endl;
    return 0;
}
```

Обратите внимание на расположение фигурных скобок тела функции `main`. Редактор кода Code::Blocks вставляет сразу две скобки {} в одной строке. Поэтому такой формат блока получается при нажатии клавиши <Enter> между ними.

Директивы

Первые две строки называются директивами. Однако это совершенно разные директивы. Using-директиву можно написать на нескольких строках:

```
using
namespace std;
```

А директиву `#include` нельзя написать на двух строках. Если мы попробуем это сделать:

```
#include
<iostream>
```

то программа не выполняется, и возникает ошибка компиляции. Отсюда следует, что директива `#include`, вообще говоря, не входит в состав языка C++ (поэтому ее нельзя написать на двух строках).

Директива `#include` является директивой препроцессора. *Препроцессор* — это системная программа, которая обрабатывает исходный код программы до компилятора и передает получившийся текст компилятору для трансляции.

Таким образом, хотя директива препроцессора `#include` не является частью языка C++, без нее пока не может обойтись ни одна программа на C++. С помощью директивы `#include` к программе могут быть подключены любые средства стандартной библиотеки. Например, для работы со строками надо написать в программе директиву `#include <string>`.

Существуют и другие директивы препроцессора, например `#define`. Все директивы препроцессора начинаются с символа «#» (решетка, диез).

Алфавит и зарезервированные слова

На языке программирования пишут тексты, поэтому язык программирования имеет алфавит. Алфавит естественного языка (например, русского) обычно содержит только буквы. Алфавит языка программирования включает все символы, которые можно написать в программе: буквы, цифры, скобки, знаки препинания (запятая,

точка, знак вопроса, восклицательный знак, двоеточие, точка с запятой) и множество специальных знаков (например, таких как знак «&» — амперсанд).

Буквы в языках программирования английские, поэтому слова языка — тоже английские. И в любом языке определен набор слов, которые называются *зарезервированными* (reserved) или *ключевыми* (keywords) словами. В языке C++ таких слов более 90, и они пишутся только маленькими буквами. В редакторе Code::Blocks они обычно выводятся синим цветом. В программе «Hello World» мы видим четыре синих слова: `using`, `namespace`, `int`, `return`.

Это и есть ключевые слова.

Модифицируем первую программу

Создадим новый проект `Prg02_02` и модифицируем программу «Hello World»: напишем вывод строки «Hello world!» в рамочке, например:

```
+--------------+
| Hello world! |
+--------------+
```

Это делается довольно просто. Нужно добавить в тело главной функции два оператора вывода перед и после вывода приветствия и немного поправить саму строку приветствия (листинг 2.4).

Листинг 2.4. Программа вывода приветствия в рамочке

```
#include <iostream>
using namespace std;
int main(){
    cout << "+--------------+" << endl;
    cout << "| Hello world! |" << endl;
    cout << "+--------------+" << endl;
    return 0;
}
```

Все очевидно и понятно: операторы вывода отличаются только литералом. Первый и третий операторы выводят верхнюю и нижнюю строку рамочки, а второй осуществляет вывод приветствия, у которого слева и справа приписан символ «|» (вертикальная черта).

Выполнив программу (нажав клавишу <F9>), увидим в консольном окне результат работы (рис. 2.1).

Все работает правильно. Однако опытный программист, посмотрев исходный код, скажет, что его можно сократить. И язык C++ помогает это сделать. Например, вывод приветствия в рамочке можно сделать в одном операторе.

Модификация программы вывода приветствия в рамочке состоит в том, что мы удалим лишние операторы вывода (листинг 2.5).

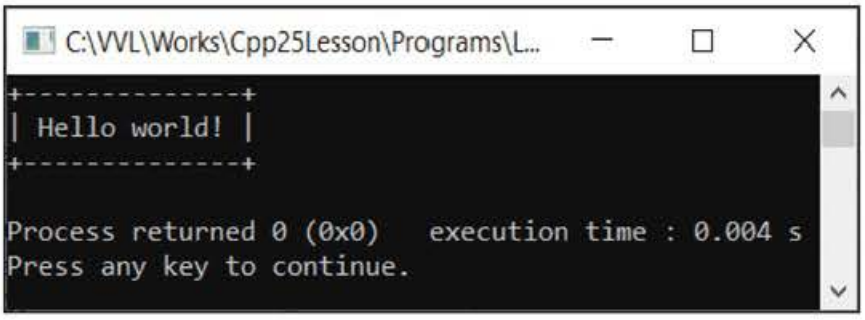


Рис. 2.1. Вывод приветствия в рамочке

Листинг 2.5. Программа вывода приветствия в рамочке в одном операторе

```
#include <iostream>
using namespace std;
int main(){
    cout << "+--------------+" << endl
         << "| Hello world! |" << endl
         << "+--------------+" << endl;
    return 0;
}
```

Удалены два слова `cout` и соответствующие им «;» (точки с запятой). Остался единственный оператор вывода, в котором заданы три операции вывода строк и три операции для работы манипуляторов `endl`. Оператор записан в трех строках для того, чтобы его было проще читать и понять.

Преобразования программы упростили код, он стал короче, его проще читать, но при этом работа программы не изменилась. Такие преобразования кода, не изменяющие поведение программы, называются *рефакторингом*.

> ***На заметку***
>
> Рефакторинг — это основная работа программиста с кодом.

Новый код программист пишет один раз, а *рефакторит* его многократно. И каждый раз нужно проверять, получаются ли те же самые результаты.

Выполнив программу (нажав клавишу <F9>), увидим в консольном окне тот же результат (см. рис. 2.1).

Специальные символы

Выше было упомянуто, что внутри кавычек в строковом литерале можно писать почти любые символы, кроме кавычки и обратной косой черты («\»). Но что делать, если нам необходимо вставить в сообщение цитату, которая сама должна быть написана в кавычках? Например, нам необходимо вывести строку «`This is "quotation"`».

Заключаем строку в кавычки и помещаем в оператор вывода:

```
cout << "This is "quotation"" << endl;
```

Однако компилятор обнаружит в таком операторе ошибку: после строкового литерала `"This is "` должна следовать новая операция вывода, а написано слово `quotation`, после которого заданы две кавычки, являющиеся пустой строкой. Естественно, компилятор «ругается».

> ***На заметку***
>
> Пустая строка — это строка, в которой отсутствуют символы, кроме кавычек.

В C были придуманы специальные символы (табл. 2.1), которые называют *escape-последовательностями*. Каждый специальный символ записывается двумя обычными символами, первым из которых должна быть обратная косая черта («\»). Второй символ определяет, что будет происходить в выходном потоке (табл. 2.1).

Таблица 2.1. *Специальные символы*

Символ	Описание
\\	Вставка обратной косой черты в выходной поток
\"	Вставка кавычки в выходной поток
\n	Перевод курсора в консольном окне в начало следующей строки
\r	Перевод курсора в консольном окне в начало текущей строки
\t	Горизонтальная табуляция

Специальные символы унаследованы языком C++ от C и впоследствии были включены во многие языки программирования. В таблице показаны только наиболее часто используемые специальные символы.

Таким образом, чтобы вывести строку «`This is "quotation"`», нужно написать в программе следующий оператор вывода:

```
cout << "This is \"quotation\"" << endl;
```

Тогда в консольном окне будет выведено сообщение:

```
This is "quotation"
```

Необрабатываемые строковые литералы

Накопленный опыт использования escape-последовательностей (в том числе и в других языках программирования) показал, что их применение частенько приводит к ошибкам. Поэтому в C++11 была добавлена возможность писать строковые константы, в которых специальные символы не обрабатываются. Такой литерал так и называется: необработанный (или необрабатываемый) строковый литерал (raw string literal). Еще его называют «сырым» литералом.

В программе на C++ «сырой» литерал записывается следующим образом:

```
R"(<строка символов>)"
```

Например, вывести строку с цитатой можно так:

```
cout << R"(This is "quotation")" << endl;
```

В консольном окне будет выведены символы, написанные в скобках:

```
This is "quotation"
```

Если нам требуется, чтобы все сообщение было выведено в кавычках, то нужно так и написать:

```
R"("This is "quotation"")"
```

В консоли сообщение появится в кавычках:

```
"This is "quotation""
```

А полное имя файла в Windows записывается без дополнительных обратных косых:

```
R"(C:\Program Files\CodeBlocks\codeblocks.exe)"
```

И выводится точно в том виде, как записано.

В дальнейшем для краткости будем называть такие литералы *R-литералами*.

Снова рефакторинг

С помощью специальных символов можно еще укоротить код, избавившись от манипуляторов `endl` и оставив одну операцию вывода (листинг 2.6).

Листинг 2.6. Программа вывода приветствия в рамочке без манипуляторов

```
#include <iostream>
using namespace std;
int main(){
    cout << "+--------------+\n"
            "| Hello world! |\n"
            "+--------------+\n";
    return 0;
}
```

Выполнив программу (нажав клавишу <F9>), увидим в консольном окне все тот же результат (см. рис. 2.1).

В этой версии программы мы видим один оператор вывода с единственной операцией вывода <<, поэтому и выводимый строковый литерал должен быть один. Однако код выглядит так, как будто выводится три строки. На самом деле вместо трех строк выводится одна длинная строка:

```
"+--------------+\n| Hello world! |\n+--------------+\n"
```

Но если ее записать в таком виде, то легко сделать ошибку (написать лишний минус или пропустить). Язык C++ предоставляет возможность разбивать выводимую строку на части, и это гораздо лучше читается. В разделенной на части строке ошибки видны сразу, и они тут же исправляются.

На заметку

R-литералы тоже можно разбивать на части, записывая каждую часть в виде отдельного R-литерала.

Вместо манипуляторов в строке в нужных местах записан специальный символ `\n`, который переводит курсор в консольном окне на начало следующей строки.

Русские буквы в консоли

До сих пор все сообщения в консольное окно были на английском языке. Однако в России нужно выводить сообщения по-русски. В системах типа Linux, версиями которой являются российские дистрибутивы МОС и «Альт», проблем с выводом русских слов в окно терминала нет никаких. Достаточно в операторе вывода (см. листинг 2.1) заменить английские слова на русские «Привет, мир!» и запустить программу (рис. 2.2).

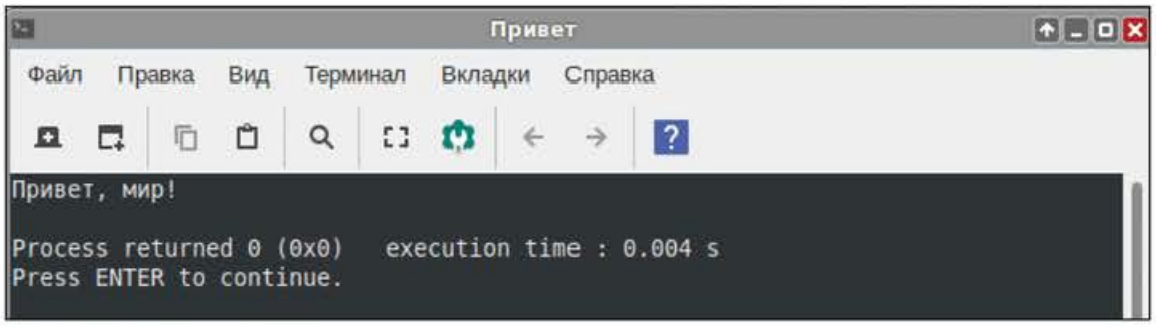


Рис. 2.2. Вывод русских символов в окно терминала в системе «Альт»

Однако в системе Windows выводятся какие-то «крякозябры»:

```
╧ЕштхЄ, ьшЕ!
```

Проблема состоит в различии *кодировок* символов в нашей программе и в консольном окне. Когда среда разработки Code::Blocks работает в Windows, весь исходный код программы (в том числе и строковый литерал «Привет, мир!») в редакторе среды набирается в кодировке ср-1251. В консольном же окне установлена по умолчанию кодировка ср-866. В обеих кодировках поддерживается русский алфавит, но по-разному (одна и та же русская буква в разных кодировках закодирована разными числами).

На заметку

В системах Linux в окне терминала установлена кодировка UTF-8, которая поддерживает кириллицу. И редактор среды разработки Code::Blocks в Linux тоже работает в кодировке UTF-8.

Поэтому для вывода сообщений в консольное окно на русском языке в системе Windows нужно модифицировать генерируемую средой программу.

Создадим проект `Prg02_03` и порефакторим код.

Вывод по-русски в системе Windows

В системе Windows существует множество способов русифицировать вывод в консоль. Воспользуемся стандартной функцией C/C++ `system`. Аргументом функции является строка — команда *командного процессора* Windows.

> ***На заметку***
>
> В любой ОС имеется командный процессор, а часто и несколько. Командный процессор «понимает» команды. Команды вводятся в консоли (терминале).

Нужно написать в коде *вызов* функции `system`:

```
system("chcp 1251>nul");
```

Эта команда `chcp` (**ch**ange **c**ode **p**age, изменить кодовую страницу) устанавливает в консоли кодировку cp-1251. Код программы представлен в листинге 2.7.

Листинг 2.7. Вывод русских букв в консоль *system*

```
#include <iostream>
#include <cstdlib>                    // стандартная библиотека C
using namespace std;
int main(){ system("chcp 1251>nul");  // установка кодировки cp-1251
    cout << "Привет, мир!" << endl;
    return 0;
}
```

И в этом коде обратите внимание на вторую директиву `#include`. Язык C++ унаследовал от языка C его стандартную библиотеку. Чтобы отличать старые средства от новых, было принято соглашение, что все имена системных файлов языка C должны начинаться на букву «c» (английская «си»). Поэтому вторая директива подключает стандартную библиотеку языка C, в которой содержится функция с именем `system`.

Этот способ обеспечивает как вывод русских букв в консольное окно, так и ввод их с клавиатуры.

В дальнейшем мы уже не будем писать в коде русификацию. Те читатели, которые работают в системе Windows, будут делать это самостоятельно.

Немного арифметики

До сих пор все наши программы выполняли самую простую работу — выводили сообщения в консоль. Теперь давайте напишем программу, которая будет что-то вычислять и показывать нам результат вычисления. В школе начинают учить арифметику, складывая и вычитая маленькие целые числа. Вспомним типичные задачи 1-го класса.

У одного ученика было 8 конфет, у другого ученика было 6 конфет. Первый ученик отдал второму 1 конфету. Сколько конфет стало у учеников?

Напишем программу, которая вычисляет количество конфет у учеников (листинг 2.8).

Листинг 2.8. Программа «Сложение и вычитание»

```
#include <iostream>
using namespace std;
int main(){
    cout <<"У 1-го ученика стало 8-1 = " <<(8 - 1) <<" конфет\n";
    cout <<"У 2-го ученика стало 6+1 = " <<(6 + 1) <<" конфет\n";
    return 0;
}
```

Операторы вывода написаны в единообразном виде: выводится поясняющая фраза, потом выводится арифметическое выражение в скобках и после него — завершающее слово со специальным символом. Самое важное в каждом операторе — арифметическое выражение. Именно в нем вычисляется то, что является ответом в задачах. Выражение написано в скобках. Арифметические операции, как мы знаем, имеют *приоритет*, который определяет порядок вычисления (умножение выполняется раньше сложения, если без скобок). Поэтому выражение заключено в скобки, чтобы оно сначала было вычислено, а потом выведен результат вычисления.

Выполним программу и увидим в консоли результаты вычислений (рис. 2.3). Как видим, программа правильно вычисляет ответы к задачам.



Рис. 2.3. Вывод программы «Сложение и вычитание»

После изучения сложения и вычитания переходят к изучению таблицы умножения. Как и в школе, мы начнем с таблицы умножения на 2 и напишем программу, которая выводит полную таблицу с 1 до 10 (листинг 2.9).

Листинг 2.9. Программа «Таблица умножения на 2»

```
#include <iostream>
using std::cout;                                    // using-объявление
int main(){
   cout << "Таблица умножения на 2\n";
   cout <<"2 * 1 = "<<(2*1)<<'\n'; cout <<"2 * 2 = "<<(2*2)<<'\n';
   cout <<"2 * 3 = "<<(2*3)<<'\n'; cout <<"2 * 4 = "<<(2*4)<<'\n';
   cout <<"2 * 5 = "<<(2*5)<<'\n'; cout <<"2 * 6 = "<<(2*6)<<'\n';
```

```
    cout <<"2 * 7 = "<<(2*7)<<'\n'; cout <<"2 * 8 = "<<(2*8)<<'\n';
    cout <<"2 * 9 = "<<(2*9)<<'\n'; cout <<"2 * 10 = "<<(2*10)<<'\n';
    return 0;
}
```

В программе мы видим пару новшеств. Во-первых, вместо using-директивы использована другая конструкция, которая называется *using-объявлением*. Аргументом using-объявления является конкретное стандартное имя с префиксом. Using-объявление применяют в тех случаях, когда в программе мало стандартных имен. Тогда не нужно объявлять полное пространство имен, а можно объявить каждое стандартное имя отдельно.

Во-вторых, в операторах вывода специальный символ `\n` написан не в кавычках, а в апострофах (одиночная кавычка). Язык C++ унаследовал из C и строковые литералы (в кавычках), и символьные литералы, которые пишутся в апострофах, например `'a'`, `'1'` и т. п. Специальный символ `'\n'`, хотя и представлен в коде двумя символами, является одиночным символом.

Ну, и вы уже поняли, что операция умножения представлена звездочкой. Все выражения, как и в предыдущем примере, заключены в скобки.

Запуск программы приводит к появлению в консольном окне 11 строк:

```
Таблица умножения на 2
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
2 * 4 = 8
2 * 5 = 10
2 * 6 = 12
2 * 7 = 14
2 * 8 = 16
2 * 9 = 18
2 * 10 = 20
```

Все вычисляется правильно. Заметим, что такая программа уже может быть реально полезна, если кому-то нужно учить таблицу умножения.

Задачи и упражнения

1. Выведите на экран любимое четверостишие.

2. Сократите количество операторов в программе из листинга 2.9.

3. Напишите программу вывода таблицы умножения на 5.

4. Выведите четверостишие внутри рамки из звездочек, например:

```
******************************
**  Аист с нами прожил лето   **
**  А зимой гостил он где-то. **
```

```
**  Воробей влетел в окно     **
**  Воровать у нас пшено.     **
**              С.Я. Маршак   **
********************************
```

5. Напишите программу, изображающую домик, например такой (рис. 2.4):



Рис. 2.4. Пример домика

- Урок 3 -

Погружаемся в C++

В предыдущем уроке мы научились выводить сообщения в консольное окно по-английски и по-русски разнообразными способами. Но программы почти всегда должны как-то взаимодействовать с пользователем. Очень многие программы работают таким образом: программа запрашивает и получает какие-то данные от пользователя, вычисляет что-то, используя эти данные, и выводит сообщение с результатом пользователю. Если программа маленькая, то вычисление возможно выполнить непосредственно в процессе вывода сообщения (как в программе «Таблица умножения на 2», см. листинг 2.9). Но для больших программ вычисления являются обычно значительной частью.

Ввод данных

Начнем с простого: напишем программу, приветствующую пользователя по имени. Приветствие нужно вывести на русском языке.

Создать такую программу довольно просто: модифицируем (отрефакторим) генерируемую программу приветствия (листинг 3.1).

Листинг 3.1. Программа приветствия по имени

```
#include <iostream>
#include <string>                       // работаем со строками
//using namespace std;                  // выявление стандартных имен
int main(){
    std::cout << "Как вас зовут ?" << std::endl;    // запрос имени
    // ввод имени пользователя
    std::string name;                   // переменная для хранения имени
    std::cin >> name;                   // ввод имени пользователя
    // вывод приветствия
    std::cout << "Приветствую вас, " << name << "!" << std::endl;
    return 0;
}
```

В программе закомментирована using-директива, чтобы продемонстрировать, насколько часто приходится писать префикс стандартных имен. Даже в такой ма-

ленькой программе используется шесть стандартных имен, и общее количество символов во всех префиксах равно 30. А using-директива занимает всего 20 символов. Очевидно, что предпочтительнее писать using-директиву.

Первый оператор нашей программы выводит вопрос пользователю. А вот следующие два оператора нужно разобрать детально.

Имя пользователя требуется ввести с клавиатуры. После ввода его надо где-то сохранить, поэтому в программе объявлена *переменная*. Переменная — это *объект*, который имеет название (имя переменной).

> ***На заметку***
>
> В языке C++ существуют безымянные объекты. Например, литералы.

В языках программирования имя переменной называется идентификатором. *Идентификатор* — это последовательность букв (больших и маленьких) и цифр, обязательно начинающаяся с буквы. Практически во всех языках к буквам относят еще и символ «_» (подчеркивание). В C++ подчеркивание используется в именах из нескольких слов вместо пробела между словами.

В языке C++ различаются большие и маленькие буквы, поэтому имена `ab`, `Ab`, `aB`, `AB` считаются различными. Количество символов в идентификаторе определяется конкретной реализацией компилятора.

Имя (идентификатор) нашей переменной — `name`. Программист может назвать переменную как угодно (например, `aa1234567890` или `A_B_C`), но гораздо лучше давать переменным осмысленные имена. Тогда при чтении кода становится понятнее, что и как делает программа.

Переменная имеет не только название, но и тип. Любой объект в C++, даже безымянный, имеет тип. Например, литералы являются безымянными объектами, но любой литерал имеет тип (ранее мы уже встречались со строковыми, символьными и целыми литералами). Нужно заметить, что тип символьного литерала отличается от типа строкового литерала.

Тип определяет, какие значения будут храниться в этой переменной и какие операции можно применять к значениям этого типа. Например, числа можно складывать, вычитать, делить и умножать. А без операций числа и не нужны.

> ***На заметку***
>
> C++ является статически типизированным языком программирования. Это означает, что тип переменной, заданный в объявлении, потом изменить нельзя.

Имя пользователя — это строка символов (например, «Никита»), поэтому тип переменной задан как `std::string`.

Тип написан с префиксом, поэтому очевидно, что этот тип реализован в стандартной библиотеке. Во второй строке программы прописан стандартный заголовок библиотеки для работы со строками:

```
#include <string>
```

Чтобы переменную можно было использовать в программе, она должна быть объявлена в *операторе объявления*:

```
std::string name;                    // переменная для хранения имени
```

Этот оператор *определяет* поименованную переменную, хотя и называется оператором объявления. Вообще говоря, в C++ объявления и определения различаются. Например, using-объявление только объявляет стандартное имя, но не определяет его (см. листинг 2.9). Мы разберемся с разнообразными объявлениями по ходу изучения C++.

Во время работы программы для переменной выделяется память, и туда помещается *начальное значение.* Для строк начальным значением является пустая строка (не содержит символов). Эта операция присвоения начального значения называется *инициализацией.* В нашей программе в операторе объявления переменной инициализация выполняется неявно. Мы научимся инициализировать переменные явным образом в дальнейшем.

Таким образом, каждая переменная, объявленная в программе, имеет имя и для нее задан тип. В разные моменты времени в переменной могут храниться разные значения, однако имя и тип изменить нельзя. В общем случае имена всех переменных должны быть разные, однако в C++ есть исключения из этого правила.

Оператор объявления переменной может быть написан в коде там, где эта переменная впервые потребуется. Но использовать переменную в других операторах можно только после объявления (определения).

> ***НА ЗАМЕТКУ***
>
> В языке C++ применяется принцип: сначала объяви (имя), потом используй. Этот принцип сейчас действует практически во всех языках программирования.

Следующий оператор после объявления переменной является оператором ввода строки символов (имени пользователя) из *входного потока* символов:

```
std::cin >> name;                    // ввод имени пользователя
```

Идентификатор `cin` (читается «си ин») — это имя системного объекта, который представляет в программе входной поток символов (**c**haracter **in**put stream) и по умолчанию связан с клавиатурой.

Этот оператор, как и оператор вывода, является оператором-выражением, в котором использована операция ввода (>>). Программа при выполнении этого оператора останавливается и ждет от пользователя нажатия каких-то клавиш. Нужно набрать свое имя и нажать клавишу <Enter> для завершения ввода. Программа преобразует последовательность нажатых клавиш в строку и помещает ее в переменную `name`.

Отметим две важных особенности ввода из входного потока. Во-первых, любая последовательность символов, набираемая на клавиатуре, должна обязательно завершаться нажатием клавиши <Enter>.

Во-вторых, если вы набрали на клавиатуре имя и фамилию (с пробелом между ними), то в переменную `name` попадет только имя (последовательность символов до

пробела). Так реализована операция ввода из входного потока. Ввод строки с пробелами делается другими средствами.

Следующий оператор выводит в консоль приветствие, состоящее из трех частей: литерала «Приветствую вас, », переменной `name` и символа «!». При выводе переменной из нее извлекается значение (имя пользователя, помещенное туда при вводе) и помещается в выходной поток (рис. 3.1).

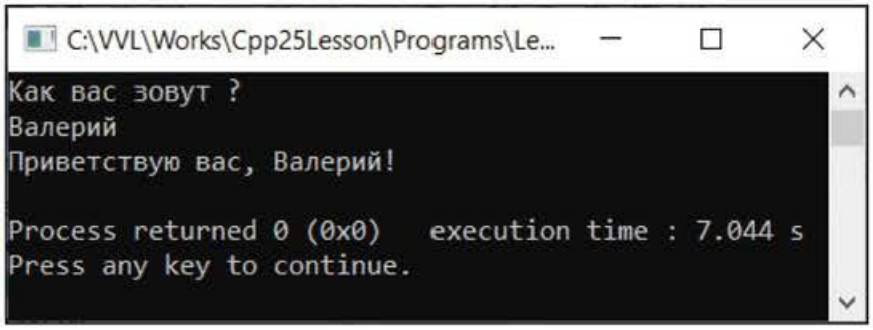


Рис. 3.1. Работа программы, приветствующей пользователя по имени

Работаем со строками

Для более близкого знакомства со строками и с возможностями ввода-вывода давайте напишем программу, которая выведет приветствие пользователю в рамочке, причем пользователь задаст имя и фамилию. Рамочку сделаем из звездочек. Вывод должен выглядеть примерно так:

```
**************************
| Привет, Валерий Лаптев! |
**************************
```

Немедленно возникает проблема: у разных пользователей имена и фамилии содержат разное количество символов, поэтому приветствие для каждого пользователя получается разного размера. Это означает, что количество звездочек в рамочке сверху и снизу нужно как-то вычислять по общему количеству символов приветствия.

> ***На заметку***
> Количество символов в строке называется ее длиной, или размером.

Не торопитесь писать код. Прежде нужно точно сформулировать, что должна делать программа и в какой последовательности. Поэтому попробуйте сами написать алгоритм программы самостоятельно, а потом сравните его с моим.

Алгоритм на псевдокоде выглядит так:

```
1.Запросить у пользователя имя и фамилию.
2.Ввести имя и фамилию.
3.формировать приветствие (вторая строка в рамочке).
4.Вычислить количество символов в приветствии.
5.Сформировать одну строку звездочек нужного размера.
6.Вывести три строки: звездочки, приветствие, звездочки.
```

Код, который я написал по этому алгоритму, представлен в листинге 3.2.

Листинг 3.2. Программа приветствия пользователя в рамочке

```
#include <iostream>
#include <string>
using namespace std;
int main(){
    cout << "Как вас зовут ?\n>";              // запрос имени пользователя
    // ввод имени пользователя
    string name;               // переменная для хранения имени
    string surname;            // переменная для хранения фамилии
    cin >> name >> surname;    // ввод имени и фамилии
    // создаем приветствие
    const string greeting = "| Привет, "+ name +" "+ surname +"! |";
    // создаем строку звездочек
    const string asterisks(greeting.length(), '*');
    // вывод приветствия в рамочке
    cout << asterisks <<'\n'    // звездочки
         << greeting  <<'\n'    // приветствие
         << asterisks <<'\n';   // звездочки
    return 0;
}
```

Первый шаг алгоритма нам уже знаком — это оператор вывода. Обратите внимание на приглашение ввести имя: последний символ в нем — знак «больше» (>). Это обычная практика вывода приглашений.

Далее разберемся с шагом 2. В программе объявлены две переменные:

- `name` — для хранения имени пользователя;
- `surname` — для хранения его фамилии.

Я написал два оператора объявления, но C++ разрешает объявлять в одном операторе несколько переменных:

```
string name, surname;
```

Оператор ввода написан один, и в нем задан ввод двух переменных: сначала вводится имя пользователя, потом его фамилия. При вводе имя и фамилия набираются в одной строке и разделяются пробелом, например:

```
Валерий Лаптев
```

В конце нажимается клавиша <Enter> один раз.

Можно написать ввод имени и фамилии в отдельных операторах:

```
cin >> name;
cin >> surname;
```

Но и в этом случае можно набирать имя и фамилию в одной строке через пробел, нажимая клавишу <Enter> один раз в конце.

Переходим к шагу 3. В следующей строке объявлена переменная `greeting` (приветствие), но объявление имеет очень важные особенности. Первое: в операторе задана явная инициализация с помощью *операции* присваивания (символ «равно» (=)). Слева от знака присваивания указана переменная, справа — строковое выражение, которое вычисляется и присваивается переменной. В выражении указана операция сложения (знак «+»), которая для строк имеет совсем другой смысл. «Сложение» строк является операцией «сцепление» (*конкатенация*). Все составные части выражения сцепляются в одну строку и помещаются в переменную `greeting`.

Здесь мы видим *перегрузку* операции сложения. Говорят, что операция имеет другую *семантику*, хотя используется тот же знак операции.

> ***На заметку***
>
> Перегрузка операций в C++ встречается регулярно. Более того, мы научимся определять нужные нам перегрузки (см. *урок 17*).

Второй особенностью оператора объявления является наличие слова `const`. С одной стороны, это слово информирует читателя кода о неизменности значения переменной во время работы программы; с другой стороны, это слово задает прямой запрет изменения переменной; и наконец, наличие слова `const` требует явной инициализации переменной. Объявление обычной переменной без слова `const` явной инициализации не требует.

Инициализирующее значение присваивается переменной при объявлении, и в дальнейшем это значение в переменной изменить нельзя. Поэтому такие переменные называются *поименованными константами*.

Следующие шаги алгоритма (4 и 5) заданы в одном операторе объявления, где определена другая поименованная константа — `asterisks` (звездочки). Константа инициализируется строкой звездочек, но совершенно другим способом по сравнению с константой `greeting`.

Выражение инициализации записано в круглых скобках и состоит из двух выражений через запятую. Первое выражение — это вычисление нужного количества звездочек. Если дословно перевести эти слова, то получится «приветствие.длина()». Круглые скобки после слова length показывают, что для объекта `greeting` вызвана функция `length`, вычисляющая длину строки (количество символов). Второе выражение — это символьный литерал, из которого должна формироваться строка рамочки. Вместо звездочки можно использовать любой другой символ.

Здесь мы впервые столкнулись с формой записи, которая используется для вызова функций в объектно-ориентированном программировании. Напомню, что тип (данных) — это множество значений и набор операций, которые можно выполнять с этими значениями. Тип `string` реализован в STL, и для него там же реализован набор операций. С двумя из них мы познакомились в нашей программе приветствия: операция сцепления и операция вычисления длины строки. Эта вторая операция представлена функцией `length`.

На заметку

В объектно-ориентированном программировании функции, реализованные для какого-нибудь типа данных, называются методами. В языке C++ их называют функциями-членами, но мы будем придерживаться традиционного для всех остальных языков названия «метод».

В библиотеке реализован и ее аналог `size`, поэтому оператор объявления константы `asterisks` можно было бы написать и таким образом:

```
const string asterisks(greeting.size(), '*');
```

Результат работы программы от такой замены не изменяется.

И наконец, последний пункт нашего плана написан в последнем операторе вывода в три строки.

Запустив программу, увидим результат в консольном окне (рис. 3.2). Все работает правильно.



Рис. 3.2. Вывод приветствия пользователя в рамочке

Но давайте запустим программу вновь и введем фамилию и имя в обратном порядке. Программа отработает «без возражений», и в консоли появится приветствие в рамочке:

```
**************************
| Привет, Лаптев Валерий! |
**************************
```

Очевидно, что фамилия попала в переменную `name`, а имя — в переменную `surname`, но программа этого «не заметила» и выполнила обычные действия. Если мы вместо имени и фамилии введем через пробел числа `222` и `333`, то программа и в этом случае выполнится и выведет приветствие в рамочке:

```
********************
| Привет, 222 333! |
********************
```

В самом деле, ведь «222» и «333» — это тоже строки символов. Поэтому при разработке программы недостаточно писать код, который правильно работает с правильными данными. Ваш код должен проверять данные, получаемые от пользователя (из внешней среды), и программа должна что-то делать, если данные были некорректные.

На заметку

Обработка некорректных данных — значительная часть разработки программы в промышленном программировании.

Мы в дальнейшем научимся писать код, проверяющий данные.

Снова таблица умножения

В предыдущем уроке мы написали простую программу, которая выводит в консоль таблицу умножения на 2 (см. листинг 2.9). Понятно, что таким же способом мы можем создать программы, которые будут нам выводить и другие таблицы умножения: на 3, 4 или на 9.

Однако таблица умножения на 3 отличается от таблицы умножения на 2 только множителем. Этот множитель можно запрашивать у пользователя и вводить с клавиатуры. Для этого нужно в программе объявить переменную и написать оператор ввода. Множитель — это целое число, поэтому переменная должна быть целого типа (листинг 3.3).

Листинг 3.3. Программа «Универсальная таблица умножения»

```
#include <iostream>
using namespace std;
int  main()
{   int k;                                  // объявление переменной
    cout <<"Введите множитель от 1 до 9:_";
    cin >> k;                               // ввод числа
    cout<< k <<"*1="<<k*1 <<'\n' << k <<"*2="<<k*2 <<'\n'
        << k <<"*3="<<k*3 <<'\n' << k <<"*4="<<k*4 <<'\n'
        << k <<"*5="<<k*5 <<'\n' << k <<"*6="<<k*6 <<'\n'
        << k <<"*7="<<k*7 <<'\n' << k <<"*8="<<k*8 <<'\n'
        << k <<"*9="<<k*9 <<'\n' << k <<"*10="<<k*10 <<'\n';
   return 0;
}
```

В этой программе оператор:

```
int k;
```

объявляет переменную целого типа с именем `k`. Программисты предпочитают говорить и писать короче: целая переменная `k`. Тип пишется без префикса, т. к. это встроенный тип. *Встроенные* типы определены непосредственно в самом языке программирования (встроены в язык). Такие типы обычно называются фундаментальными, основными, базовыми, элементарными или простыми. Названия встроенных типов в C++ — зарезервированные слова.

Напомню, что тип определяет не только множество значений (тип `int` — это множество целых чисел), но и набор операций, которые можно применять к значениям

этого типа (числа складывают, вычитают, умножают и делят). Арифметические и другие операции со встроенными типами тоже являются частью языка C++.

Если мы запустим программу, то в консольном окне появится сообщение:

```
Введите множитель от 1 до 9: _
```

Программа остановится, ожидая от нас ввода целого числа. Нажмем клавишу с цифрой 5 и клавишу <Enter> и увидим на экране следующие десять строк:

```
5*1=5
5*2=10
5*3=15
5*4=20
5*5=25
5*6=30
5*7=35
5*8=40
5*9=45
5*10=50
```

Каждая строка оператора вывода работает так: сначала на экран выводится значение переменной `k` (в данном случае число 5), затем без пропуска — строковый литерал, затем вычисляется выражение и выводится его значение. Специальный символ `'\n'` переводит курсор на следующую строку.

Вычисления

В школе на уроках математики и физики мы научились вычислять по формулам площадь круга, периметр многоугольника, длину окружности, объем пирамиды, пройденный путь, среднюю скорость и т. п. В формулах нам встречались дробные числа: число π (пи), ускорение свободного падения g и т. п. В языках программирования помимо целых есть и дробные числа, которые называются вещественными. Вещественные числа — это один из встроенных типов C++. Эти числа называют числами «с плавающей точкой».

Вещественные числа

Напишем простую программу, которая запрашивает у пользователя стороны прямоугольника и вычисляет его периметр и площадь (листинг 3.4).

Листинг 3.4. Прямоугольник (периметр и площадь)

```
#include <iostream>
using namespace std;
int main(){
    cout << "Задайте стороны прямоугольника\n";     // приглашение
    // ввод данных
    double a;    cout << "A = "; cin >> a;
    double b;    cout << "B = "; cin >> b;
```

```
    // вычисления
    double S = a * b;           // площадь
    double P = 2 * (a + b);     // периметр
    // вывод результатов
    cout << "Площадь  прямоугольника = " << S << endl;
    cout << "Периметр прямоугольника = " << P << endl;
   return 0;
}
```

В программе объявлены две переменных `a` и `b` вещественного типа `double`, куда заносятся вводимые числа — стороны прямоугольника. Две другие переменные вещественного типа `S` и `P` инициализируются при объявлении. Как и при инициализации строковой переменной `greeting` (см. листинг 3.2), справа от операции присваивания записано выражение. Такие выражения называются *арифметическими*, т. к. это выражения с числами. Выражение вычисляется, и результат заносится в переменную. Переменная `S` получает значение, равное площади прямоугольника, а в переменную `P` заносится значение, равное периметру этого прямоугольника.

Запустим программу и зададим значения сторон `1.1` и `1.1` (рис. 3.3).

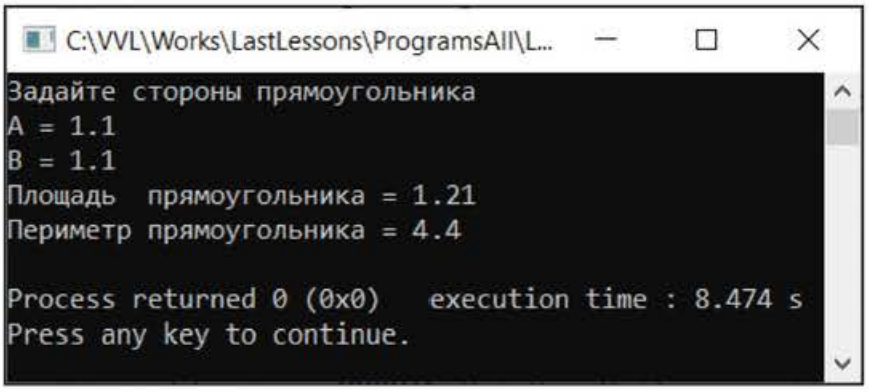


Рис. 3.3. Вычисление площади и периметра прямоугольника

Программа вычисляет результаты правильно, если размеры сторон заданы корректно. Ввод целых чисел в качестве размеров тоже является корректным. Например, если мы зададим размеры сторон как `2` и `3`, то получим в ответе:

```
Площадь  прямоугольника = 6
Периметр прямоугольника = 10
```

Выясним, какой вид вещественных чисел является корректным.

Как писать вещественные константы

Нужно сразу сказать, что вид (формат) вещественных чисел, которые задает пользователь при вводе, и вид вещественных констант, которые нам еще предстоит писать в коде, ничем не отличаются.

Вещественные константы (литералы) в коде (и при вводе) записываются в двух видах (форматах): обычном и научном. Обычный вид — он и есть обычный, только

вместо запятой, отделяющей дробную часть от целой, используется точка: `2.0`, `-123.4567`, `0.0` — и т. п. Точка в константе является признаком, что число вещественное. На рис. 3.3 размеры заданы именно в таком виде.

Научный вид (формат) позволяет нам записывать или очень большие, или очень маленькие числа, например: `1.234e45`, `3.24e+12`, `0.678e-23`. В этих константах само число называется *мантиссой*, а буква e (можно писать и большую букву E) называется *экспонентой* и играет роль «10 в степени». Степень часто называют «порядок». Буквы e или E, естественно, должны быть английские. Таким образом, первое число имеет значение 1.234×10^{45}, второе — 3.24×10^{12}, а третье — 0.623×10^{-23}. Понятно, что обычная форма записи не подойдет, т. к. придется писать очень много нулей.

В отличие от обычного вида, в научном виде мантиссу числа можно писать без точки, например: `1e45`, `324e-12`. Очевидно, что первое число имеет значение 10^{45}, а второе — 324×10^{-12}. Хотя первое число в принципе целое, но в компьютере оно представлено как вещественное. В научном формате признаком «вещественности» числа является экспонента.

В C++ разрешается не писать ноль перед точкой или после точки. Например, `2.0` можно записать как `2.`, а `0.0` можно записать как `.0` или `0.` (если число записано без точки, то это целое число).

> ***Совет***
>
> Никогда не пишите дробные константы в таком виде (без нулей около точки).

Описанные форматы дробных констант используются практически во всех языках программирования: Java, C#, Go, Pascal и др.

Тестируем программу

Программы всегда нужно тестировать. С помощью тестирования проверяется правильность работы программы. Прежние программы мы тоже тестировали, но не упоминали этого термина. Например, тестирование программы вывода приветствия по имени и фамилии пользователя (см. листинг 3.2) показало нам, что программа правильно работает только в том случае, если пользователь правильно задаст имя и фамилию.

Существует много разных видов тестирования, в том числе и тестирование ввода/вывода. Но сейчас нас интересует, правильно ли вычисляются площадь и периметр прямоугольника. Поэтому будем задавать при вводе только правильные данные.

Процесс тестирования вычислительных программ включает несколько шагов.

1. Подобрать (записать) такой набор входных данных, для которых легко вычислить результат вручную.
2. Вычислить (и тоже записать) результаты для каждого экземпляра входных данных.
3. Запустить программу для каждого экземпляра входных данных.
4. Сравнить полученные результаты с вычисленными вручную.

Если обнаруживается, что результаты программы для какого-то экземпляра входных данных не совпадают с вычисленными, то возможны два варианта:

- программа считает неправильно;
- программист неправильно посчитал результаты.

Надо выбирать данные, с которыми легко вычислить результат вручную, чтобы не ошибаться. Пример «хорошего» набора данных для программы вычисления площади и периметра приведен в табл. 3.1.

Таблица 3.1. *Набор данных для тестирования*

A	B	Площадь	Периметр
1	1	1	4
1	2	2	6
2	1	2	6
2	2	4	8
1.1	1.1	1.21	4.4
0.1	1	0.1	2.2
1.1	0.9	0.99	4

Одна строка таблицы — это один тест. A и B — входные данные, справа — результаты, которые должна выводить программа.

Конечно, для больших программ набор данных должен быть гораздо больше. Можно даже написать программу, которая создаст подобный набор данных для тестируемой программы. Но так как наша программа небольшая, то тестировать мы ее будем вручную.

Для каждого теста (одной строки в таблице) запускаем программу, вводим A и B, получаем результат в консольном окне и сравниваем с табличными результатами. Сделайте это прямо сейчас и убедитесь, что программа вычисляет площадь и периметр абсолютно правильно для всех A и B.

После тестирования можно с большой долей уверенности утверждать, что программа работает правильно. Но не все так просто, как может показаться.

Проведем эксперименты с входными данными: будем задавать корректные числа A и B, но с большим количеством знаков после запятой (после точки). Сначала будем задавать числа в обычном формате, потом в научном.

Пример 1. Зададим больше знаков в дробной части числа, например:

```
A = 1.0009
B = 1.0009
```

В результате получаем:

```
Площадь  прямоугольника = 1.0018
Периметр прямоугольника = 4.0036
```

Периметр `P` подсчитан правильно, а площадь `S` — нет. Точное значение площади для таких данных равно `1.00180081`.

Пример 2. Зададим еще чуть больше знаков после точки:

```
A = 1.00009
B = 1.00009
```

В результате получаем:

```
Площадь  прямоугольника = 1.00018
Периметр прямоугольника = 4.00036
```

Количество выведенных цифр увеличилось, но, как и в предыдущем примере, периметр посчитан верно, а площадь — снова неправильно. Точное значение должно быть таким: `1.0001800081`.

Пример 3. Добавим еще по одной цифре в дробную часть размеров сторон:

```
A = 1.000099
B = 1.000099
```

Получаем неправильный результат не только площади, но и периметра:

```
Площадь  прямоугольника = 1.0002
Периметр прямоугольника = 4.0004
```

Точные результаты должны быть такими:

```
Площадь  прямоугольника = 1.000198009801
Периметр прямоугольника = 4.000396
```

Пример 4. Попробуем задать числа в научном формате:

```
A = 1.000099e+5
B = 1.000099e+5
```

Получаем неправильный результат не только площади, но и периметра:

```
Площадь  прямоугольника = 1.0002e+10
Периметр прямоугольника = 400040                // верный ответ 400039.6
```

Опять получили неверный результат и для площади, и для периметра. Причем числа выведены в разных форматах: площадь показана в научном виде, а периметр — в обычном. Проведем еще несколько экспериментов.

Пример 5. Сначала увеличим экспоненту:

```
A = 1.000099e+10
B = 1.000099e+10
```

И опять получаем неправильный результат в обоих случаях:

```
Площадь  прямоугольника = 1.0002e+20
Периметр прямоугольника = 4.0004e+10
```

Пример 6. Попробуем теперь задать числа с отрицательной степенью:

```
A = 1.000099e-10
B = 1.000099e-10
```

Получаем ответ, аналогичный предыдущему, только с отрицательной степенью:

```
Площадь  прямоугольника = 1.0002e-20
Периметр прямоугольника = 4.0004e-10
```

Выясняется, что мы чего-то не учли, когда писали код: результаты не всегда получаются те, которые должны быть. Разберемся, почему наша программа, написанная абсолютно правильно (надеюсь, все помнят формулы площади и периметра прямоугольника), выводит ошибочные значения.

Вывод данных

Если внимательно присмотреться к выведенным числам, то можно увидеть, что выводится всегда не более шести цифр мантиссы. В таких случаях обычно говорят, что вещественное число выводится *с точностью* до шести знаков. Взглянем на третий и четвертый примеры: понятно, что при выводе результат вычисления периметра округлен. В третьем примере после округления результат получился `4.00040`, но последний ноль просто не выводится. В четвертом — шесть десятых добавляют единицу к последней цифре 9 и получается результат `400040`.

Такой режим вывода установлен по умолчанию. И наша задача — найти средство, которое позволит нам выводить больше цифр в числе. Такое средство в библиотеке потокового ввода/вывода есть — манипуляторы.

Манипуляторы

Мы знакомы пока только с манипулятором `endl`, но в стандартной библиотеке их реализовано более трех десятков. Нам нужен манипулятор, с помощью которого можно увеличить количество цифр вещественного числа при выводе в консоль. В стандартной библиотеке этот манипулятор так и называется — `setprecision` (установить точность). Чтобы использовать этот манипулятор, нужно подключить библиотеку `<iomanip>`.

После этого нужно изменить в коде (см. листинг 3.4) два оператора вывода:

```
cout <<"Площадь  прямоугольника = "<< setprecision(15) << S << endl;
cout <<"Периметр прямоугольника = "<< setprecision(15) << P << endl;
```

Запустив программу, введем те же данные:

```
A = 1.000099
B = 1.000099
```

Получим правильные результаты:

```
Площадь  прямоугольника = 1.000198009801
Периметр прямоугольника = 4.000396
```

Зададим данные в научном формате:

```
A = 1.000099e+10
B = 1.000099e+10
```

И опять получаем правильные результаты в обоих случаях:

```
Площадь  прямоугольника = 1.000198009801e+20
Периметр прямоугольника = 40003960000
```

Отметим, что вещественное число, которое не имеет дробной части и степень которого не слишком велика, выводится как обычное целое число. В нашем случае это периметр.

Снова вычисляем

Продолжим изучение вычислений с дробными числами и вспомним школьные формулы для вычисления площади круга, длины окружности и объема шара. Для всех этих формул необходим радиус, а также известная константа π (пи). Обозначим радиус буквой R. Тогда длина окружности L вычисляется так:

$$L = 2\pi R, \tag{3.1}$$

площадь круга S вычисляется по формуле:

$$S = \pi R^2, \tag{3.2}$$

а объем шара V этого радиуса можно вычислить таким образом:

$$V = \frac{4}{3}\pi \cdot R^3 \tag{3.3}$$

Радиус, естественно, задает пользователь. А где же взять константу π (пи)? Удивительно, но в стандартной библиотеке математических констант нет! Только в стандарте C++20 появилась отдельная библиотека констант, которая так и называется numbers (числа).

> ***Примечание***
>
> В операционных системах МОС и «Альт» установлены компиляторы C++, которые в значительной степени поддерживают стандарт C++20. Оба компилятора позволяют подключить стандартную библиотеку `<numbers>`.

Так как мы работаем в стандарте C++17, то должны задавать константу в коде самостоятельно. Можно задать константу π явным образом:

```
const double pi = 3.141592654;
```

Однако лучше вычислить эту константу с помощью функции. Константа π вычисляется с помощью функции арккосинуса. Нужно объявить в программе поименованную константу и присвоить ей значение арккосинуса от (–1). Для этого нужно подключить математическую библиотеку `<cmath>`, в которой реализованы элементарные математические функции, часто используемые в разнообразных вычислительных программах.

Заглянув в справочник, выясняем, что *прототип* нужной функции имеет вид:

```
double acos(double);
```

Прототип — это заголовок функции без тела, завершающийся точкой с запятой. Прототип является *объявлением* функции. В нем указываются имя функции, тип возвращаемого значения и список типов параметров. Функция `acos` принимает один параметр типа `double` и возвращает результат типа `double`. Для вычисления константы π функцию нужно вызвать и присвоить результат нашей поименованной константе.

Формулы площади круга и объема шара содержат степени радиуса. В языке C++ отсутствует операция возведения в степень. Поэтому в выражениях возведение в небольшую целую степень обычно делается умножением. Например, площадь круга можно вычислять так:

```
S = pi * r * r;
```

Теперь мы готовы написать программу. Код показан в листинге 3.5.

Листинг 3.5. Математические вычисления с числом π

```
#include <iostream>
#include <cmath>
using namespace std;
int main(){
   double const pi {acos(-1.0)};                    // число π
   cout.precision(10);                              // точность для вывода
   double r {0.0};                                  // радиус
   cout << "Задайте радиус: "; cin >> r;             // ввод радиуса
   double L {2*pi*r};                  // длина окружности = 2*pi*r
   cout << "Длина окружности = " << L << endl;
   double S {pi*r*r};                  // площадь круга = pi*r^2
   cout << "Площадь круга\t = " << S << endl;
   double V {4.0/3*pi*r*r*r};          // объем шара = 4/3*pi*r^3
   cout << "Объем шара\t = " << V << endl;
   return 0;
}
```

Поименованная константа `pi` объявлена в коде, но инициализирована она не так, как мы это делали прежде. Инициализирующее выражение (вызов функции `acos`) записано в фигурных скобках. И другие переменные (`L`, `S`, `V`) проинициализированы тем же способом. Такая форма появилась в C++11, и называется она *универсальная инициализация*.

В предыдущем примере (см. листинг 3.4) для установки повышенной точности вывода вещественных чисел мы применили манипулятор `setprecision`. Но в этой программе точность устанавливается другим способом: для объекта `cout` можно вызывать методы, реализованные для типа `ostream`. В операторе:

```
cout.precision(10);
```

вызван метод `precision`, устанавливающий точность равной 10. Этот режим работает для всех следующих операторов вывода.

Для выравнивания результатов при выводе в строковых литералах написан спецсимвол `\t` (табуляция).

Запустим программу и зададим радиус `1` (опять задаем значение, с помощью которого легко вычислить результат). Результат представлен на рис. 3.4.

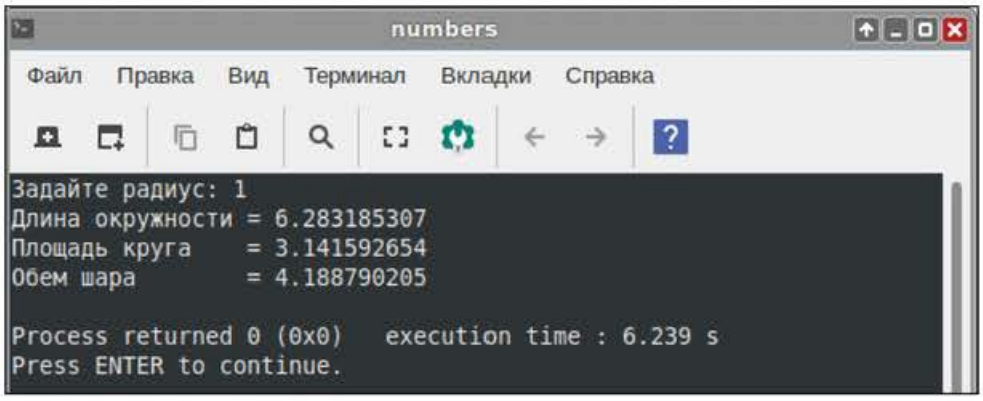


Рис. 3.4. Результат математических вычислений с числом π

Как можно видеть, это результат выполнения программы в системе «Альт».

Программа для МОС и «Альт» немного отличается от показанной в листинге 3.5. В этих системах можно установить режим трансляции по стандарту C++20 и подключить библиотеку `<numbers>`. Код программы показан в листинге 3.6.

Листинг 3.6. Математические вычисления с числом π (стандарт C++20)

```
#include <iostream>
#include <numbers>                    // pi определено здесь
using namespace std;
using namespace numbers;              // пространство имен для чисел
int main(){
   cout.precision(10);
// тот же код, что в листинге 3.5
   return 0;
}
```

В этой программе нужно обратить внимание на using-директивы. Во-первых, для использования констант необходимо задать вторую using-директиву с новым пространством имен. Если ее не написать, то придется все константы `pi` в программе задавать с префиксом `numbers::`. Во-вторых, директивы должны быть записаны именно в таком порядке. Пространство имен `numbers` входит в пространство имен `std`, поэтому если не написать первую директиву (или вовсе удалить ее), то придется во второй указать префикс `std::` перед `numbers`:

```
using namespace std::numbers;
```

В остальном эта программа не отличается от предыдущей.

Задачи и упражнения

1. Ввести с клавиатуры два числа (в интервале от 0 до 9999). Вывести их на экран столбиком в виде, удобном для сложения или умножения, т. е. первый разряд (справа налево) первого числа под первым разрядом второго числа, второй разряд — под вторым, третий — под третьим.

Например, 6 и 738:

```
6
738
```

2. Дано время в часах (h) и минутах (m). Вычислить в минутах и секундах.
3. Товар стоил X рублей, по акции скидка составляет A процентов. Сколько стоит товар по акции? (A и X — целые, цена со скидкой — вещественное). Вывести ответ с двумя знаками после запятой.
4. Написать программу, которая вычисляет площадь треугольника по высоте и основанию. Ответ вывести с точностью до четырех знаков после запятой.
5. Написать программу, которая вычисляет площадь кольца, если известны его внешний радиус R и внутренний радиус r. Ответ вывести с четырьмя знаками после запятой.
6. Составьте наборы тестовых данных и занесите их в таблицы для всех программ этого урока.

- Урок 4 -

Встроенные типы данных

Предыдущий урок научил нас объявлять переменные, инициализировать константы и вводить данные. Мы первый раз написали алгоритм будущей программы и выполнили настоящее тестирование, немного познакомились с целыми и вещественными числами и слегка погрузились в мир вычислений.

Настала пора познакомиться более детально со встроенными типами данных. Напомню, что каждый тип включает множество значений и набор операций, которые применяются к значениям. В любом языке программирования обычно есть некоторый набор встроенных типов. Операции со встроенными типами, естественно, тоже определены в языке.

В C++ встроенные типы делятся на два больших вида: арифметические и неарифметические. К арифметическим типам относятся все числа: целые и вещественные. Имеются и неарифметические типы данных — символьные и булевские (логические).

> ***На заметку***
>
> В C++ есть еще указатели, но с ними мы познакомимся позже.

Символьные, булевские и целые совместно называются целочисленными (*интегральными*) типами. Рассмотрим сначала все вещественные и целые типы, а потом и остальные встроенные типы.

Арифметические типы данных

До сих пор из арифметических типов мы имели дело только с двумя: с целым типом `int` и с вещественным типом `double`. Общее число арифметических типов в C++ тринадцать, и каждый имеет свои особенности.

Вещественные типы

Мы уже использовали тип `double` для вычислений по формулам, писали вещественные литералы, объявляли переменные и поименованные константы этого типа. Напомню, что по историческим причинам вещественные числа во всех языках назы-

ваются числами «с плавающей точкой». В C++ определено три «плавающих» типа: `float`, `double`, `long double`. Слово float так и переводится — плавать.

Вещественные константы (см. *урок 3*) тоже являются числами одного из трех указанных типов. По умолчанию константы имеют тип `double`. Чтобы задать константы других двух типов, нужно использовать *суффиксы*, которые задаются после числа (без пробела). Константа `3.14159` — типа `double`, эта же константа с суффиксом F (или f) — `3.14159F` — типа `float`, а она же с суффиксом L (или l) — `3.14159L` — типа `long double`. Лучше писать L, т. к. маленькая буква l очень похожа на 1 (единица), и при чтении кода их можно легко перепутать (даже здесь в этом тексте не видно различия). Два суффикса одновременно писать нельзя.

Вещественные типы (и числа, конечно) отличаются *диапазонами значений*, *точностью* и *размером*. *Диапазон* — это разность между самым большим и самым маленьким числом этого типа. *Точность* — это количество цифр мантиссы. *Размер* — это количество единиц памяти компьютера, которое занимают объекты этого типа. Размер, точность и диапазон связаны друг с другом: чем больше размер, тем больше точность и диапазон.

Однако стандарт C++ говорит, что эти свойства чисел зависят от реализации. В нынешних компьютерах реализован международный стандарт IEEE-754 представления чисел с плавающей точкой. Диапазоны, точность и размеры типов `float` и `double` определяются этим стандартом. А вот реализация типа `long double` не является стандартной и зависит от компилятора.

Чтобы узнать размеры вещественных чисел, надо выполнить программу, код которой приведен в листинге 4.1.

Листинг 4.1. Определение размеров вещественных чисел

```
#include <iostream>
using namespace std;
int main(){
   cout << sizeof(float) << endl;
   cout << sizeof(double) << endl;
   cout << sizeof(long double) << endl;
   return 0;
}
```

Слово `sizeof` является зарезервированным словом языка C++ и переводится как размер (чего-то). В C++ `sizeof` является операцией. Форма записи `sizeof(тип)` позволяет получить размер любого типа (и встроенного, и реализованного).

Запустив программу, мы увидим в консольном окне три числа в столбик: 4, 8, 16. На персональном компьютере, на котором я запускал программу, память состоит из *байтов*. Таким образом, эти три числа показывают, что вещественные числа типа `float` занимают в памяти 4 байта, `double` — 8 байт и `long double` — 16 байт.

Размеры констант в памяти соответствуют их типам. Чтобы убедиться в этом, замените в листинге 4.1 типы на константы (или допишите еще три оператора вывода констант с суффиксами). В консоли появятся в столбик те же числа 4, 8, 16 — это показывает, что константы имеют те же размеры.

Диапазоны и точность можно выяснить, используя библиотеку `<limits>` из STL (листинг 4.2).

Листинг 4.2. Диапазоны и точность вещественных типов

```
#include <iostream>
#include <limits>
using namespace std;
int main(){
    cout << "float numbers" << endl;
    cout << numeric_limits<float>::min() << endl;
    cout << numeric_limits<float>::max() << endl;
    cout << numeric_limits<float>::digits10 << endl;
    cout << "double numbers" << endl;
    cout << numeric_limits<double>::min() << endl;
    cout << numeric_limits<double>::max() << endl;
    cout << numeric_limits<double>::digits10 << endl;
    cout << "long double numbers" << endl;
    cout << numeric_limits<long double>::min() << endl;
    cout << numeric_limits<long double>::max() << endl;
    cout << numeric_limits<long double>::digits10 << endl;
    return 0;
}
```

В программе использовано современное средство для получения важных характеристик вещественных чисел: определенный в библиотеке `<limits>` *класс-шаблон* `numeric_limits`. Название класса является «говорящим» и переводится как «числовые пределы».

В первом операторе вывода выводится значение вычисляемого выражения `numeric_limits<float>::min()`, которое является вызовом функции `min` с префиксом. Функция `min` выдает минимально возможное значение для типа, заданного в префиксе в угловых скобках, — это тип `float`. Указание типа в угловых скобках является признаком класса-шаблона.

Следующий оператор выводит значение `max` — максимально возможное значение для типа `float`. В третьем операторе выводится точность мантиссы числа типа `float` в десятичных цифрах — `digits10`. После этого выводятся три тех же величины для типа `double` и затем — для типа `long double`.

Запустив программу, получим в консоли (рис. 4.1) все необходимые сведения о диапазонах и точности вещественных чисел (на моем компьютере).

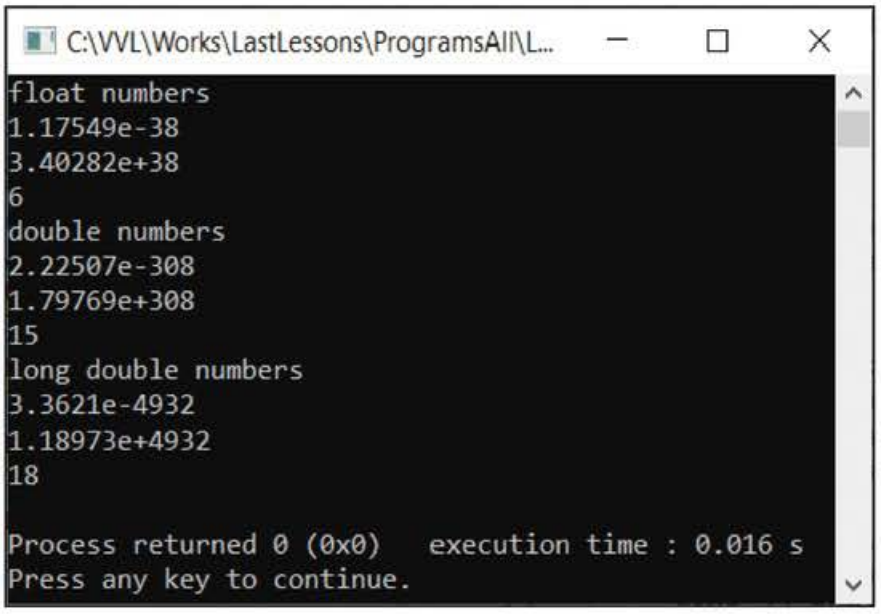


Рис. 4.1. Характеристики «плавающих» типов

Вывод показывает, что положительные (со знаком плюс) вещественные числа типа `float` имеют диапазон от 1.17549×10^{-38} до $3.40282\times10^{+38}$. Точность таких чисел — шесть десятичных цифр. Соответственно, числа типа `double` изменяются от 2.22507×10^{-308} до $1.79769\times10^{+308}$. Точность чисел `double` — 15 десятичных цифр. Диапазон чисел `long double` — от 3.3621×10^{-4932} до $1.18973\times10^{+4932}$. Точность длинных вещественных чисел равна 18 десятичных цифр.

Диапазоны отрицательных чисел те же, только со знаком минус.

> ***На заметку***
>
> Пределы вещественных чисел всех типов определены еще в унаследованной библиотеке `<cfloat>`.

С первого взгляда трудно оценить, насколько велики или малы эти числа. Для сравнения можно привести расстояние от Земли до Солнца. В Интернете находим, что среднее расстояние между нашей планетой и нашей звездой равно 149 597 870 700 метров. В научном формате это число записывается так: 1.495978707×10^{11} (метров). Во-первых, точности чисел типа `float` не хватает для представления этого числа, и оно может быть представлено числом типа `double`. Во-вторых, даже самые «маленькие» числа с плавающей точкой типа `float` намного превосходят это число.

Целые типы

Стандарт C++17 определяет пять типов целых чисел со знаком. Слова «со знаком» означают, что числа этих типов могут быть как положительными, так и отрицательными. Стандартные названия целых знаковых типов, определенные в C++, следующие:

```
signed char
signed short int
```

```
signed int
signed long int
signed long long int
```

Слово `signed` так и переводится — знаковый. Беззнаковые типы отличаются только первым словом — пишется `unsigned` (беззнаковый). «Беззнаковость» означает, что числа этих типов могут быть только положительные.

Целые числа отличаются размерами и диапазонами. Я не буду приводить код программы, которая показывает размеры целых чисел, — напишите ее по аналогии с программой в листинге 4.1 самостоятельно. При записи типа в коде можно слово `signed` не писать, т. к. целые типы считаются знаковыми по умолчанию. Разрешается не писать и слово `int`, т. е. вместо `short int` можно написать только `short`, вместо `long int` — `long`, вместо `long long int` — можно писать только `long long`.

Запустив программу, получим очень интересные результаты: знаковые и беззнаковые типы имеют одинаковый размер. В порядке, показанном выше, эти размеры составляют 1, 2, 4, 4, 8 байт.

> ***На заметку***
>
> Пределы целых чисел всех типов определены в унаследованной библиотеке `<climits>`.

Так же, как и для вещественных чисел, размеры целых не регламентированы в стандарте C++, а определяются характеристиками компьютера, на котором выполняется программа. В данном случае эти числа показывают свойства памяти моего компьютера. На компьютерах с другой организацией памяти (не байтовой) числа могут быть другими.

Диапазоны можно выяснить, написав программу, аналогичную показанной в листинге 4.2 (напишем ее чуть позже). Диапазоны указаны в табл. 4.1.

Таблица 4.1. *Диапазоны целых типов*

Тип целого	Минимум	Максимум
signed char	-128	+127
signed short	-32 768	+32 767
signed int	-2 147 483 648	+2 147 483 647
signed long	-2 147 483 648	+2 147 483 647
signed long long	-9 223 372 036 854 775 808	+9 223 372 036 854 775 807
unsigned char	0	255
unsigned short	0	65 535
unsigned int	0	4 294 967 295
unsigned long	0	4 294 967 295
unsigned long long	0	18 446 744 073 709 551 615

Изучив таблицу, можно сделать несколько интересных выводов. Во-первых, типы с одинаковыми размерами имеют одинаковые диапазоны. Во-вторых, можно сообразить, что верхняя граница беззнаковых целых равна сумме абсолютных значений знаковых диапазонов, например:

```
128 + 127 = 255
32768 + 32767 = 65535
```

И т. д. В-третьих, мы видим несимметричность знаковых диапазонов: нижняя граница — четная, а верхняя — нечетная. Это не просто так, а тоже связано с устройством компьютера и с двоичным кодированием целых чисел.

В языке C++ можно писать десятичные, шестнадцатеричные, восьмеричные и двоичные константы (литералы), но пока мы будем использовать только десятичные числа.

> ***На заметку***
>
> Первая цифра в десятичных числах не должна быть нулем. С нуля начинаются шестнадцатеричные, восьмеричные и двоичные целые числа, которые мы рассмотрим далее (см. *урок 16*).

Целые десятичные литералы (константы) всегда являются `signed`, если заданы без суффикса — это константы со знаком, даже если знак не указан. Целые литералы типа `char` и `short` в C++ отсутствуют, все целые литералы — типа `int`, `long` или `long long`. Тип литерала без суффикса зависит от величины числа. Например, если число не помещается в диапазон `int`, но укладывается в диапазон `long long`, то и тип ему присваивается `long long`.

Явное указание типа можно осуществить с помощью суффиксов.

Суффикс L (или l) позволяет задать константе тип `long`, а суффикс LL (или ll) — тип `long long`. Более «узкий» тип задать нельзя, только более «широкий». Таким образом, константа `2` имеет тип `int`, константа `2L` — тип `long`, константа `2LL` — `long long`.

Суффикс U (или u) позволяет задать целые беззнаковые литералы. Например, константа `2U` имеет тип `unsigned int`, константа `2uL` — тип `unsigned long`, а константа `2LLU` — `unsigned long long`.

В стандарте C++14 появилось важное свойство, позволяющее значительно повысить «читабельность» числовых констант (как целых, так и дробных) и уменьшить количество ошибок при их написании. Цифры в литерале можно группировать, разделяя их символом одинарной кавычки (апостроф). Группы пользователь определяет сам. Например, номер мобильного телефона можно теперь записывать в коде так:

```
+7'907'365'4227
```

Стандарт разрешает так писать и вещественные литералы.

Операции, выражения и присваивания

С данными арифметических типов в языке C++ можно выполнять очень много разнообразных операций, особенно с целыми.

Арифметические операции

С объектами вещественных и целых типов разрешается выполнять обычные арифметические операции: сложение, вычитание, умножение, деление. Если `x` и `y` — константы или переменные арифметических типов (вещественные или целые), то эти операции записываются так:

```
x + y    // сложение
x - y    // вычитание
x * y    // умножение
x / y    // деление
```

Приоритеты операций традиционные: если не задано скобок, то сначала выполняются умножение и деление, а потом — сложение и вычитание. Все операции являются *левоассоциативными*, т. е. выполняются слева направо при отсутствии скобок. Арифметические операции — *бинарные*, т. к. в них участвуют по два объекта (операнда).

Операции выполняются последовательно слева направо, но операнды могут вычисляться в любом порядке. Например, в программе есть выражение:

```
(5+2)*(7-4)
```

В этом выражении порядок вычисления операндов операции умножения `(5+2)` и `(7-4)` может быть какой угодно. Компилятор может решить, что надо сначала вычислить `(7-4)`, а потом `(5+2)`. Или наоборот. Порядок операндов в стандарте не определен, и все решает разработчик компилятора. Но результат операции умножения, естественно, гарантирован: `7 * 3 = 21`.

Операция деления выполняется по-разному для вещественных и для целых чисел. Если вещественное число `5.0` поделить на `2` (целое или вещественное), то результатом будет вещественное число `2.5`:

```
5.0 / 2 = 2.5
```

Если же целое `5` поделить на целое `2`, то результатом будет целое число `2`:

```
5 / 2 = 2
```

Говорят, что целые числа делятся нацело, дробная часть отбрасывается. Если вы внимательно посмотрите код в листинге 3.5, то заметите, что в выражении вычисления объема шара выражение деления имеет вид `4.0/3`:

```
double V {4.0/3*pi*r*r*r};              // объем шара = 4/3*pi*r^3
```

Делимое задано вещественной константой для того, чтобы результат деления был вещественным числом. Если написать `4/3`, то результат будет равен `1` — дробная часть будет отброшена, т. к. число `4` и число `3` являются целыми, и с ними «работает» деление нацело.

Для целых в языке существует еще одна (пятая) операция: *получение остатка от деления*, которая обозначается символом `%` (процент). Эта операция еще называется *делением по модулю*. Особенности работы этой операции можно увидеть, написав программу, код которой представлен в листинге 4.3.

Листинг 4.3. Деление по модулю

```
#include <iostream>
using namespace std;
int main()
{  cout << 5 % 2 << endl;
   cout << -5 % 2 << endl;
   cout << 5 % -2 << endl;
   cout << -5 % -2 << endl;
   return 0;
}
```

Результаты получаются несколько неожиданные: `1`, `-1`, `1`, `-1`, т. е. знак остатка зависит только от знака делимого и не зависит от знака делителя. Такие результаты тоже определяются свойствами компьютера.

Эту операцию часто используют для определения четности-нечетности числа. Если в целой переменной `k` хранится целое положительное число, то выражение `k % 2` равно нулю для четного числа и равно `1` — для нечетного.

Заметим один замечательный факт: в языке Pascal, который создал Никлаус Вирт для обучения начинающих программистов, операции целого деления и деления по модулю имеют собственные названия — `div` и `mod`. Этим Вирт явно указал для начинающих программистов, что эти операции отличаются от обычного деления, которое обозначается традиционно символом «/».

Присваивание

Результаты вычисления арифметических выражений надо куда-то положить (присвоить). Присваивание — это, как и арифметические операции, тоже встроенная бинарная операция. Присваивание выглядит так:

```
имя = выражение
```

Имя — это имя переменной (переменная должна быть объявлена ранее), а знак равенства (символ «=») является обозначением операции присваивания. Выражение справа от знака равенства вычисляется, и значение заносится в переменную. При этом, как во время вычисления выражения, тоже может выполниться *неявное преобразование типа* (см. ниже).

Точка с запятой в конце не поставлена, т. к. эта конструкция является выражением, а не оператором. Можно превратить это выражение в оператор, поставив в конце точку с запятой.

Поскольку операция присваивания — операция, то ее можно использовать в выражениях много раз. Такое *многократное присваивание* выглядит так:

```
имя_1 = имя_2 = имя_3 = … = имя_n = выражение
```

Эта запись тоже является выражением. Присваивания выполняются справа налево (операция *правоассоциативна*): выражение вычисляется, и значение заносится в переменную `имя_n`, потом в предыдущую переменную и так далее до переменной `имя_1`.

Операцию присваивания можно смешивать с другими операциями в выражениях, например:

```
d = 5 +(b = 6);
```

В этом операторе переменная `b` получает значение `6`, а переменная `d` будет равна `11 (5+6)`.

В программах очень часто приходится выполнять такое присваивание:

```
переменная = переменная + число;
```

В этой записи и слева, и справа от символа «=» (равно) указана одна и та же переменная. Например, в программе может быть такая запись:

```
i = i + 2;
```

Выполняется этот оператор так: к значению переменной `i` (например, `3`) прибавляется константа `2` и полученное значение `5` заносится в ту же самую переменную. Для таких выражений создатели языка C решили включить в язык специальные арифметические операции *с присваиванием*. Обозначаются такие операции двойным символом «#=», где решетка обозначает одну из арифметических операций: `+`, `-`, `*`, `/`, `%`.

Общая форма записи этих операций:

```
переменная #= выражение
```

Такая форма означает:

```
переменная = переменная # (выражение)
```

Сначала вычисляется выражение, потом выражение и переменная участвуют в операции `#`, а затем результат заносится в переменную.

Таким образом, увеличение переменной на `2` можно записать короче:

```
i += 2;
```

Как и обычное присваивание, такие операции тоже являются многократными и правоассоциативными — выполняются справа налево. В любых операциях присваивания левый операнд (куда присваивают) называется l-value, а правый (что присваивают) — r-value.

Инкремент и декремент

Перед любым объектом арифметического типа можно поставить знак «+» (плюс) или знак «–» (минус). Такой знак тоже является операцией, и она в этом случае называется *унарной* операцией (один операнд). В C++ есть много унарных операций, которые были определены еще в C. Например, почти в любой программе требуется увеличить или уменьшить значение переменной на `1` (единица). Можно писать так:

```
x += 1; i -= 1;
```

Однако можно писать короче, используя унарные операции *инкремента* (двойной символ плюс, «++») и *декремента* (двойной символ минус, «--»). Обе операции имеют префиксный и постфиксный вид:

```
x++     // инкремент – постфиксный вид операции
x--     // декремент – постфиксный вид операции
```

```
++x     // инкремент – префиксный вид операции
--x     // декремент – префиксный вид операции
```

Переменная x может иметь любой целый или вещественный тип. Все это, как обычно, выражения. Поставив после выражения точку с запятой, получаем оператор изменения переменной x на единицу, например:

```
++x;    // x = x + 1
x--;    // x = x - 1
```

Возникает вопрос: почему операции включены в язык в двух формах, если действие их одно и то же? Ответ очевиден: операции работают по-разному. Проявляется это в выражениях с некоторыми другими операциями (одна из них присваивание). Пусть переменная x имеет значение 5. Рассмотрим два оператора присваивания:

```
d = ++x;
```

и

```
d = x++;
```

В первом случае переменная x увеличится на 1, и новое значение 6 запишется в переменную d. Второй оператор работает по-другому: сначала число 5 присваивается переменной d, а потом значение x увеличивается на 1. Нужно отметить, что скобки в данном случае не помогают — оператор:

```
d = (x++);
```

работает точно так же — сначала число 5 попадает в d, потом происходит увеличение x на 1. Разница проявляется и в другом: операции префиксного инкремента/декремента являются многократными. Например, мы можем для увеличения переменной x на 2 написать оператор:

```
++++x;
```

И компилятор правильно «разруливает» эту ситуацию, дважды увеличивая переменную x на единицу. А вот пост-операции таким образом использовать нельзя: компилятор «ругается».

Операции декремента работают аналогично.

Преобразование типов

В арифметических операциях нередко участвуют объекты разных типов, поэтому перед операцией осуществляется *неявное преобразование* типов.

> ***На заметку***
>
> Неявные преобразования осуществляются во многих конструкциях C++. Это иногда приводит к неожиданным ошибкам при выполнении программы.

В арифметических выражениях работает всего два принципа неявного преобразования (приведения) типов по умолчанию:

1. Тип меньшего диапазона приводится к типу большего диапазона.
2. Целый тип приводится к вещественному типу.

Те же самые принципы работают и для операции присваивания. Коротко правила преобразования по умолчанию можно описать такой цепочкой:

```
[[char,short]->int]->unsigned int->long->unsigned long->float
float->double->long double
```

В скобках записано обязательное преобразование типов `char` и `short` перед выполнением операции — эти типы всегда преобразуются в `int`, а затем выполняется операция. Естественно, если потребуется, то объект типа `int` без промежуточных преобразований может быть сразу преобразован в объект типа, например, `double`.

По умолчанию может выполняться и обратное преобразование, например:

```
float -> int
```

Такое преобразование чаще всего происходит при выполнении операции присваивания. Приведения такого рода всегда связаны с потерей информации (дробная часть отбрасывается), поэтому оно называется «сужающим».

Конечно, в C++ имеются средства явного преобразования типов. Явные преобразования программист пишет в коде там, где это нужно по задаче.

Конструкции для явных преобразований унаследованы из C, и в C++ были добавлены новые. Явное преобразование типа в стиле C может записываться в двух формах:

```
тип(выражение)
(тип)выражение
```

С точки зрения результата эти формы абсолютно эквивалентны. Страуструп называет первый вариант приведением «в функциональном стиле». Я обычно пользуюсь первым вариантом и практически не применяю второй. Просто такой вид приведения гораздо лучше читается. Кроме того, по современному стандарту первый вариант можно написать с фигурными скобками:

```
тип{выражение}
```

А второй вариант с фигурными скобками писать нельзя.

В стандарт C++ включены несколько не взаимозаменяемых конструкций для явного преобразования типов. На данном этапе нас интересует только одна из них (всего таких конструкций четыре):

```
static_cast<тип>(выражение)
```

Эта операция осуществляет преобразование типов с проверкой на этапе компиляции. Например, все преобразования в приведенной выше цепочке можно выполнять явно с помощью конструкции `static_cast`.

Теперь вернемся к программе, которая выдает нам диапазоны целых чисел (листинг 4.4). В ней необходимо задать явное приведение к целому типу для «маленьких» чисел типа `char` и `unsigned char`. Покажем только эти строки, остальной код аналогичен коду в листинге 4.2, только типы надо проставить знаковые и беззнаковые целые.

Листинг 4.4. Приведение типов *char* и *unsigned char*

```
#include <iostream>
#include <limits>
```

```
using namespace std;
int main(){
    cout << int(numeric_limits<char>::min()) << endl;
    cout << int(numeric_limits<char>::max()) << endl;
    cout << int(numeric_limits<unsigned char>::min()) << endl;
    cout << int(numeric_limits<unsigned char>::max()) << endl;
    return 0;
}
```

Я написал преобразование в стиле C, поскольку так меньше писать. В стиле C++ оператор вывода был бы очень длинный:

```
cout << static_cast<int>(numeric_limits<char>::min()) << endl;
```

Без явного приведения система ввода-вывода видит тип `char` и выводит не целое, а символ. В консоли появляются либо какие-то символы, либо вообще ничего не появляется (просто пустая строка).

Объявление переменных и констант

Как мы уже знаем, переменные любых типов объявляются с помощью операторов объявления. Оператор объявления начинается с типа, который может быть встроенным или реализованным. В одном операторе объявления может быть определено несколько переменных через запятую.

Вещественные переменные объявляются так:

```
float x, y, z;
double a;
long double v, w;
```

При объявлении переменные можно явно инициализировать, т. е. задавать (присваивать) начальные значения. Это можно делать разными способами:

```
float t = 0.456, p = 1e-23;
double d (2.71828);
long double dd {1.1}, ld{2.2};
```

Первый вариант (с присваиванием) появился еще в C. Второй способ (с круглыми скобками) появился в C++. Этот способ очень часто применяется для инициализации переменных и констант реализованных типов. Напомню: константа `asterisk` (см. листинг 3.2) проинициализирована таким способом.

Однако этот способ в некоторых конструкциях провоцировал ошибки, поэтому в C++11 появился третий вариант — с фигурными скобками. Этот способ инициализации называется «универсальная инициализация», и во всех учебниках по C++ настоятельно рекомендуют инициализировать переменные (и константы) именно в таком виде.

Универсальная инициализация позволяет обнулить переменную, не задавая нуля в фигурных скобках:

```
int k{};
```

Допускается смешивать в одном операторе объявления разные способы инициализации, например:

```
float a{1.1}, b(1.2), c = 0.8;
```

Однако лучше так не делать и всегда инициализировать переменные одним и тем же способом. Я по привычке инициализирую переменные с помощью присваивания, но рекомендую использовать универсальную инициализацию как более современный и безопасный способ. Безопасность реализована реально: нельзя инициализировать целую переменную вещественной константой — возникает ошибка компиляции. Например, оператор объявления:

```
int y {7.};
```

вызовет ошибку компиляции:

```
error: narrowing conversion of '7.0e+0' from 'double' to 'int'
```

Слова narrowing conversion переводятся как «сужающее преобразование». И это именно то преобразование из вещественного в целое, о котором мы говорили выше. Сообщение об ошибке компилятор выводит только для универсальной инициализации. Операторы объявления с инициализацией по первому или второму варианту не вызывают даже предупреждения.

Для вещественных переменных допускается в качестве начальных значений писать и целые константы:

```
long double x = 1, z {0};
```

В этом случае выполняется приведение типов: целая константа преобразуется в вещественную указанного типа и присваивается переменной.

Инициализировать переменную можно другой переменной, например:

```
float a{1.1}, b(a), c = b;
double d{a};
```

Инициализация выполняется в порядке объявления переменных, и уже объявленную проинициализированную переменную можно использовать для инициализации следующей. Как мы уже видели (см. листинг 3.6), можно инициализировать переменные с помощью выражения, если все переменные в выражении объявлены ранее и имеют значение.

Вместо переменной можно объявить поименованную константу, как мы это делали в программе приветствия пользователя в рамочке (см. листинг 3.2):

```
const double dd = 0.123456;
float const exponenta {2.718281828};
```

Разрешается писать слово `const` как перед типом, так и после него — оба варианта имеют одинаковый смысл.

В стандарте C++11 появилось новое зарезервированное слово — `constexpr`, с помощью которого также можно задавать поименованные константы:

```
constexpr double dd = 0.123456;
float constexpr exponenta {2.718281828};
```

Конечно, объявление поименованных констант с помощью слова `constexpr` отличается от объявления с `const`, но эта разница для нас несущественна.

Целые переменные и поименованные константы объявляются аналогично:

```
unsigned long long int constexpr Length {1'000'000};
```

Такие длинные объявления программисты писать очень не любят, поэтому изыскивают средства укоротить написание типа. Такие средства в C++ есть и широко используются.

Объявления *auto*

Можно объявление переменной или поименованной константы написать с помощью зарезервированного слова `auto`, например:

```
auto variable {8.7};                    // double
auto var = 1.2F;                        // float
auto const ldvar {2.2L};                // long double
constexpr auto ivar {8};                // int
auto Length {18'000'000'000uLL};        // unsigned long long
```

В таких объявлениях инициализация обязательна, поскольку тип переменной заранее неизвестен и все зависит от типа инициализирующего значения (или выражения): компилятор разбирается, какого типа константу (выражение) мы написали, и переменная получает этот тип. Говорят, что тип переменной *выводится* из типа инициализирующего выражения.

Такое объявление требует внимательности. Если вы по какой-то причине не поставите в константе точку, то тип константы будет целый и переменная будет объявлена как целая. Или, наоборот, поставите точку там, где не нужно, и переменная будет объявлена как `double`.

Мы увидим в дальнейшем, что объявление `auto` позволяет сократить код для некоторых типов, реализованных в стандартной библиотеке.

Синонимы стандартных целых типов

Так как полные имена целых типов слишком длинные, в библиотеке C++ для этих типов определены короткие синонимы. Нужно подключить библиотеку:

```
#include <cctype>
```

Как видим, библиотека унаследована от C (программисты не любили писать длинные типы уже в те далекие времена). Но язык C тоже развивается и имеет свои стандарты, поэтому в C++11 унаследована более современная библиотека C `<cstdint>`, и лучше подключать ее.

> ***На заметку***
>
> В библиотеке `<cstdint>` определены пределы для каждого синонима типа.

После этого можно в программе писать короткие имена целых типов, например:

```
int8_t  ai;             // signed char
int16_t bi;             // signed short
```

```
int32_t ci;          // signed int
int64_t di;          // signed long long
uint8_t  au;         // unsigned char
uint16_t bu;         // unsigned short
uint32_t cu;         // unsigned int
uint64_t du;         // unsigned long long
```

По традиции, принятой еще в C, синонимы типов имеют суффикс `_t`. Размеры этих типов соответствуют размерам, написанным справа в комментариях. Вы можете в этом убедиться, написав программу с выводом размеров, как это показано в листинге 4.1.

В названии типов мы видим числа `8`, `16`, `32` и `64`. Эти числа равны количеству *битов*, которые занимает в памяти этот тип. Таким образом, можно сделать вывод, что 1 байт содержит 8 бит.

> ***На заметку***
>
> Без знания битовой структуры памяти и данных невозможно разобраться с некоторыми проблемами, регулярно возникающими в коде.

Однако размеры целых типов зависят от реализации (в том числе и типы `char`). В стандарте C++ прописано единственное требование, чтобы типы символов занимали 1 единицу памяти:

```
sizeof(char) = sizeof(signed char) = sizeof(unsigned char) = 1
```

В C++20 добавлен еще один встроенный символьный тип — `char8_t`, размер которого тоже должен быть равен 1 единице памяти.

Все целые типы по размеру должны быть *не меньше* символьного.

Отмечу еще одну особенность приведенных синонимов целых типов. Так как 8-битные типы `int8_t` и `uint8_t` являются синонимами символьных типов `signed char` и `unsigned char`, то при выводе переменных в консоль надо задавать явное преобразование в целый тип:

```
cout << int(ai) << endl;
```

иначе вместо целого будет выводится символ.

Объявление синонимов

В языке C++ есть средства, позволяющие программисту самому написать синоним длинного типа. Синоним по традиции называют *алиасом*. Средств, обеспечивающих объявление алиаса, в языке два: старый оператор `typedef` и using-объявление, добавленное в C++11.

Оператор `typedef` для встроенных типов имеет формат:

```
typedef имя_типа синоним;
```

Например, мы можем объявить синоним для `double`:

```
typedef double real;
```

После этого можно объявлять переменные как типа `double`, так и типа `real`. В этом операторе можно на месте синонима указать несколько синонимов через запятую, например:

```
typedef char byte, small;
```

Все эти синонимы эквивалентны.

Объявление с помощью `using` имеет формат:

```
using синоним = тип;
```

Для встроенных типов using-объявление аналогично оператору `typedef`. Но имеет более широкие возможности при работе с шаблонами. Поэтому лучше всегда использовать для определения синонимов типов using-объявление.

Порешаем задачи

Программисты постоянно думают о том, как бы сократить код программы и ускорить выполнение. Решение первой задачи покажет нам, как достигается экономия кода.

Задача 1. Математика должна быть экономной

Написать программу, вычисляющую значения `y` и `z` по формулам:

$y = 1 + 2x + 3x^2 + 4x^3$

$z = 1 - 2x + 3x^2 - 4x^3$

В программе разрешается использовать суммарно не более восьми операций сложения, вычитания и умножения (других операций быть не должно).

Входные данные: x (задается пользователем).

Выходные данные: y z.

Эта задача об экономии: надо за восемь операций вычислить y и z. Однако только в одной первой формуле насчитывается три операции сложения, умножение для получения x^2, умножение для получения x^3, три операции умножения на константы 2, 3, 4. А еще во второй формуле есть два вычитания.

Очевидно, что требуется преобразование формул. Заметим, что:

$y = (1 + 3x^2) + (2x + 4x^3)$

$z = (1 + 3x^2) - (2x + 4x^3)$

Поэтому мы вычислим сначала две величины `a` и `b`:

$a = (1 + 3x^2)$

$b = (2x + 4x^3)$

Для вычисления обеих величин нам требуется значение x^2, т. к.:

$b = x(2 + 4x^2)$

И вся последовательность вычислений будет такой:

```
x2 = x × x
a = (1 + 3 × x2)
```

```
b = x ×(2 + 4 × x2)
y = a + b
z = a - b
```

В этой последовательности действий всего восемь операций, как и требовалось в условии задачи. Мы написали алгоритм решения. Код — в листинге 4.5.

Листинг 4.5. Математика должна быть экономной

```
#include <iostream>
using namespace std;
int main(){
    double x{0}; double a {0}, b{0};
    cout << "Задайте число x: "; cin >> x;
    double x2 = x * x; a = (1 + 3 * x2); b = x *(2 + 4 * x2);
    double y = a + b; double z = a - b;
    cout << y << ' ' << z << '\n';
    return 0;
}
```

Программу надо протестировать. Минимальный набор данных, для которых проще всего вычислить значения этих многочленов: `x = 0`, `x = 1` и `x = -1`. Программа в этом случае выдает совершенно правильные значения y и z:

```
1 и 1 для x = 0;
10 и -2 для x = 1;
-2 и 10 для x = -1.
```

Можете потестировать программу с другими значениями x и убедиться, что во всех случаях программа выдает правильные результаты.

Мы уже немного представляем себе, как работать с вещественными числами, а с целыми еще практически не имели дела. Напишем несколько программ, в которых разберемся, как работают операции деления целых и деления по модулю.

Задача 2. Стоимость учебников

Стоимость одного учебника составляет `N` рублей и `K` копеек. Для класса купили M учебников. Написать программу, которая вычислит полную стоимость партии из `M` учебников в рублях и копейках.

Входные данные: N, K, M.

Выходные данные: количество рублей, количество копеек.

Самый простой способ решения задачи — перевести все в копейки, умножить на количество учебников, а потом перевести все обратно в рубли и копейки.

Единственное, что требует внимания, — это типы переменных, в которых будет храниться и вычисляться полная стоимость учебников в копейках. Стоимость одного учебника небольшая. Но учебников может быть очень много — несколько мил-

лионов, и результат может не поместиться в переменную типа `int`. Если значение не помещается в переменную, то такая ситуация называется *переполнением*.

Поэтому для хранения стоимости закупки нужно объявить переменную типа `long long`. С учетом этого код выглядит так (листинг 4.6).

Листинг 4.6. Стоимость учебников

```
#include <iostream>
using namespace std;
int main(){
    cout << "Задайте стоимость учебника (N рублей, K копеек)\n";
    int N{0}, K{0};
    cout << "N рублей = "; cin >> N;
    cout << "K копеек = "; cin >> K;
    cout << "Задайте количество учебников\n";
    int M{0};
    cout << "M учебников = "; cin >> M;
    int S {N * 100 + K};
    long long Summa {S * M};
    cout << Summa/100 << ' ' << Summa%100 << '\n';
    return 0;
}
```

Первое выведенное число означает количество рублей, второе — копеек.

Задача 3. Число-палиндром

Палиндромом называется последовательность символов (предложение, слово, число), которая читается слева направо и обратно одинаково. Например, слово «потоп» является палиндромом. Если речь идет о числах, то числовой палиндром обычно называют симметричным числом.

Дано целое число `K`, содержащее не более четырех цифр. Написать программу, которая определит, является ли число симметричным. Например, число 4334 является симметричным, число 770 тоже является симметричным, т. к. ведущий ноль присутствует, но не пишется.

Если число симметричное, программа должна выдавать 1 (единица). Если число не симметрично, то программа должна выдать любое другое число.

Входные данные: число K, $0 \le K \le 9999$.

Выходные данные: 1, если число симметричное; если нет, то любое целое.

Требуется получить ответ из числа K. Надо придумать формулу, по которой для симметричного числа вычисляется единица, а если число не палиндром, то формула дает не единицу. И просто вывести это число.

Очевидно, что надо получить все цифры числа, а потом уже думать, как их скомбинировать. Числа вычисляются так:

```
d1 = K % 10;          // единицы
d2 = K / 10 % 10;     // десятки
d3 = K / 100 % 10;    // сотни
d4 = K / 1000 % 10;   // тысячи
```

Теперь можно вычислить разности `d4 – d1` и `d3 – d2`. Эти разности должны быть равны нулю, если цифры одинаковые. Если их сложить и прибавить 1, то результат должен быть равен 1. Проверьте это для числа 2772.

Таким образом, пишем формулу:

```
S = (d4 - d1) + (d3 - d2) + 1
```

Однако мы не учли один важный случай: разности могут компенсировать друг друга, если одна разность положительная, а другая отрицательная. Пусть K равно 2020. Тогда `d4-d1=+2`, а `d3-d2=-2`. В этом случае наша формула тоже выдает единицу, однако число не является симметричным. Поэтому нужно складывать не сами разности, а их абсолютные значения (листинг 4.7).

Листинг 4.7. Симметричное число

```
#include <iostream>
#include <cmath>
using namespace std;
int main(){
    int K{0};
    cout << "Задайте число K (0 <= K <= 9999): "; cin >> K;
    int d1 = K % 10;                    // единицы
    int d2 = K / 10 % 10;               // десятки
    int d3 = K / 100 % 10;              // сотни
    int d4 = K / 1000 % 10;             // тысячи
    int S {abs(d4-d1) + abs(d3-d2) + 1};        // формула симметричности
    cout << S << '\n';
    return 0;
}
```

Естественно, подключена математическая библиотека `<cmath>`, чтобы иметь возможность вызвать функцию `abs()`.

Задачи и упражнения

1. Дано пятизначное число. Симметрично ли оно?
2. Напишите набор тестовых данных для проверки программы в листинге 4.6.
3. Дано четырехзначное число. Обменять первую и последнюю цифры.

   ```
   1234 -> 4231
   1230 ->  231
   ```

4. Дано число a. Вычислить a^8 за три умножения.
5. Задано положительное число — количество секунд. Выразите это время в часах, минутах и секундах.

```
73 сек -> 0 ч 1 мин 13 сек
600 сек -> 0 ч 10 мин 0 сек
3601 сек -> 1 ч 0 мин 1 сек
```

6. Вычислить площадь фигуры, получившейся в центре квадрата, если из каждой его вершины как из центра провели дугу радиуса R (рис. 4.2).

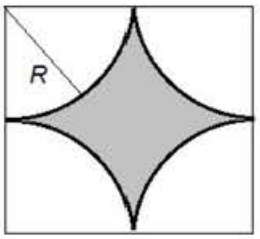


Рис. 4.2. Фигура

- Урок 5 -

Выбор есть всегда

Пока все наши программы содержали операторы, которые выполнялись один за другим в порядке следования в программе. Однако часто возникает необходимость выполнить либо одну последовательность операторов, либо другую в зависимости от некоторых условий. Например, данные, получаемые программой от пользователя, могут содержать ошибки, поэтому их нужно обязательно проверить. Если данные некорректные, то программа должна сообщить об этом пользователю и закончить работу. Или потребовать ввести правильные значения. Если же ошибок не обнаружено, то программа должна перейти к вычислениям.

Подобные действия записываются с помощью операторов управления. Оператор `return`, с которым мы познакомились при изучении программы «Hello World» (см. листинг 2.1) и который будем использовать регулярно, тоже является оператором управления. Такие операторы есть в любом языке программирования. В C++ операторы управления обязательно начинаются с зарезервированного слова.

Для того чтобы иметь возможность использовать эти конструкции в коде, нужно сначала разобраться, как задавать условия в операторах управления.

Логический тип

В язык C++ (как и во все языки программирования) включены операции сравнения. Всего в языке существует шесть операций сравнения. Объекты целых и вещественных типов, естественно, можно сравнивать друг с другом:

```
x < y    // x меньше y
x > y    // x больше y
x <= y   // x меньше или равно y
x >= y   // x больше или равно y
x == y   // x равно y
x != y   // x не равно y
```

> ***На заметку***
>
> Обратите внимание на операцию равенства «==». Новички часто ошибаются, записывая операцию сравнения как операцию присваивания `x = y`.

Операции сравнения имеют интересное свойство: операнды операции имеют один тип (числа), а результат — другого типа. В жизни мы с такой ситуацией сталкиваемся ежедневно. Когда мы хотим узнать, во сколько раз одна сумма денег больше другой суммы, мы делим одну сумму на другую. Результатом операции деления являются совсем не деньги, а вещественное число. То же самое происходит и в данном случае: результатом любой операции сравнения с объектами любых типов является *логическое* значение «истина» или «ложь».

Логическое значение не относится к арифметическим типам, и для работы с такими значениями все языки программирования включают логический тип. В языке C++ этот тип называется `bool` (как обычно, название типа является зарезервированным словом). Такое несколько странное название произошло от имени британского математика Джорджа Буля, который создал алгебру логики, названную в его честь булевой алгеброй.

Логический тип включает всего два значения, константы логического типа `true` (истина) и `false` (ложь) являются зарезервированными словами.

Логический тип относится к целочисленным (интегральным) типам.

Логические операции и выражения

Напомню, что любой тип определяет не только множество значений, но и множество операций, которые можно применять к значениям этого типа. В языке C++ определено три операции, операндами в которых могут быть логические значения:

- операция `&&` — логическое И;
- операция `||` — логическое ИЛИ;
- операция `!` — логическое НЕ.

Первые две операции — бинарные (два операнда), третья — унарная (один операнд). Первая операция часто называется логическим умножением, а вторая — логическим сложением. Третья операция — логическое отрицание.

При отсутствии скобок сначала выполняется унарная операция отрицания (как унарный минус в арифметических выражениях), затем, естественно, логическое умножение и в последнюю очередь — логическое сложение.

Для этих (и некоторых других) логических операций в C++ существуют традиционные имена (которые, например, имеются в языке Pascal):

- операция `and` — логическое И;
- операция `or` — логическое ИЛИ;
- операция `not` — логическое НЕ.

В логических выражениях можно использовать и первую, и вторую нотацию. Но старайтесь не смешивать разные нотации в одном выражении.

Логическое ИЛИ выполняется по следующим правилам:

```
false || false = false
false || true  = true
```

```
true  || false = true
true  || true  = true
```

Логическое И выполняется по следующим правилам:

```
false && false = false
false && true  = false
true  && false = false
true  && true  = true
```

Операции ИЛИ и И гарантируют порядок вычисления операндов: сначала всегда вычисляется левый операнд, а затем правый. В связи с этим операции являются сокращенными: второй операнд не вычисляется, если уже по первому ясен результат. Например, выражение:

```
(true || выражение)
```

всегда будет равно `true`, поэтому после вычисления первого аргумента нет нужды вычислять второй. Аналогично выражение:

```
(false && выражение)
```

всегда равно `false`, поэтому второй аргумент также можно не вычислять.

Логическое отрицание подчиняется следующим правилам:

```
!false = true
!true  = false
```

Эти операции широко используются в объектно-ориентированной разработке при реализации операций сравнения для нового типа данных. Сначала непосредственно реализуются всего две операции: операция равенства и операция «меньше». Остальные операции реализуются на основе этих двух с помощью логических операций. Пусть у нас реализованы операции:

```
x == y          // x равно y
x < y           // x меньше y
```

Тогда остальные можно будет реализовывать так:

```
!(x == y)                    // x не равно y
!(x < y)                     // x больше или равно y
(x == y)||(x < y)            // x меньше или равно y
!((x == y)||(x < y))         // x больше y
```

Я написал все выражения со скобками. Просто я не помню на память приоритеты логических операций и операций сравнения, поэтому с помощью скобок гарантировал нужный мне порядок выполнения операций.

> ***На заметку***
>
> В C++ вообще все операции имеют приоритет и ассоциативность.

Логические операции работают точно так же, как и в математической логике. Если `a` и `b` — логические выражения, то выполняются следующие равенства:

```
!!a == a
!(a || b) == !a && !b
!(a && b) == !a || !b
```

Первое равенство называется законом двойного отрицания. Второе и третье равенства — это известные в математической логике законы де Моргана.

Проверяем входные данные

Будем учиться работать с булевскими данными на практике. Займемся решением типовой школьной задачи: найти корень уравнения $ax + b = 0$. Вроде бы ничего сложного: надо $-b$ поделить на a и получим x.

Но давайте сначала напишем алгоритм (напоминаю, это надо делать всегда, когда начинается разработка более или менее «серьезной» программы). Первым пунктом, естественно, является ввод данных:

```
1. Ввод a и b.
```

Вторым пунктом неопытный программист сразу напишет:

```
2. x = -b / a.
```

Однако тут сразу возникают проблемы: а если пользователь задаст значение a, равное нулю? В этом случае x вычислить нельзя — решения нет. А если еще и значение b равно нулю, то x может быть любым. Поэтому в алгоритм надо включить проверку значений, которые ввел пользователь. Нужно проверить, не равно ли a нулю. Если не равно, то можно смело делить. Если же a равно нулю, то надо проверить b. Если b не равно нулю, то решения нет, в противном случае x — любое число.

Напишем все это в виде алгоритма на псевдокоде:

```
1. Ввод a и b.
2. Если a не ноль, то x = -b / a, вывести x.
3. Иначе: (тут имеем a = 0)
        если и b не ноль, то вывести «решения нет».
4. Иначе: (тут имеем a = 0 и b = 0)
        вывести «x - любое число».
5. Завершить программу.
```

Здесь уже видны некоторые сложности. Во-первых, требуется проверить значения переменных. Во-вторых, требуется выводить три разных результата в зависимости от значений входных данных. Нам помогут разобраться с этими задачами две очень необычные операции C++: *условная* операция и операция *запятая*.

Условная операция

Условная операция «?:» — единственная операция, имеющая три аргумента:

```
выражение_условие? выражение_1: выражение_2
```

Выполняется она так: вычисляется `выражение_условие` и преобразуется в значение типа `bool`. Преобразование выполняется так: если результат не равен нулю, то `true`, иначе `false`. Потом, если результат преобразования равен `true`, вычисляется `выраже-`

ние_1 и выдается в качестве результата операции; если же результат преобразования равен false, то вычисляется выражение_2.

Эта операция позволяет осуществлять условные вычисления. Например, выбор максимума из двух чисел можно записать так:

```
max = (a > b)? a: b;
```

Можно первый операнд записывать и без скобок, но со скобками понятней. Если a и b — выражения, то можно записать их в скобках, например:

```
max = ((a)>(b))?(a):(b);
```

Вообще говоря, писать иногда «лишние» скобки бывает совсем не лишним.

Заметим, что первый операнд (выражение_условие) может быть какого угодно типа, даже вещественного. В этом можно убедиться, написав в коде оператор:

```
(1.5)? cout << "!=0" : cout << "==0";
```

Сначала выполняется неявное сужающее преобразование вещественного к целому, а потом не равное нулю целое преобразуется к true и в консоль выводится сообщение "!=0".

С помощью условной операции можно выразить логические операции:

- !выражение равно (выражение?false:true);
- (выраж_1&&выраж_2) равно (выраж_1?(выраж_2?true:false):false);
- (выраж_1||выраж_2) равно (выраж_1?true:(выраж_2?true:false)).

Условная операция имеет такой же приоритет, как и операции присваивания.

Операция *запятая*

Удивительная, единственная в своем роде, бинарная операция:

```
выражение_1, выражение_2
```

Операция гарантирует порядок вычисления операндов: сначала вычисляется выражение_1 и только после него — выражение_2. Результат выполнения операции равен выражение_2.

В языке C++ на месте любого выражения можно записать два выражения через запятую. Так как приоритет этой операции самый низкий среди всех операций, она выполняется самой последней. Операция левоассоциативна. Эта операция так всегда и используется: в том месте, где надо гарантировать нужную последовательность вычислений, их можно записать через запятую.

С помощью операции запятая операцию постфиксного инкремента:

```
d = x++;
```

можно формально записать так:

```
d = (t = x, x += 1, t);
```

где переменная t должна иметь тот же тип, что и x. Напишите короткую программу и убедитесь, что эти два оператора дают один и тот же результат.

Решаем задачу

С помощью этих двух операций можно решить нашу задачу. Ввод проблем не доставляет:

```
double a {0.0}, b{0.0};
cout << "Задайте a и b: \n";
cout << "a = "; cin >> a;
cout << "b = "; cin >> b;
```

Далее объявим переменную `x` и установим точность при выводе `x`:

```
double x {0.0};
cout.precision(10);
```

Пункт 2 нашего алгоритма выглядит так:

```
(a != 0)?(x = -b/a, cout << x)...
```

Это в точности то, что написано в пункте 2, только на языке C++. Если `a` не ноль, то вычисляется `выражение_1`, которое у нас состоит из двух выражений через запятую: первое выражение вычисляет `x`, второе его выводит.

Теперь надо написать пункты 3 и 4 (если $a = 0$).

Это вторая часть условной операции, которая начинается с двоеточия, и мы напишем ее тоже в скобках:

```
:(...);
```

Так как вся условная операция представляет собой оператор-выражение, в конце необходимо поставить точку с запятой. Внутри скобок тоже должна быть написана условная операция, т. к. надо проверить условие $b = 0$:

```
(b == 0)?(cout << "x любое"):(cout << "нет решения")
```

Собирая все вместе, получим такой оператор:

```
(a != 0)                              // проверка: a != 0 ?
?(x = -b/a, cout << x)                // да: вычисляем x и выводим
:((b == 0)                            // иначе: проверка: b == 0 ?
  ?(cout << "x любое")                // да: x - любое
  :(cout << "нет решения")            // иначе: решения нет
 )                                    // конец иначе
;                                     // конец оператора
```

Мы справились с задачей, однако для более сложных проверок условная операция слабовата. Если во втором или третьем операнде надо писать очень большую программу, внутри которой тоже могут быть проверки каких-то данных, то нам придется несладко. Поэтому нужны более развитые средства управления ходом выполнения — операторы управления.

Условный оператор

Вспомним программу «Универсальная таблица умножения» (см. листинг 3.3). Если мы вновь запустим программу и введем значение `3`, то увидим на экране таблицу умножения на 3. Более того, мы можем задать любое целое положительное число,

например 147, и получим таблицу умножения на 147. Мы даже можем вычислять таблицы умножения для отрицательных чисел. Запустим программу и введем значение -2 (минус два). Программа правильно вычислит и выведет на экран отрицательную таблицу умножения. Таким образом, программа работает и достаточно универсальна.

Тем не менее изначально требовалось иметь программу, вычисляющую таблицы умножения от 1 до 9. Наша программа делает «лишнюю» работу. Гуру программирования утверждают: если программа выполняет работу, не предусмотренную первоначальным заданием, то такая программа *ошибочна.* Поэтому в программе умножения нужно проверять значения, которые вводит пользователь, и отвергать некорректные. Пусть программа в этом случае выводит на экран вежливое предупреждение и заканчивает работу.

Напишем это более формальным способом:

```
если 1 ≤ к ≤ 9, то
   вывести таблицу умножения;
иначе
   вывести сообщение об ошибке;
```

Чтобы запрограммировать эти действия, требуется *условный оператор.* Такой оператор входит во все языки программирования. В C++ условный оператор имеет две формы. Первая форма более проста и записывается так:

```
if (условие) оператор;
```

Этот вид оператора работает следующим образом: если условие истинно, то выполняется заданный оператор. Затем выполняется следующий после условного оператор. Если же условие ложно, то заданный оператор пропускается, и сразу начинает работать следующий после if оператор.

Вторая форма записывается следующим образом:

```
if (условие)
   оператор_1;
else
   оператор_2;
```

Этот вид оператора работает так: если условие истинно, то выполняется заданный оператор_1; если условие ложно, то работает заданный оператор_2. Затем выполняется оператор, следующий после условного.

Как мы видим, вторая форма оператора практически совпадает с написанным нами псевдокодом. Однако нам надо решить две проблемы. Первая проблема состоит в том, что фраза «вывести таблицу умножения» означает запись на языке C++ десяти операторов вывода, в то время как в операторе if можно указывать только *один* оператор. Эта проблема решается очень просто — с помощью фигурных скобок. Оператор if в этом случае приобретает вид:

```
if (условие)
{  cout << k << "*1=" << k*1 << endl << k << "*2=" << k*2 << endl;
   cout << k << "*3=" << k*3 << endl << k << "*4=" << k*4 << endl;
```

```
   cout << k << "*5=" << k*5 << endl << k << "*6=" << k*6 << endl;
   cout << k << "*7=" << k*7 << endl << k << "*8=" << k*8 << endl;
   cout << k << "*9=" << k*9 << endl << k << "*10=" << k*10 << endl;
}
else cout << "Неправильное значение множителя!" << endl;
```

Несколько операторов, заключенных в фигурные скобки, называются *составным* оператором. В тексте программы составной оператор может стоять везде, где может быть записан один оператор. Если нам потребуется, мы можем задать составной оператор и после `else`.

Теперь разберемся, как писать условие. Если «дословно» переписать условие, написанное в псевдокоде, то на языке C++ оно выглядит так: `1 <= k <= 9`. Однако в таком виде условие работать не будет. Чтобы убедиться в этом, напишем простую программу, код которой приведен в листинге 5.1.

Листинг 5.1. Проверка условий (неправильная)

```
#include <iostream>
using namespace std;
int  main()
{  int k;
   cout <<"Введите целое значение от 1 до 9: "; cin >> k;
   if (1 <= k <= 9)     cout <<"Истинно" <<endl;
   else                 cout <<"Ложно  "<<endl;
   return 0;
}
```

Программа транслируется без ошибок и выполняется. Однако выясняется, что при *любых значениях* переменной `k` всегда выводится слово «Истинно». Дело в том, что операции сравнения левоассоциативны, и выражение `1 <= k <= 9` вычисляется последовательно слева направо: `(1 <= k) <= 9`. Сначала вычисляется выражение `1 <= k`, и результат сравнивается с 9. Так как результатом сравнения является логическое значение `true` или `false`, происходит *неявное преобразование* логической константы в целое число. Если выражение истинно, то результат преобразования равен `1` (единица); если же выражение ложно, то результат — `0` (ноль). Теперь понятно, почему наше условие всегда истинно: и `0 < 9`, и `1 < 9`.

Вместо привычной математической записи мы должны в данном случае написать *логическое выражение*. Для этого сначала нужно разделить наше условие на два отдельных, а затем объединить их с помощью логической операции И. На языке C++ это выглядит так:

```
(1 <= k)&&(k <= 9)
```

Немного переделаем условие в соответствии со стилем, принятым в C++:

```
(0 < k)&&(k < 10)
```

С точки зрения смысла ничего не изменилось, но такое выражение проще писать и легче читать. Окончательный вариант нашей программы умножения представлен в листинге 5.2.

Листинг 5.2. Таблица умножения с проверкой множителя

```
#include <iostream>
using namespace std;
int  main()
{  int k;                                  // объявление переменной
  cout <<"Введите множитель от 1 до 9: "; cin >> k;
  if ((0 < k)&&(k < 10))                   // внешние скобки обязательны
  { cout << k << "*1=" << k*1 << endl << k << "*2=" << k*2 << endl;
    cout << k << "*3=" << k*3 << endl << k << "*4=" << k*4 << endl;
    cout << k << "*5=" << k*5 << endl << k << "*6=" << k*6 << endl;
    cout << k << "*7=" << k*7 << endl << k << "*8=" << k*8 << endl;
    cout << k << "*9=" << k*9 << endl << k << "*10="<< k*10<< endl;
  }
  else cout << "Неправильное значение множителя!" << endl;
  return 0;
}
```

Эту программу вполне можно использовать для изучения полной таблицы умножения от 1 до 9.

Решаем задачи

Напишем несколько программ, в которых условный оператор используется более интенсивно и в разнообразных ситуациях. Сначала заново напишем программу для решения уравнения $ax + b = 0$.

Задача 1. Решить уравнение $ax + b = 0$

Итак, смотрим наш алгоритм на псевдокоде:

```
1. Ввод a и b.
2. Если a не ноль,
        то x = -b / a, вывести x.
3. Иначе: (тут a = 0)
        если и b не ноль, то вывести «решения нет».
4. Иначе: (тут a = 0 и b = 0)
        вывести «x - любое число».
5. Завершить программу.
```

Теперь переводим псевдокод на язык C++. Результат показан в листинге 5.3.

Листинг 5.3. Программа решения уравнения *ax + b = 0* (*if*)

```
#include <iostream>
using namespace std;
int main(){
   double a {0.0}, b{0.0};
   cout << "Задайте a и b: \n";
   cout << "a = "; cin >> a;
   cout << "b = "; cin >> b;
   double x {0.0}; cout.precision(10);
   if(a != 0)
     x = -b/a, cout << x;                    // оператор-выражение
   else if(b == 0)
     cout << "x - любое";                    // оператор-выражение
   else
     cout << "нет решения";                  // оператор-выражение
   cout << endl;
   return 0;
}
```

Программа точно соответствует алгоритму.

Для тестирования программы достаточно трижды запустить программу со следующими парами чисел:

- a = 1, b = 1; результат = –1;
- a = 0, b = 1; результат = нет решения;
- a = 0, b = 0; результат = x любое.

Внимательно изучите код и еще раз вспомните о том, что любое выражение становится оператором, если в конце поставить точку с запятой.

Задача 2. Високосный год

Как известно, високосными годами являются все года, которые делятся на 4 без остатка (остаток равен нулю), за исключением тех, которые делятся на 100. Если год делится на 100, то он является високосным только в том случае, если делится на 400. Слово «делится» здесь означает, что остаток от деления равен нулю. Например, високосными годами являются 1600, 2000, 2004, 1996. А 1900 или 2003 годы високосными не являются.

Таким образом в условном операторе надо проверять, равны ли остатки от деления на 4, 100, 400 нулю или не равны. Очевидно, что переменная, в которой должен храниться год, должна быть целая беззнаковая.

Учитывая все эти соображения, напишем программу (листинг 5.4).

Листинг 5.4. Високосный год

```
#include <iostream>
using namespace std;
int  main(){
    unsigned int year {0};
    cout <<"Задайте год: "; cin >> year;
    if ((year % 100) != 0)
        if ((year % 4) == 0) cout <<"Год високосный!\n";
        else                    cout <<"Год не високосный!\n";
    else
        if ((year % 400) == 0) cout <<"Год високосный!\n";
        else                     cout <<"Год не високосный!\n";
return 0;
}
```

Для проверки программы надо задать четыре варианта значения-года: два високосных и два не високосных. Последовательно введем значения `2000`, `1996`, `1700`, `1997`. В первых двух случаях программа должна сообщить, что год високосный, в остальных — не високосный.

В коде надо обратить внимание на условные операторы. Мы использовали *вложенные* условные операторы. Схема вложенных `if` выглядит так:

```
if (условие) if (условие) оператор; else оператор;
```

Поскольку при такой записи неясно, к какому `if` относится `else`, в языке C++ принято соглашение, что такая запись эквивалентна следующей:

```
if (условие) { if (условие) оператор; else оператор; }
```

При такой записи становится совершенно очевидно, что `else` связывается с ближайшим оператором `if`.

Можно написать операторы со скобками:

```
if ((year % 100) != 0){
    if ((year % 4) == 0) cout <<"Год високосный!\n";
    else                    cout <<"Год не високосный!\n";
} else {
    if ((year % 400) == 0) cout <<"Год високосный!\n";
    else                cout <<"Год не високосный!\n";
}
```

В этом случае явно показано, что в первый оператор `if` вложены два других.

Можно упростить код, если объединить все условия в одном `if`. Тогда можно убрать лишние операторы вывода:

```
if (((year % 100) != 0) && ((year % 4) == 0) || ((year % 400) == 0))
        cout <<"Год високосный!\n";
else    cout <<"Год не високосный!\n";
```

Такая запись, на мой взгляд, тоже весьма проста и понятна.

Булевские, целые и условия

Условия високосности в условных операторах можно записать короче:

```
if (((year % 100) && !(year % 4)) || !(year % 400))
        cout <<"Год високосный!\n";
else    cout <<"Год не високосный!\n";
```

Во-первых, удалены все явные сравнения с нулем. Во-вторых, операция логического НЕ «!» применяется к результату операции деления по модулю. Все объясняется просто. C++ вырос из C, а этот язык не имел логического типа данных. Поэтому в условиях (а это не только условный оператор) использовался тип `int`. Значение 0 интерпретировалось как `false`, а любое другое (ненулевое, в том числе отрицательное) — как `true`.

Логические операции работали с целыми значениями 0 и 1. Например, логическое И выполнялось в соответствии со следующими правилами:

```
0 && 0 = 0
0 && 1 = 0
1 && 0 = 0
1 && 1 = 1
```

Замените в определениях логических операций `false` на 0, `true` на 1 — и вы увидите, как логические выражения работали в C. Язык C++ гарантирует обратную совместимость с C, поэтому в C++ все это продолжает работать. Условный оператор в C++ можно писать так:

```
if(выражение)оператор;
```

Выражение вычисляется, далее оно преобразуется в целое (если требуется, при этом выполняются неявные преобразования), целое приводится к `bool`.

Можно написать `if` и так:

```
if(!выражение)оператор;
```

Здесь после вычисления `выражения` и преобразования в целое (если требуется) работает операция отрицания «!», которая нулевое значение превращает в 1, а ненулевое — в 0. А далее — в `true` или `false`.

Преобразование целых в булевские прописали в стандарте после включения в C++ типа `bool`. Но следы работы с целыми числами в условиях и логических операциях остались. Например, выведем результат операции сравнения:

```
cout << (1 < 2) << endl;
```

Мы увидим в окне единицу, а не `true`. Аналогично можно увидеть единицу в консоли при таком выводе:

```
cout << !(0.0) << endl;
```

В этом операторе вещественный `0.0` преобразуется в целый ноль, потом операция отрицания «!» превращает его в единицу, и мы ее видим на экране. Более того, даже если мы явно напишем в операторе вывода true или false, все равно выведется 1 или 0 — вне условий неявное преобразование целого в булевское не выполняется.

Чтобы вывести `true` или `false`, надо применить манипулятор `boolalpha`:

```
cout << std::boolalpha << (1 < 2) << endl;
cout << std::boolalpha << !(0.0) << endl;
cout << std::boolalpha << true << endl;
```

Никаких дополнительных библиотек подключать не требуется, манипулятор реализован в библиотеке ввода/вывода. Там же реализован противоположный манипулятор `noboolalpha`, который отменяет манипулятор `boolalpha`.

Ныряем глубже

Продолжим изучать условный оператор на примере решения новых задач.

Задача 3. Кирпич и отверстие

Пусть у нас есть кирпич с размерами сторон `a`, `b`, `c`, и есть прямоугольное отверстие размером `x` на `y`. Нужно проверить, пройдет ли кирпич в отверстие.

Очевидно, что кирпич пройдет в отверстие, если хотя бы две из его сторон меньше размеров отверстия. Упорядочим размеры отверстия `(x < y)` и «выстроим» размеры сторон кирпича в возрастающем порядке: `a < b < c`. Тогда для ответа на вопрос достаточно одной проверки:

```
если (a < x) и (b < y), то ответ "ДА", иначе ответ "НЕТ"
```

Выстроить стороны кирпича в нужном порядке можно, обменяв значения переменных, как в следующем псевдокоде:

```
если (a > b) то { t = a; a = b; b = t; }
если (b > c) то { t = b; b = c; c = t; }
если (a > b) то { t = a; a = b; b = t; }
```

Двух перестановок недостаточно. Пусть, например, `a = 10`, `b = 8`, `c = 6`. После первой проверки `a = 8`, `b = 10`, `c = 6`; после второй — `a = 8`, `b = 6`, `c = 10`.

Наши рассуждения «дословно» переводятся в C++ (листинг 5.5).

Листинг 5.5. Кирпич и отверстие

```
#include <iostream>
using namespace std;
int main()
{  double a, b, c;      // размеры кирпича
   cout << "Задайте размеры кирпича: ";   cin >> a >> b >> c;
   double x, y;                 // размеры отверстия
   cout << "Задайте размеры отверстия: "; cin >> x >> y;
   if (x > y) { double t = x; x = y; y = t; }
   if (a > b) { double t = a; a = b; b = t; }
   if (b > c) { double t = b; b = c; c = t; }
   if (a > b) { double t = a; a = b; b = t; }
```

```
    if ((a < x)&&(b < y)) cout << "Кирпич пройдет в отверстие!\n";
    else                  cout << "Кирпич не пройдет в отверстие!\n";
    return 0;
}
```

Анализ кода этой программы весьма поучителен. Самое интересное в этом коде то, что объявление переменной `t` четыре раза не вызывает «протестов» компилятора. А фокус заключается в том, что эта переменная объявляется *внутри составного оператора.* Составной оператор с объявленными внутри переменными по многолетней традиции называют *блоком.* В C++ принято, что в любом месте программы, где может стоять составной оператор, может стоять и блок.

Переменные, объявленные в блоке, считаются *локальными переменными* в блоке. Так как тело функции `main` тоже является блоком, то все переменные, объявленные в теле функции `main`, являются локальными переменными в этой функции. Немедленно возникает вопрос: а можно ли определить переменную вне тела функции `main` (вне блока)? Да, можно. Такие переменные называются *глобальными.*

> ***НА ЗАМЕТКУ***
>
> Мы рассмотрим локальные и глобальные переменные более детально при изучении функций.

Продолжим изучение кода программы. Очевидно, что обмен переменных в программах выполняется регулярно, поэтому такое действие просто обязано быть в стандартной библиотеке. И действительно, в STL есть стандартная функция `swap`, с помощью которой выполняется обмен двух переменных. При использовании `swap` код становится значительно короче:

```
if (x > y) swap(x,y);
if (a > b) swap(a,b);
if (b > c) swap(b,c);
if (a > b) swap(a,b);
```

Для использования этой функции нужно подключить библиотеку `<utility>`.

Никогда так не делайте

Иногда в некоторых книгах или в олимпиадном программировании советуют выполнять обмен без использования третьей переменной. В коде программы это может выглядеть так:

```
int a = 10, b = 3;
a = a + b;              // a = 13, b = 3
b = a - b;              // a = 13, b = 10
a = a - b;              // a = 3,  b = 10
```

В данном случае использовались операции сложения и вычитания. Имеются варианты обмена переменных с помощью умножения и деления, с битовыми операциями, но мы не будем их показывать, а просто обсудим проблемы.

На первый взгляд, все прекрасно! Однако писать обмен таким образом — это искать приключений на свою программистскую голову. Во-первых, это работает только для чисел. Если ваши объекты не являются числовыми (практически всегда так и есть), то с ними нельзя выполнять арифметические действия[1]. Во-вторых, даже с числами могут возникнуть проблемы. Мы уже упоминали о *переполнении*. В операторе, где числа складываются, такое вполне может произойти хоть с целыми, хоть с вещественными.

> ***На заметку***
>
> С вещественными числами есть еще одна проблема — точность. О всех этих проблемах мы обязательно поговорим (см. *уроки 15* и *16*).

Просто никогда так не делайте, чтобы не возникало проблем.

Снова про ввод данных

В задаче про кирпич и отверстие нужно еще обратить внимание на операторы ввода. Вспомним, как мы записывали ввод в программе решения уравнения $ax + b = 0$ (см. листинг 5.3):

```
cout << "Задайте a и b: \n";
cout << "a = "; cin >> a;
cout << "b = "; cin >> b;
```

Сначала выводится общее приглашение и курсор переводится на следующую строку. Выводится приглашение задать ***a***, и выполняется его ввод. После набора числа нажимается клавиша <Enter>, которая переводит курсор на следующую строку, где эти действия повторяются при вводе значения для ***b***.

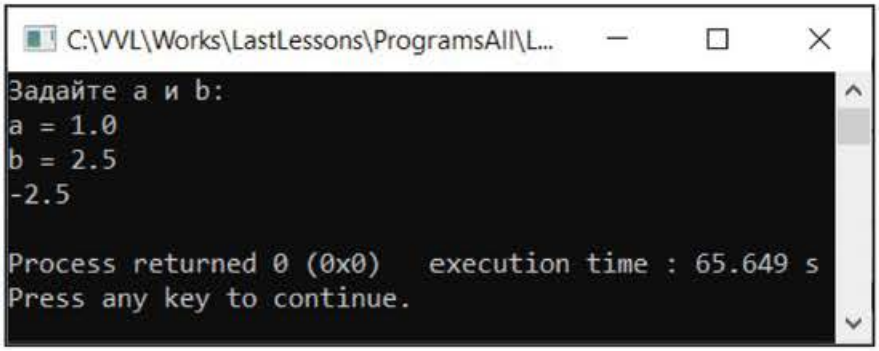


Рис. 5.1. Правильный ввод значений для решения уравнения $ax + b = 0$

Однако пользователь делает ошибки. И вот я, как пользователь, сделал ошибку, и вместо клавиши <Enter> после первого числа набрал пробел, а <Enter> нажал только после второго числа. Посмотрите, как некрасиво это выглядит (рис. 5.2).

---

[1] Справедливости ради нужно заметить, что имеется возможность перегрузить стандартные операции, и в принципе такие операции можно реализовать для нечисловых объектов.

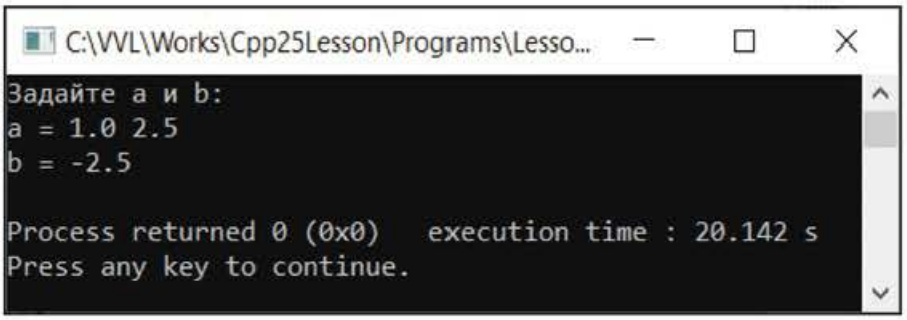


Рис. 5.2. Ошибочный ввод значений через пробел для решения уравнения *ax + b = 0*

Хотя программа работает совершенно правильно, вывод несколько обескураживает. Как будто задача ничего не сделала и просто закончила работу без видимого результата.

Напомню, что мы уже вводили данные через пробел, когда писали программу приветствия пользователя по имени (см. листинг 3.2). Там не было подсказок по каждой переменной. Точно так же написан ввод размеров отверстия и кирпича в программе из листинга 5.5. Написано одно общее приглашение, и после него оператор ввода:

```
cout << "Задайте размеры кирпича: "; cin >> a >> b >> c;
```

В этом случае вполне можно задавать значения при вводе через пробел. И это будет выглядеть, как показано на рис. 5.3.

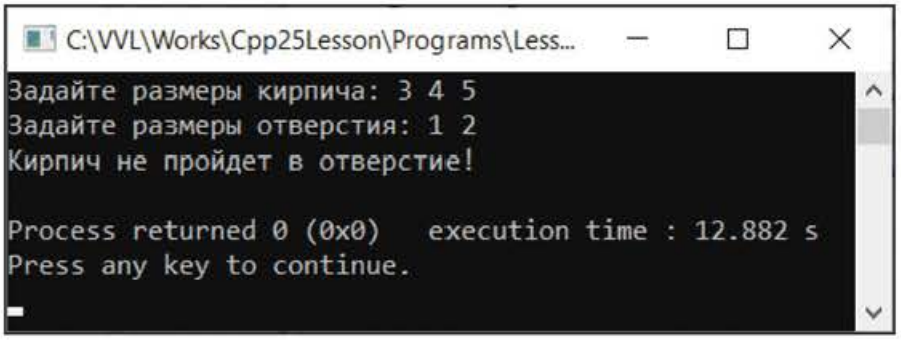


Рис. 5.3. Ввод значений через пробел

Клавиша <Enter> нажимается при таком вводе только один раз после набора последнего значения. При вводе размеров кирпича — после третьего, а при вводе размеров отверстия — после второго.

Такой ввод легче читать. При этом ошибок совершается меньше, т. к. пользователь может после каждого числа набрать либо пробел, либо нажать клавишу <Enter>.

Таким образом, при написании операторов ввода/вывода нужно заранее продумывать их последовательность. Система потокового ввода/вывода предоставляет относительно простые возможности, поэтому и программист должен в коде писать операторы ввода по возможности просто, не перемежая ввод отдельных переменных выводом приглашений.

Задачи и упражнения

1. Написать программу, которая по заданным трем сторонам вычисляет площадь треугольника. Для вычисления площади использовать формулу Герона (найдите в Интернете). Программа должна проверять существование треугольника. Реализовать три способа проверки — каждую в отдельной программе.

   Треугольник существует, если выполняется неравенство треугольника: любая сторона меньше суммы двух других сторон.

   Первый способ: написать несколько вложенных друг в друга операторов полного ветвления, каждый из которых проверяет, выполняется ли для данной стороны неравенство треугольника.

   Второй способ: написать один оператор ветвления, в котором проверяется составное условие со всеми сторонами сразу.

   Третий способ: можно сначала упорядочить стороны и проверить, что самая длинная сторона меньше суммы двух оставшихся.

2. Даны три вещественных числа A, B, C. Найти x_1 и x_2 — корни квадратного уравнения $Ax^2 + Bx + C = 0$ или сообщить, что уравнение не имеет решения.

   Подсказка: формула корней квадратного уравнения $x_{1,2} = \frac{-b \pm \sqrt{b^2 - 4a \cdot c}}{2a}$.

   Если выражение под знаком корня (дискриминант) отрицательно, то уравнение не имеет решения. При A = 0 уравнение не является квадратным, можно сообщить об этом и не решать.

3. Написать программу, которая проверяет, находится ли заданная точка внутри заданного прямоугольника. Даны точки A (Xa, Ya) и C (Xc, Yc) — вершины прямоугольника ABCD, стороны которого параллельны координатным осям. Проверить, существует ли такой прямоугольник (прямоугольник не будет существовать, если совпадают абсциссы или ординаты точек A и C). Если прямоугольник существует, выяснить, принадлежит ли точка с координатами (x, y) этому прямоугольнику (считаем, что точка принадлежит прямоугольнику, если она лежит внутри или на одной из сторон).

4. Составить наборы тестовых данных для всех программ этого урока.

- Урок 6 -

Повторение — мать учения

Регулярно встречающаяся в жизни и в программировании ситуация — надо много раз повторять одни и те же вычисления с разными данными. Например, бухгалтер вычисляет заработную плату множества работников в организации, выполняя совершенно однотипные расчеты (но с разными данными) для каждого работника.

Конструкции повторения вычислений с давних времен входят во все без исключения языки программирования. В C/C++ эти конструкции составляют половину всех операторов управления. По традиции эти операторы называют операторами цикла или просто циклами.

Но они появились в языках программирования не сразу. Например, в первых версиях языка Fortran их не было. Однако повторять вычисления было надо, поэтому, естественно, языки программирования содержали более простые операторы, с помощью которых это можно было делать. В C/C++ такие операторы тоже входят, т. к. это наследство C. Мы рассмотрим на конкретных примерах кода и эти операторы тоже.

Повторяем работу

Вспомним программу для определения високосности года (см. листинг 5.4). Хотелось бы при одном запуске программы задавать год не один, а несколько раз. А когда вычислять високосность года больше не требуется, каким-то образом сообщить программе, что надо завершить работу.

Чтобы сообщить программе, что не нужно вычислять високосность года, можно просто задать `0` при вводе и прописать в коде условный оператор, проверяющий заданный год на ноль:

```
Вывести "Задайте год", ввести год
Если год > 0, то
   Выполнить проверку високосности
```

В противном случае программа просто завершается.

Но это однократное действие, а нам требуется после проверки високосности года каким-то образом опять попросить пользователя задать год, потом снова его проверить и т. д. Обратите внимание, что написать в коде еще раз то же самое — это не

решение проблемы. Тогда надо тот же код писать и третий раз, и четвертый, и пятый. Но нам нужно другое: повторить выполнение кода в зависимости от значения введенного года.

Как обычно, сначала напишем алгоритм. Нам требуется делать примерно следующее:

```
1. Начало.
2. Вывести "Задайте год", ввести год.
3. Если год > 0, то
   3.1. Если год високосный,
   то вывести "Год високосный!"
   иначе вывести "Год не високосный!"
   3.2. Перейти к началу.
4. Завершить работу.
```

Если «дословно» написать этот алгоритм на C++, то получится код, показанный в листинге 6.1.

Листинг 6.1. Високосный год (*goto*)

```
#include <iostream>
using namespace std;
int  main(){
    unsigned int year {0};
    cout << "Определение високосности года\n";
    cout << "Для завершения работы введите 0.\n";
    start:                    // метка
     cout << "Задайте год: "; cin >> year;
     if(year > 0){
        if ( ((year % 100)&&(!(year % 4))) || (!(year % 400)) )
              cout << "Год високосный!\n";
        else    cout << "Год не високосный!\n";
        goto start;           // переход на метку
    }
    cout << "Заканчиваем работу\n";    return 0;
}
```

Идентификатор `start` («начало»), после которого поставлено двоеточие, называется меткой. В качестве метки можно указывать любой допустимый идентификатор. Этот идентификатор предварительно никак не объявляется. Двоеточие после него и указывает, что этот идентификатор является меткой.

В блоке условного оператора записан оператор `goto` («иди к»), в котором эта метка и задана. Оператор `goto` так и называется: оператор *перехода*. И его действие заключается именно в том, что программа возвращается к метке `start` и работа начинается заново.

Выполним программу и определим високосность 2000, 1997, 1996 и 1900 годов (рис. 6.1).

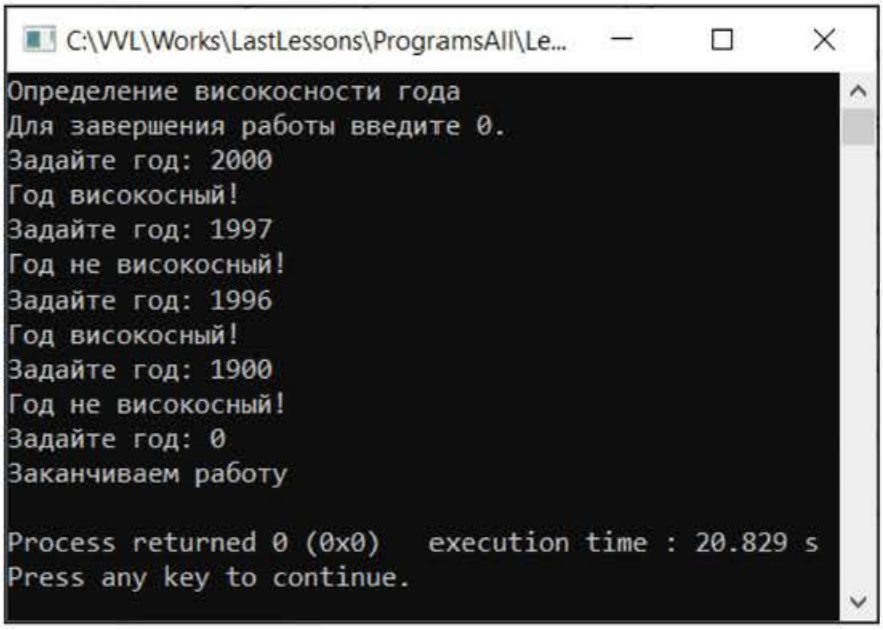


Рис. 6.1. Выполнение программы определения високосности года (`goto`)

Как можно видеть, программа работает совершенно правильно и заканчивает работу, как только пользователь задаст `0` при вводе года.

Оператор `goto` входил во все языки программирования без исключения и использовался как раз для организации повторения выполнения. Однако если открыть учебники по программированию, то можно обнаружить, что многие авторы не рекомендуют для организации циклов использовать оператор `goto`. Более того, в некоторых языках программирования, появившихся в конце 20-го века, оператор `goto` отсутствует (например, в Java). Но я вполне допускаю использование оператора перехода при определенных условиях.

А теперь перейдем к изучению операторов цикла.

Операторы цикла

Перепишем программу «Високосный год» с использованием оператора цикла. В C++ есть четыре оператора цикла, и начнем мы с самого простого.

Оператор *while*

Этот оператор называется *цикл с предусловием* и имеет вид:

```
while (условие) оператор;
```

Формат оператора фактически совпадает с форматом оператора `if`, только первое зарезервированное слово новое, которое переводится как «пока». Этот оператор цикла так и работает: *пока* условие истинно, выполняется заданный оператор. Та-

кое условие будем называть *условием продолжения* (цикла). Как только условие стало ложным, заданный оператор пропускается и начинает работать следующий после цикла оператор.

Условие записывается и вычисляется точно так же, как и в операторе `if`. И точно так же на месте единственного оператора можно прописать составной оператор в фигурных скобках, который обычно называют *телом* цикла.

Код новой программы для определения високосности года с оператора `while` представлен в листинге 6.2.

Листинг 6.2. Високосный год (*while*)

```
#include <iostream>
using namespace std;
int  main(){
    unsigned int year {0};
    cout << "Для завершения работы введите 0.\n";
    cout << "Задайте год: "; cin >> year;
    while(year > 0){
        if ( ((year % 100)&&(!(year % 4))) || (!(year % 400)) )
             cout << "Год високосный!\n";
        else cout << "Год не високосный!\n";
        cout << "Задайте год: "; cin >> year;
    }
    cout << "Заканчиваем работу\n";
    return 0;
}
```

В коде надо обратить внимание на два оператора ввода: первый написан перед циклом, а второй — последним в теле цикла. Первый оператор нужен, чтобы условие в операторе цикла было проверено первый раз. Если пользователь задаст год для проверки високосности, то условие будет истинно и программа *войдет в цикл*.

В теле цикла выполнится условный оператор, определяющий високосность заданного года, и после него при выполнении оператора ввода будет задан следующий год. Тело цикла закончилось, операторов в нем больше нет, и *управление возвращается* к проверке условия с *новым значением* года. Если был задан год для проверки, то опять будет выполнено тело цикла. При выполнении последнего оператора в теле опять будет введено число.

Если это число равно нулю, то условие становится ложным и цикл заканчивается: *управление* передается к следующему после тела цикла оператору, который выводит сообщение о завершении программы.

Отметим роль последнего оператора ввода в теле цикла: его работа состоит в том, чтобы изменить данные для условия, проверяемого при следующем повторении цикла. Каждое повторение цикла называется *шагом* цикла или *итерацией*.

Опять таблица умножения

Вернемся к программе «Таблица умножения с проверкой множителя» (см. листинг 5.2). Разъединим объединенные операторы вывода:

```
cout << k << "*1=" << k*1 << endl;
cout << k << "*2=" << k*2 << endl;
...
cout << k << "*9=" << k*9 << endl;
cout << k << "*10="<< k*10<< endl;
```

Они отличаются один от другого всего в двух местах: в текстовой константе (`"*1="`, `"*2="`) и в выражении умножения (`k*1`, `k*2`). Хотелось бы заменить десять очень похожих операторов одним-единственным, который отработает за все десять.

Разумеется, это можно сделать с помощью оператора цикла `while`. Однако этот цикл несколько отличается от того, который мы писали для определения високосности года (см. листинг 6.2). Во-первых, в программе високосности количество итераций может быть какое угодно (пока пользователь не задаст `0`), а для таблицы умножения количество итераций ограничено — 10 раз. Во-вторых, на каждом шаге цикла требуется в операторе вывода изменить текстовую константу и второй множитель в выражении умножения.

Для решения этих проблем мы заменим явную целую константу переменной. Эта переменная должна увеличиваться на 1 при каждом повторении. Назовем нашу переменную именем `i`. Значение переменной `i` представляет собой номер итерации. Обычно такие переменные называют счетчиками цикла.

> ***На заметку***
>
> В программировании существует очень давняя традиция именовать счетчики (которые являются переменными целых типов) буквами i, j, k, l, m, n, и мы будем ее придерживаться.

Схема цикла на псевдокоде выглядит так (будем выводить таблицу умножения от 1 до 9, как это напечатано на обложке тетрадей):

```
i = 1;
пока (i < 10){
   вывод одной строки таблицы умножения
   увеличить i на 1
}
```

Как и в предыдущей программе (см. листинг 6.2), *перед* циклом выполняется действие, которое необходимо для первой проверки условия и входа в цикл. В теле цикла сначала выполняются действия, которые требуется повторять, а *последнее* действие в цикле меняет данные, чтобы на следующей итерации проверка условия выполнялась с новыми данными. Как видим, схема цикла фактически та же, что и в программе проверки високосности.

Наша схема на C++ запишется так:

```
i = 1;                          //-- начальное значение
while (i < 10)                  //-- условие продолжения
```

```
{ cout << k << " * " << i << " = " << k*i << endl;
  ++i;                          //-- увеличение переменной на 1
}
```

Мы преобразовали текстовую константу в операторе вывода: по сравнению с первоначальным вариантом от нее осталось только два неизменных символа: звездочка и равно, — остальное превратилось в переменные, которые могут иметь разные значения в разных итерациях цикла.

Теперь можно написать полную программу с циклом. Используем первый вариант цикла (листинг 6.3).

Листинг 6.3. Таблица умножения с циклом *while*

```
#include <iostream>
using namespace std;
int main()
{   int k;                              //-- объявление переменной
    cout << "Введите множитель от 1 до 9: "; cin >> k;
    if ((0 < k) && (k < 10))            //-- внешние скобки обязательны
    { int i = 1;                        //-- начальное значение
      while (i < 10)                    //-- условие продолжения
      { cout << k << "*" << i << "=" << k*i << endl;
        ++i;                            //-- увеличение переменной на 1
      }
    }
    else cout << "Неправильное значение множителя!\n";
    return 0;
}
```

Как и в предыдущей версии (см. листинг 5.2), на экран выводится приглашение, и программа ждет ввода значения. Если задано недопустимое число, то программа выводит сообщение об ошибке и заканчивает работу. При вводе правильного значения выполняется блок после условия оператора `if`.

Сначала объявляется переменная `i`, которая сразу же получает значение 1 — это первая итерация. Потом оператор цикла проверяет условие, которое в данном случае истинно (1 < 10). Следовательно, выполняется тело цикла.

В теле сначала работает оператор вывода. Первым выводится значение переменной `k`, потом (без пропусков) — символ «*», значение `i` (в данном случае — единица), знак «=» (равно) и значение выражения `k*i`. Курсор экрана переводится на следующую строку.

Потом значение переменной `i` увеличивается на 1 и становится равным 2 — работает оператор инкремента `++i;`. На этом месте тело цикла заканчивается, и *управление возвращается* к проверке условия. Условие (2 < 10) истинно, поэтому повторяется только что описанный процесс.

И так продолжается до тех пор, пока при очередном увеличении переменная `i` не станет равна 10. Тогда при проверке (10 < 10) условие окажется ложным (10 не больше 10) и произойдет *выход из цикла.* И сразу же — выход из оператора `if`. Выполнится оператор `return` — программа завершит работу.

Оператор *do while*

В C++ включен другой оператор цикла, в котором условие проверяется *после* выполнения тела цикла. Форма записи цикла с *постусловием* следующая:

```
do оператор; while(условие);
```

Как обычно, на месте единственного оператора можно задать составной оператор в фигурных скобках.

Важное отличие оператора цикла с постусловием цикла `while` состоит в том, что в данном случае тело цикла обязательно выполнится хотя бы один раз. А при вычислении предусловия может возникнуть такая ситуация, что тело цикла не сработает ни разу. Само условие вычисляется точно так же.

Наша программа с циклом `do while` — в листинге 6.4.

Листинг 6.4. Таблица умножения с циклом *do while*

```
#include <iostream>
using namespace std;
int main()
{  int k;                                  // объявление переменной
   cout << "Введите множитель от 1 до 9: "; cin >> k;
   if ((0 < k) && (k < 10))                // внешние скобки обязательны
   {  int i = 1;                           // начальное значение
      do { cout << k << "*" << i << "=" << k*i << endl;
           ++i;                           // увеличение переменной на 1
      } while (i < 10);                    // условие продолжения
   }
   else cout << "Неправильное значение множителя!" << endl;
   return 0;
}
```

Как мы видим, программа фактически не изменилась, кроме непосредственно оператора цикла. Однако надо отметить, что работа этого оператора цикла несколько отличается от цикла с предусловием. Вывод первой строки таблицы умножения выполняется безусловно, без проверки условия. Только после увеличения `i` (`i` становится равно 2) выполняется проверка условия и происходит переход в начало составного оператора, где снова выполняется оператор вывода.

Оператор *for* со счетчиком

Рассматривая оба варианта программы про таблицу умножения, можно заметить, что организация цикла в обеих программах имеет некоторые сходные черты:

1. И в том, и в другом варианте у нас есть переменная-счетчик, которой один раз присваивается начальное значение.
2. Эта переменная увеличивается при каждой итерации, и шаг увеличения равен 1.
3. На каждом шаге цикла выполняется проверка условия.

Практика программирования показала, что такие циклы встречаются в программах очень часто. Поэтому разработчики C включили в язык третий вид оператора цикла, который так и называется — *цикл со счетчиком*:

```
for(выражение_1; условие; выражение_3) оператор;
```

Точки с запятой внутри круглых скобок являются обязательными, и при их отсутствии возникают ошибки компиляции.

`Выражение_1` предназначено для инициализации счетчика цикла. `Условие` — это такое же условие, как и в других операторах цикла. Как обычно, условие может быть произвольным выражением, результат которого приводится к `bool`. `Выражение_3` — это то, что делает в наших циклах оператор инкремента `++i;`. Но на этом месте в операторе может быть задано любое допустимое выражение.

Естественно, вместо единственного оператора в теле цикла можно писать составной оператор в фигурных скобках.

Этот оператор цикла работает так:

- вычисляется `выражение_1`; это происходит единственный раз при первом выполнении оператора `for`;
- проверяется `условие`; если условие истинно, то выполняется заданный оператор в теле цикла; если условие ложно, то происходит выход из цикла и выполняется следующий после цикла оператор;
- после выполнения заданного оператора работает `выражение_3`;
- происходит переход к проверке условия.

Таким образом, при первой итерации выполняются первые два «пункта» в круглых скобках, а при последующих — второй и третий, только в обратном порядке. Как можно видеть, это ровно та же последовательность действий, которую мы реализовали с использованием цикла `while`. С оператором `for` наша программа выглядит так (листинг 6.5).

Листинг 6.5. Таблица умножения с циклом *for*

```
#include <iostream>
using namespace std;
int  main()
{ int k; cout << "Введите множитель от 1 до 9: "; cin >> k;
```

```
  if ((0 < k)&&(k < 10))
     for(int i=1; i < 10; ++i)
          cout << k << " * " << i << " = " << k*i << endl;
  else cout << "Неправильное значение множителя!" << endl;
  return 0;
}
```

Программа стала значительно короче. У нас исчезли составные операторы с их скобками — программу стало легче читать. Отсюда вывод: если у вас в программе есть цикл со счетчиком, то используйте оператор `for`.

Решаем задачи

После изучения основ написания циклов перейдем к практике и порешаем задачи, в которых без использования циклов не обойтись.

Полная таблица умножения

Напишем, наконец, полную таблицу умножения, в которой будут выведены все произведения. Нужно вывести таблицу из 9 строк и 9 столбцов. Элемент таблицы — это произведение номера строки на номер столбца:

$a_{ij} = i \times j$, *где* $2 \le i, j \le 9$.

Таким образом, надо написать цикл в цикле: в каждой строке нужно вывести 9 произведений, и всего таких строк 9. Кроме самих произведений нужно еще вывести номера строк и столбцов: номера столбцов — в шапке, а номера строк — в каждой строке. Код представлен в листинге 6.6.

Листинг 6.6. Полная таблица умножения

```
#include <iostream>
#include <iomanip>
#include <string>
using namespace std;
int main(){
   cout << "Полная таблица умножения\n"
        << " |\t 2\t 3\t 4\t 5\t 6\t 7\t 8\t 9\n"
        << string(66,'-') << endl;              // 66 минусов
   for (int i = 1; i < 10; i++)
   {  cout << i << "| ";                        // номер строки
      for (int j = 2; j < 10; j++) cout << '\t' << setw(2) << j*i ;
      cout << endl;
   }
   cout << endl;
   return 0;
}
```

Запустим программу на выполнение. Результат в виде довольно стройной таблицы представлен на рис. 6.2.

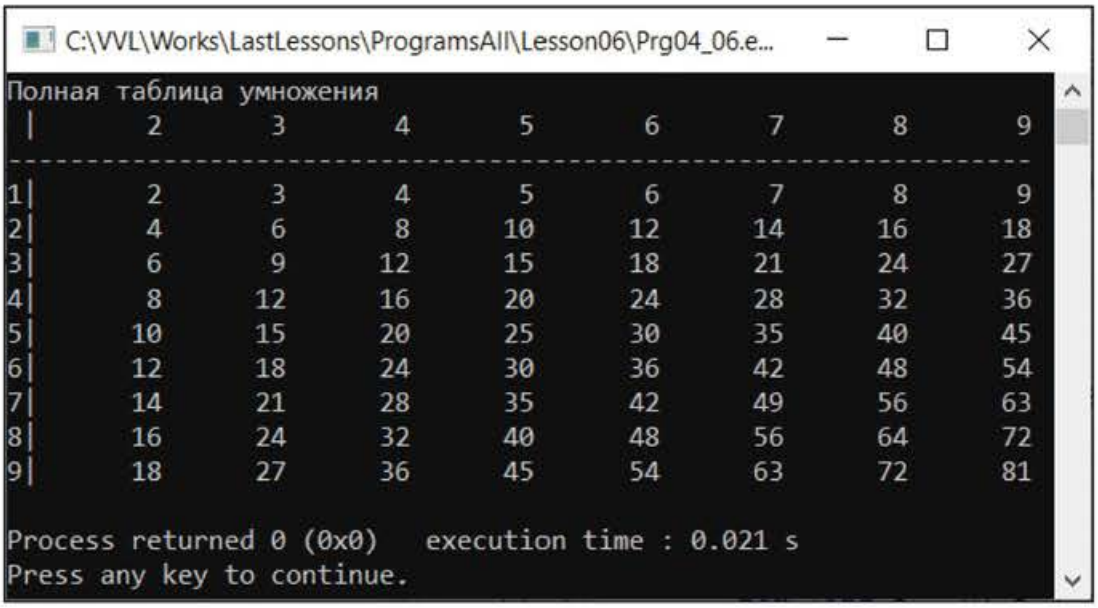


Рис. 6.2. Полная таблица умножения

Посмотрите, как вывод в консоли связан с операторами вывода в коде. Перед циклом задан оператор вывода заголовка с номерами столбцов. В этой строке используются специальные символы табуляции `\t`, чтобы каждый столбец выводился в очередной позиции табуляции и таблица выглядела стройно.

После этого выводится строка минусов, чтобы отделить «шапку» от строк с содержимым. Строка представляет собой временный объект типа `string`. Тут я поступил просто, подобрав нужное количество минусов.

Внешний цикл управляет выводом строк: счетчик `i` изменяется от 1 до 9 (`i < 10`). В теле этого цикла два оператора. Сначала выводится номер строки `i`, а затем оператор цикла выводит произведения `i*j` для `j` от 2 до 9 (`j < 10`). Чтобы все выглядело красиво, используется манипулятор `setw(2)`, который устанавливает ширину вывода в две позиции.

Возведение в степень

В языке C++ отсутствует операция возведения в степень. Попробуем написать программу, которая это делает. Хотя в стандартной математической библиотеке `<cmath>` для этого реализована функция `pow`, но она слишком универсальна. Наша задача более проста: надо вещественное число возвести в целую неотрицательную степень.

Пусть x — любое вещественное число, n — неотрицательное целое число. Наша программа должна вычислять значение x^n по следующим правилам:

- $0^n = 0$ для любого $n > 0$;
- $1^n = 1$ для любого n;

- $x^0 = 1$ для любого x, в том числе $0^0 = 1$;
- $x^n = x \times x \times \ldots \times x$, для $n > 0$.

Программа сначала должна проверить степень `n` на равенство нулю, потом проверить, не равно ли `x` нулю или единице. Если нет, то в цикле вычислить произведение x^n. На псевдокоде это выглядит так:

```
v = 1.0          -- результат
если n > 0 то
{    если x == 0.0 то v = 0.0
     а если x == 1.0 то v = 1.0
     иначе цикл вычисления v = x^n
}
```

В переменную `v` заносится результат возведения `x` в степень `n`.

Код показан в листинге 6.7.

Листинг 6.7. Возведение в степень

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;
int  main(){
   double x = 2;                                  // число
   uint32_t n = 7;                                // степень
   double v = 1.0;                                // результат
   if(n > 0){
     if(x == 0.0) v = 0.0;
     else if(x == 1.0) v = 1.0;
     else while(n){ v *= x; --n; }
   }                                              // если n = 0, то v = 1.0
   cout.precision(12);  // количество выводимых цифр результата
   cout << setw(12) << v << endl;                 // наш результат
   cout << setw(12) << pow(2,7) << endl;          // стандартная функция
   return 0;
}
```

Псевдокод фактически переписан на C++ дословно. Я не стал писать ввод данных, просто объявив переменные с инициализацией. В конце программы для проверки выводятся два значения: наш результат возведения в степень и результат работы стандартной функции возведения в степень. Как обычно, для лучшего представления результатов используются манипуляторы.

Заметим, что мы написали не самый эффективный алгоритм возведения в степень. Но эффективность алгоритмов — это отдельная очень большая тема, которая сама требует написания отдельной книги.

Для тестирования программы необходимо задать начальные значения x и n, проставить те же значения в вызов функции pow и запустить программу на выполнение. Пары чисел для тестирования показаны в табл. 6.1. К ним можно добавить еще возведение нечетного числа (например, 3) в четную и нечетную степень. Но и этих данных вполне достаточно, чтобы быть уверенным в правильности программы.

Таблица 6.1. *Тесты для программы возведения в степень*

x	n	Результат
0	0	1
0	1	0
1	0	1
1	3	1
2	0	1
2	1	2
2	6	64
2	7	128
2	32	4 294 967 296

Обратите внимание, аргументы в вызове функции pow должны совпадать с начальными значениями переменных x и n, объявленными выше. Но вы можете написать ввод этих значений — это трудностей не представляет.

Вариации на тему циклов

Операторы циклов можно чрезвычайно гибко использовать. Например, можно цикл for написать очень похоже на цикл while. Цикл for в коде из листинга 6.5 (таблица умножения) можно написать так:

```
for(int i = 1; i < 10;){
  cout << k << " * " << i << " = " << k*i << endl;
  ++i;
}
```

Здесь мы вынесли выражение_3 из оператора for в тело цикла. Точку с запятой пропускать нельзя, она требуется по синтаксису оператора for.

Можно пойти еще дальше и тот же цикл написать практически совсем похожим на цикл while:

```
int i = 1;
for(;i < 10;){
  cout << k << "*" << i << "=" << k*i << endl;
  ++i;
}
```

Так, конечно, никто не пишет, но это демонстрирует, что разница между операторами чисто синтаксическая.

В C++ можно поступать и наоборот, т. е. оператор вывода можно из тела перенести в оператор цикла. При использовании операции `запятая` можно написать этот же цикл следующим образом:

```
for(int i{1}; i < 10;
    cout << k << " * " << i << " = " << k*i << endl,    // не оператор
    ++i
);                                  // пустое тело
```

В этом цикле тело цикла пустое, без операторов. Вернее, в теле записан *пустой* оператор (точка с запятой). Зато на месте `выражения_3` записаны два выражения — это два операнда операции `запятая`. Причем порядок записи имеет значение.

Оператор цикла работает так: выполняется `выражение_1` — инициализируется счетчик цикла `i`; потом проверяется условие — оно истинно; выполняется тело, в котором нет операторов; управление возвращается в оператор цикла и выполняется `выражение_3`, в котором сначала выполняется первое выражение (вывод); затем работает второй операнд операции `запятая` — увеличивается переменная-счетчик; выполняется пустое тело, и управление возвращается в оператор цикла на проверку условия. И т. д.

Напишем более серьезную программу с применением операции `запятая`: вычисление среднего арифметического последовательности чисел, которые пользователь вводит с клавиатуры. Нам потребуется переменная для суммы и переменная для количества введенных чисел. Программа должна работать для любого количества чисел, и мы заранее не знаем, сколько их будет. Числа могут быть как положительные, так и отрицательные (листинг 6.8).

Листинг 6.8. Цикл *while* с операцией *запятая*

```
#include <iostream>
using namespace std;
int main()
{  unsigned int counter {0};
   double summa = 0.0, number;
   while (cout << "Введите число: ", cin >> number)     // условие
      summa+=number, ++counter;                        // один оператор
   cout << "Среднее =" << summa/counter << endl;
   return 0;
}
```

В программе написан совершенно необычный оператор цикла. В условии прописаны вывод приглашения и ввод числа — два операнда операции `запятая`. Оператор цикла будет выполняться до тех пор, пока пользователь не нажмет специальную комбинацию клавиш <Ctrl>+<Z> (и потом <Enter>).

На заметку

Эта комбинация работает в системе Windows. В системах Linux завершение ввода осуществляется с помощью <Ctrl>+<D> (и потом <Enter>). Буква D (как и Z в Windows) может быть как заглавной, так и строчной.

Эта комбинация означает «конец ввода»: результат выполнения ввода (второе выражение в условии) станет равным `false`, и оператор цикла прекратит свою работу.

Тело цикла содержит один оператор, в котором, однако, вычисляется два выражения. В первом выражении накапливается сумма введенных чисел, а во втором — подсчитывается их количество. После окончания выполнения цикла вычисляется и выводится среднее арифметическое.

Операторная область видимости

Теперь напишем ту же программу вычисления среднего арифметического с использованием оператора `for`. Код представлен в листинге 6.9.

Листинг 6.9. Цикл *for* с операцией *запятая*

```
#include <iostream>
using namespace std;
int main()
{  unsigned int counter = 0;
   for(double summa = 0.0, number;                    // это не оператор
   (cout << "Введите число: ", cin >> number);        // это не оператор
   summa+=number, ++counter);                         // пустое тело
   cout << "Среднее = " << summa/counter << endl;     // ошибка!
   return 0;
}
```

Однако запуск на выполнение приводит к ошибке компиляции на операторе вывода среднего арифметического:

```
error: 'summa' was not declared in this scope
```

Дело в том, что переменные, объявленные в `выражении_1`, являются *локальными* в пределах оператора цикла `for` и его тела. Вне цикла они не видны, поэтому возникает ошибка, которая переводится так: `summa` не объявлена в данной области видимости (scope). То же относится и к переменной `number`. Надо вынести обе переменные из оператора `for`:

```
double summa {0.0}, number;
for(;(cout<<"Введите число: ",cin>>number);     // это не оператор
    summa+=number, ++counter);                  // пустое тело
```

В таком виде программа работает нормально. Завершение программы происходит аналогично предыдущей: пользователь нажимает комбинацию клавиш <Ctrl>+<Z> и потом <Enter>.

Локальная область видимости существует не только в операторе цикла `for`, но и в условии оператора цикла `while`, и в условии оператора `if`, например:

```
if (int y = 1) cout << y << endl;
```

Допускается и универсальная форма инициализации:

```
if (int y{1}) cout << y << endl;
```

Можно в качестве типа использовать `auto`:

```
if (auto x{2.0}) cout << x << endl;
```

Во всех этих случаях условием является сравнение объявленной переменной с нулем. Однако разрешается после объявления переменной поставить точку с запятой и написать явное выражение, например:

```
auto const var {1e30F};
if(auto x{var}; x > 1.0) cout << x << endl;
```

Заметим, что точно так же можно писать условие и в операторе `while`.

Область действия такой переменной — оператор `if`, причем как «ветка» `true`, так и «ветка» `false`. После завершения условного оператора попытки использования переменной с локальной областью видимости приводят к ошибкам компиляции типа «необъявленная переменная», как это мы видели выше в листинге 6.9.

С другой стороны, попытки объявить переменную с тем же именем в теле цикла или в одном из блоков условного оператора вызывают ошибку типа «повторное определение переменной».

Операторы *break* и *continue*

Операторы цикла можно использовать при проверке данных, задаваемых пользователем с клавиатуры. Идея такова: цикл должен выполняться до тех пор, пока пользователь не задаст правильное число. Опять обратимся к нашей хорошо знакомой задаче про вывод таблицы умножения.

Модифицируем код из листинга 6.5: заменим оператор `if` на оператор цикла `do while`:

```
int k;
do { cout << "Введите множитель от 1 до 9: "; cin >> k;
} while(!((0 < k)&&(k < 10)));
```

Если пользователь сразу задаст значение множителя от 1 до 9, то условие цикла станет ложным и произойдет выход из цикла. В противном случае цикл будет выполняться до тех пор, пока пользователь не задаст правильный множитель.

Иногда для решения подобных задач удобнее написать *бесконечный* цикл. Однако из цикла надо выходить, если число введено правильно. В прежние времена для этого использовался уже известный нам оператор перехода `goto`. Но поскольку в конце 60-х годов прошлого столетия этот оператор был признан «вредным», в языки программирования стали включать средства, позволяющие выходить из блока вниз по коду.

В C включили оператор `break`, но оператор `goto` исключать не стали.

Поэтому можно пользоваться любым из способов (при использовании `goto` не забудьте поставить в коде метку). Посмотрите, как оператор `break` можно использовать в нашей программе с таблицей умножения:

```
int k;
for(;;){                                          // бесконечный цикл
   cout << "Введите множитель от 1 до 9: "; cin >> k;
   if((0 < k)&&(k < 10)) break;                   // выход из цикла вниз
}
```

В этом операторе выход из цикла происходит при проверке значения `k`.

Бесконечный цикл можно задать и по-другому:

```
while(1) { ... }
```

или

```
while(true) { ... }
```

Точно таким же образом можно использовать и цикл с постусловием.

Оператор `continue` выполняет переход на условие цикла. Поэтому в циклах `while` и `for` выполняется переход вверх, а в цикле `do while` — вниз.

Замечу, что я никогда не использовал оператор `continue` в своей практике, из-за того, что его поведение меняется в зависимости от вида цикла. Лично мне это очень затрудняет понимание кода. Если уж приходится писать программу с переходами (что тоже не рекомендуется), то я предпочитаю использовать оператор `goto`, т. к. для него нужно написать метку, которая явно показывает, куда выполняется переход.

Поэтому мы не будем рассматривать работу этого оператора.

Задачи и упражнения

1. Квадрат трехзначного числа оканчивается тремя цифрами, которые как раз и являются этим числом. Написать программу, которая находит все такие трехзначные числа.
2. Для заданного натурального n найти все тройки чисел a, b, c, для которых выполняется равенство $n = a + b + c$.
3. Написать программу, которая определяет, делится ли сумма цифр заданного числа N на k без остатка; $2 \le N \le 1000000000$, $2 \le k \le 100$.
4. Сумма цифр трехзначного числа кратна 7, само число также делится на 7. Найти все такие числа.
5. Дано натуральное число $2 \le N \le 15$. Найти сумму $2^2 + 2^3 + \ldots + 2^N$.
6. Факториалом натурального числа N (обозначается N!) называется произведение всех натуральных чисел от 1 до N включительно.

Математики договорились, что факториал нуля равен 1.

```
0! = 1
1! = 1
2! = 1·2 = 2
3! = 1·2·3 = 6
N! = 1·2·3·...·(N - 1)·N = (N - 1)!· N
```

Задано натуральное число N. Написать программу, которая позволяет:

- найти N!;
- найти 1! + 2! + 3! + ... + N!;
- найти наибольшее число M такое, что M! ≤ N.

7. Старинная задача. Сколько можно купить быков, коров и телят при условии, что плата за быка 10 рублей, за корову — 5 рублей, за теленка — полтинник (0,5 рубля). На 100 рублей надо купить 100 голов скота.

8. FizzBazz

   Дано натуральное число N. Вывести в столбик все натуральные числа, не превосходящие N, а справа от числа надписи:

   - `Fizz`, если число без остатка делится на 5;
   - `Bazz`, если число без остатка делится на 3;
   - `FizzBazz`, если число делится без остатка и на 3, и на 5.

   Если число не делится ни на 3, ни на 5, выводится само число.

   Пример для N = 8:

```
1
2
3 Bazz
4
5 Fizz
6 Bazz
7
8
```

- Урок 7 -

Функции

Подпрограммы появились, несомненно, во «второй день творения». Это было так давно, что современные программисты уже практически забыли сам термин «подпрограмма». Тем не менее это настолько важное понятие, что без него невозможно представить ни один серьезный язык программирования.

Писать *действительно* большую программу одному (например, Linux) — очень долго. Такие большие программы можно написать только коллективно. Нужно разделить программу на части и поручить разработку отдельных частей разным программистам.

Поэтому в языках программирования нужны средства, с помощью которых можно выделить часть программы в отдельную конструкцию. Исторически первой такой конструкцией были подпрограммы. В прежние времена языки программирования различали два вида подпрограмм: функции и процедуры. В C++, который унаследовал механизм подпрограмм из C, процедур нет и функции играют обе роли.

Второй причиной появления подпрограмм стала необходимость сохранять и накапливать уже сделанную работу. Компьютеры работали так медленно, что транслировать программу лишний раз было очень затратно по времени. Придумали технологию сборки, позволяющую без трансляции уже готовых частей собирать новую версию программы. Так появились *библиотеки стандартных подпрограмм.* Ранее мы уже пользовались библиотекой математических функций `<cmath>` и некоторыми другими. Все библиотеки C++, унаследованные от C, являются такими классическими библиотеками стандартных подпрограмм.

Важнейшими вопросами при изучении подпрограмм в любом языке являются следующие: как определить подпрограмму, как вызвать ее для выполнения, какие типы параметров допустимы, каким образом они передаются в подпрограмму и как возвращаются результаты, допускается ли рекурсия. Однако помимо этих традиционных вопросов C++ включает много интересных особенностей, которых нет в других языках: перегрузка функций, шаблоны функций, лямбда-выражения и ряд других.

Объявление и вызов функции

В языке программирования C все стандартные функции были вынесены в библиотеки. Чтобы использовать любую функцию в C/C++, нужно точно знать, в какой библиотеке она находится. Обычно программисты C/C++ пользуются следующими «источниками» функций (не считая объектно-ориентированных библиотек):

- стандартные библиотеки, оставшиеся в наследство от C;
- стандартные библиотеки C++, составляющие STL;
- API — множество функций, предоставляемых операционной системой;
- нестандартные библиотеки, например `boost`, `asio` и т. п.

Однако, чтобы использовать функцию, мало знать, в какой библиотеке она находится. Мы должны знать, как называется функция, какие параметры необходимы ей для правильной работы и какой она возвращает результат. В языке C++ для указания всей этой информации используется *объявление* функции, которое выглядит так:

```
тип имя (список параметров);
```

Этот оператор (в конце поставлена точка с запятой) представляет собой заголовок некоторой функции без тела и называется *прототипом*. Мы уже сталкивались с прототипами стандартных функций из библиотеки `<cmath>`, когда писали программу математических вычислений с числом π (см. листинги 3.5 и 3.6). Нам придется писать прототипы собственных функций, поэтому сразу разберемся, что и как нужно задавать.

Имя — это имя функции, оно является обычным идентификатором C++. Список параметров определяет количество, порядок, типы, способ передачи и, возможно, начальные значения параметров. Параметры в объявлении (и в определении) называются *формальными*. Если их нет, то можно оставить скобки пустыми (как мы поступали в заголовках функции `main` всех наших программ в этой книге).

Тип слева от имени определяет тип возвращаемого значения. Этот элемент важен для понимания, как и в каком контексте может быть вызвана функция. Вызов функции в C++ — это выражение, поэтому по типу возвращаемого значения можно определить, в каких выражениях разрешается использовать вызов этой функции. Вызов имеет вид:

```
имя(список аргументов)
```

Аргументы, задаваемые при вызове функции, называются *фактическими* параметрами, или *аргументами*. Список аргументов должен соответствовать списку формальных параметров, хотя в C++ возможны исключения из этого общего правила.

Функция, возвращающая значение некоторого типа, обычно должна вызываться как один из элементов выражения того же типа, например:

```
double y = sin(x) + 2*cos(x) + 1.53;
```

Если функция не возвращает значения, то вместо типа нужно задавать слово `void`. Функции, не возвращающие значений, аналогичны процедурам в других языках

программирования, и вызов таких функций должен быть отдельным оператором, например:

```
std::swap(x,y);
```

Вообще-то так можно вызывать любую функцию, но для функции, которая возвращает значение, это не имеет смысла — значение просто теряется.

Прототипы всех стандартных функций, которые разрешается использовать в интегрированной среде, можно найти во включаемых файлах, размещенных в каталоге `include` интегрированной среды. Но лучше смотреть прототипы и их описание непосредственно в стандарте C++.

Как написать функцию

Как всякий уважающий себя язык программирования, C++ предоставляет программисту массу возможностей писать свои собственные функции. На примере функции `main` мы уже знаем, что каждая функция состоит из заголовка и тела. Заголовок — это прототип, только в конце точка с запятой не ставится. В заголовке задается имя функции, список параметров и тип возвращаемого значения.

Тело функции — это блок. Тело функции определяет алгоритм работы. Как мы помним (см. программу «Кирпич и отверстие», листинг 5.5), в блоке можно писать не только исполняемые операторы, но и объявлять локальные переменные и константы.

Напишем простую функцию, вычисляющую максимум из трех целых чисел (пример приводится в стандарте C++):

```
int mmax (int a, int b, int c){
   int t = (a > b)? a: b;
   return (t > c)? t: c;
}
```

Заголовок функции понятен: три параметра целого типа, и возвращается значение целого типа. В теле функции определена локальная переменная `t`, которая при объявлении проинициализирована выражением с условной операцией выбора максимума из двух чисел `a` и `b`. Далее написан известный нам оператор возврата, в котором в качестве возвращаемого значения задано выражение окончательного выбора максимума.

Эту же функцию можно написать значительно короче (листинг 7.1), если для ее реализации использовать стандартную функцию вычисления максимума из двух чисел.

Листинг 7.1. Функция выбора максимума из трех целых чисел

```
int mmax (int a, int b, int c){
   return std::max(std::max(a, b), c);
}
```

В теле функции — один оператор возврата, в котором все и выполняется. Вызов функции `mmax` делается аналогично вызовам стандартных функций:

```
int r = mmax(x, 5, z);
```

Ранее (см. *урок 3*) мы написали программу вычисления длины окружности, площади круга и объема шара (см. листинги 3.5 и 3.6). Сейчас мы каждую из формул можем записать в виде функции, которые потом можно будет использовать в других наших математических программах.

Рассмотрим первую — вычисление длины окружности. В английском есть специальный термин circumference — это и будет именем нашей функции. Параметр единственный — радиус окружности типа `double`. Возвращаемое значение тоже `double`. Таким образом, с учетом этого предварительного анализа, заголовок (и прототип) нашей функции будет таким:

```
double circumference(double r)
```

Тело состоит из одного оператора возврата, в котором записана формула, но не забываем о необходимости представить число π, если мы программируем в стандарте C++17 или меньше. Полный код функции показан в листинге 7.2.

Листинг 7.2. Функция вычисления длины окружности

```
double circumference(double r){
   double const pi {acos(-1.0)};                    // число π
   return 2*pi*r;
}
```

Совершенно аналогично пишутся и две другие функции. Вызов этих функций ничем не отличается от вызова любой стандартной математической функции.

Как пример «процедуры» напишем программу вывода таблицы умножения в виде функции. Назовем функцию `tables`. Параметром является множитель. Функция должна проверять параметр (множитель от 1 до 9).

Понятно, что эта функция не вычисляет никакого полезного значения для последующего использования, поэтому можно было бы написать слово `void` в качестве типа возвращаемого значения. Однако проверка множителя может закончиться неудачей, и функция должна как-то сообщить в место вызова, что множитель был задан некорректный.

В таких ситуациях лучше всего возвращать некоторое целое значение, которое покажет, получила ли функция правильный параметр или нет. Пусть возвращаемое значение `0` сигнализирует о получении корректного параметра, а отрицательное значение `-1` — о том, что параметр неверен. Такие числа, возвращаемые функцией, не вычисляющей никакого полезного результата, обычно называются *кодами возврата*. Код функции приведен в листинге 7.3.

Листинг 7.3. Функция «Таблица умножения»

```
int tables (int k)
{ if((0 < k)&&(k < 10))
  {  for (int i = 1; i < 10; ++i)
        cout << k << " * " << i << " = " << k*i <<endl;
     return 0;          //  нормальный возврат
  }
  else return -1;       //  параметр неверен
}
```

В функции написано два оператора `return` в разных ветвях условного оператора. Язык C++ вслед за языком C не накладывает никаких ограничений на количество операторов возврата в функциях.

Принятие решения о возвращении значения `-1` (минус 1) при получении неправильного параметра — важнейший момент в *проектировании* функции. При написании почти любой функции программист всегда должен думать о том, что и как будет делать функция, если полученные данные некорректны. Например, пусть при получении неверного множителя функция возвращает этот множитель. Тогда возникает проблема, что возвращать, если множитель корректный. Ноль вернуть нельзя, т. к. ноль — неправильный множитель. При таком решении сложно придумать, как сообщать о нормальном поведении функции. Таким образом, обработка ошибок должна анализироваться и разрабатываться столь же тщательно, как и алгоритм нормального поведения.

Вызов этой функции осуществляет главная функция, которая выглядит так:

```
int main(){
  int k;
  cout << "Введите множитель от 1 до 9: "; cin >> k;
  int err = tables(k);                  //  получение кода возврата
  if (err)                              //  если ошибка
    cout << "Неправильное значение множителя! " << endl;
  return 0;
}
```

Как видите, функция `main` значительно упростилась. Вызов функции `tables` возвращает код ошибки (ноль или не ноль), и этот код проверяется. Если множитель задан корректный, то функция нормально выполняет свою работу и возвращает `0` (приводится к `false`) — вывод сообщения об ошибке не выполняется. Если же множитель задан неверно, то функция возвращает `-1` (приводится к `true`) и на экране появляется сообщение об ошибке.

Определение функции

Мы с вами написали несколько *определений* функций, особо не вдаваясь в подробности. Теперь поговорим о формальностях.

Каждая функция в программе на C++ должна быть *один раз определена* и может быть объявлена несколько раз по мере необходимости. То есть прототип функции можно писать несколько раз в разных частях программы. По правилам языка C++ функция должна быть определена (или объявлена) до того, как использована (вызвана).

Определение функции, как уже было неоднократно сказано, состоит из заголовка и тела. В общем случае заголовок функции содержит довольно много элементов:

- атрибуты;
- спецификаторы `extern` или `static`;
- модификаторы `inline` и `constexpr`;
- тип возвращаемого значения, возможно, `void`;
- имя функции;
- список объявления параметров, возможно, `void`;
- спецификацию исключений.

Аттрибуты и спецификаторы нам не понадобятся.

Модификаторы тоже можно не задавать. Их отсутствие говорит компилятору о том, что функцию надо компилировать стандартным способом. Модификатор `constexpr` для функций означает, что при некоторых условиях результат работы функции может быть вычислен во время компиляции (с этим мы еще будем разбираться).

Спецификация исключений — это тоже указание компилятору для создания более простого и быстрого кода. Спецификация исключений показывает, может ли функция *сгенерировать исключения* или она не будет этого делать ни при каких обстоятельствах (исключения мы будем рассматривать позже). Если спецификация исключений не указана, то компилятор считает, что функция может сгенерировать исключение, и создает более универсальный код. Спецификация имеет три формата, но мы будем применять самую простую форму — `noexcept`.

Тело функции — это блок, т. е. последовательность объявлений переменных и исполняемых операторов в фигурных скобках. Даже если функция не выполняет никаких действий, в определении тело функции должно быть. В этом случае тело может состоять просто из скобок { }:

```
void f(){ }
```

Эта совершенно корректная запись тела функции говорит нам о том, что для void-функций можно не писать в теле оператор возврата.

Осваиваем функции

Теперь рассмотрим пример и разберем некоторые особенности и ограничения определений и объявлений.

Математические функции с числом π

Выполним рефакторинг программы вычислений с числом π (формулы 3.1–3.3, см. листинг 3.5). По аналогии с показанной выше функцией `circumference` (см. листинг 7.2) напишем еще две функции: вычисление площади круга и вычисление объема шара. Понятно, что параметры и возвращаемые значения у них такие же, как и у функции `circumference`.

> ***На заметку***
>
> Превращение некоторой части кода в функцию — это один из наиболее часто применяемых приемов рефакторинга.

В функции `main` просто вызовем эти функции в операторе вывода на экран. Код первой версии программы приведен в листинге 7.4.

Листинг 7.4. Математические вычисления с числом π (функции)

```
#include <iostream>
#include <cmath>
using namespace std;
double circumference(double r){                    // длина окружности
   double const pi {acos(-1.0)};                   // число пи
   return 2*pi*r;
}
double circlearea(double r){                       // площадь круга
    double const pi {acos(-1.0)};
    return pi*r*r;
}
double ballvolume(double r){                       // объем шара
    double const pi {acos(-1.0)};
    return 4.0/3*pi*r*r*r;
}
int main(){
    cout.precision(15);
    cout << circumference(1.0) << '\n';
    cout << circlearea(1.0) << '\n';
    cout << ballvolume(1.0) << '\n';
    return 0;
}
```

Почти все технические детали в этом коде нам знакомы. Поэтому внимание обратим, в первую очередь, на *организацию* кода. Все определения функций размещены выше функции `main`, где они вызываются в полном соответствии с принципом «сначала определи (объяви), потом используй».

Можно любую функцию или даже все разместить после функции `main`. Но тогда выше нее нужно разместить объявление функции (прототип).

А вот внутри функции `main` никакую из функций разместить нельзя: определения функций не должны быть *вложенными*. Это правило впервые появилось в C и унаследовано C++.

> ***НА ЗАМЕТКУ***
>
> В C++ есть средства, которые позволяют обойти это ограничение (см. *урок 17*).

В отличие от определений, прототипы (объявления функции) могут быть вложенными в другие функции. Главное, чтобы прототип был прописан до вызова. Разместим, например, функцию `ballvolume` после функции `main`, а прототип — внутри функции `main`.

```
int main(){
    double ballvolume(double R);                // объявление функции
    . . .               // какая-то работа
    cout << ballvolume(1.0) << '\n';            // вызов функции
    return 0;
}
// определение функции
double ballvolume(double r){                    // объем шара
    double const pi {acos(-1.0)};
    return 4.0/3*pi*r*r*r;
}
```

Имена параметров в прототипе (имя `R`) и в определении (имя `r`) функции могут не совпадать. Более того, в прототипе можно вообще не указывать имен параметров, достаточно указать только типы. Однако наличие имен улучшает читабельность программы, поэтому рекомендую всегда писать в прототипах имена параметров.

Второе, на что надо обратить внимание, — наличие константы π в каждой функции. Настоящие программисты смотреть спокойно на такое безобразие не могут: надо избавиться от трех констант и оставить одну. Первое и самое очевидное решение — удалить константу из функций и объявить перед функциями один раз:

```
double const pi {acos(-1.0)};           // число пи
double circumference(double r){         // длина окружности
   return 2*pi*r;
}
double circlearea(double r){            // площадь круга
   return pi*r*r;
}
double ballvolume(double r){            // объем шара
   return 4.0/3*pi*r*r*r;
}
```

Решение для данной программы вполне приемлемое. Однако константа стала *глобальной*, а любой программист вам скажет, что глобальные константы — это не очень хорошо. Поэтому мы превратим константу в функцию `pi`. Естественно,

нужно будет проставить во всех формулах вызов этой функции `pi()`. Код новой версии программы показан в листинге 7.5.

Листинг 7.5. Математические вычисления с функцией *pi*

```
#include <iostream>
#include <cmath>
using namespace std;
double pi(){ return acos(-1.0); }    // функция pi
double circumference(double r){      // длина окружности
   return 2*pi()*r;
}
double circlearea(double r){         // площадь круга
   return pi()*r*r;
}
double ballvolume(double r){         // объем шара
   return 4.0/3*pi()*r*r*r;
}
int main(){
    cout.precision(15);
    cout << circumference(1.0) << '\n';
    cout << circlearea(1.0) << '\n';
    cout << ballvolume(1.0) << '\n';
    return 0;
}
```

Запуск этой программы на выполнение показывает, что программа выдает те же самые результаты, что и программы из листингов 3.5 и 7.4.

И, наконец, последнее: наши функции не проверяют параметры. Почему? В данном случае я поступил точно так же, как поступили разработчики математических функций стандартной библиотеки `<cmath>`: ни одна математическая функция не проверяет свой аргумент — это должна делать вызывающая программа. Например, для вычисления квадратного корня из некоторого числа сначала надо удостовериться, не является ли число отрицательным:

```
if(number >= 0) cout << sqrt(number) << '\n'
```

В противном случае программа закончится аварийно.

Таким образом, прежде чем вызывать одну из наших функций, вызывающая программа должна проверить, корректное ли число передается в функцию в качестве радиуса:

```
if(radius > 0) cout << circumference(radius) << '\n';
```

Ведь это значение может быть получено извне (с клавиатуры, из сети, из базы данных) или являться результатом вычислений в другой функции.

Функция-предикат

Превратим в функцию программу про високосность года (см. листинги 5.4 и 6.2). Функцию назовем `leapyear` (високосный год). Параметром является целое число — год, который надо проверить. Очевидно, что возвращать функция должна значение типа `bool`: год либо високосный (`true`), либо нет (`false`).

Функция, возвращающая булевское значение, имеет специальное название: *функция-предикат*.

Соответственно поправим цикл в функции `main` из листинга 6.2. Первая версия программы с функцией `leapyear` представлена в листинге 7.6.

Листинг 7.6. «Високосный год» с функцией-предикатом

```
#include <iostream>
#include <cctype>
using namespace std;
bool leapyear(uint16_t year){
   if (((year % 100)&&(!(year % 4)))||(!(year % 400)))
        return true;
   else return false;
}
int  main(){
    uint16_t year {0};
    cout << "Для завершения работы введите 0.\n";
    cout << "Задайте год: "; cin >> year;
    while(year > 0){
        if (leapyear(year))                    // вызов функции
            cout << "Год високосный!\n";
        else cout << "Год не високосный!\n";
        cout << "Задайте год: "; cin >> year;
    }
    cout << "Заканчиваем работу\n";
    return 0;
}
```

Тип параметра в функции `leapyear` задан как `uint16_t` — это синоним `unsigned short`. Короткое целое вполне подходит, т. к. все года меньше 65 535 (верхняя граница `unsigned short`).

Программа работает точно так же, как и код в листинге 6.2. Можно запустить и задать для проверки все те же четыре года: 1900, 1996, 1997, 2000. Все результаты совпадают с прежними.

Теперь порефакторим код — он в этом нуждается. Внимательно рассмотрим код функции-предиката `leapyear`. Условный оператор в ней — лишний.

Ведь выражение проверки високосности:

```
( ((year % 100)&&(!(year % 4)))||(!(year % 400)) )
```

принимает значение `true`, если год високосный, и `false` в противном случае. Поэтому проверять его не нужно, а надо просто вернуть его как результат. Код функции сильно упрощается (листинг 7.7).

Листинг 7.7. Функция-предикат «Високосный год»

```
bool leapyear(uint16_t year){
   return( ((year % 100)&&(!(year % 4)))||(!(year % 400)) ); }
```

Если мы подставим эту функцию вместо прежней в код программы из листинга 7.6, то все будет работать в точности, как и прежде.

Список параметров и вызов функции

Рассмотрим технические подробности передачи параметров в функцию. В заголовке определения функции формальные параметры перечисляются через запятую. Наиболее часто спецификация отдельного параметра имеет синтаксис, аналогичный синтаксису обычных объявлений переменных:

```
тип имя
```

Тип аргумента может быть любым, который только возможен в C++: либо одним из встроенных простых типов, либо типом, определенным тем или иным способом. Имя — это обычный идентификатор. Если параметров несколько, то все имена должны быть различны. Определение параметров отличается от определения переменных: если при объявлении переменных можно для нескольких имен указать один тип, то при объявлении параметров для каждого имени указывается свой собственный тип.

Примеры функций с несколькими параметрами — функции вычисления периметра и площади прямоугольника. Как обычно, выполним рефакторинг уже написанной программы (см. листинг 3.4) и «инкапсулируем» формулы в функции (листинг 7.8).

Листинг 7.8. Функции периметра и площади прямоугольника

```
double Square(double a, double b){
    return a * b;
}
double Perimeter(double a, double b){
    return 2 * (a + b);
}
```

Функцию `main` модифицируйте самостоятельно. Замечу только, что код главной функции должен значительно сократиться, т. к. переменные `S` и `P` нам теперь не нужны, и вызов функций можно написать непосредственно в операторах вывода.

Обычно говорят, что в C/C++ существует три способа передачи параметров:

- передача по значению;
- передача по указателю;
- передача по ссылке.

На самом деле, первые два способа — это наследие языка C, и в C++ их можно не использовать, поскольку механизм передачи по ссылке позволяет осуществить и тот и другой способ более элегантно и эффективно. Тем не менее передача по значению активно применяется в C++. Заметим, что передача по указателю (с которой мы познакомимся позднее) на самом деле является просто передачей указателя по значению.

Важно отметить, что при вызове функции с несколькими параметрами *порядок вычисления аргументов не определен.* Напомню, что при вычислении выражений порядок вычисления операндов тоже не определен.

Передача параметров по значению

Передача по значению — самый распространенный и надежный метод передачи, работающий и во многих других языках программирования. При этом способе передачи параметров предполагается, что каждый объявленный в заголовке параметр является локальной переменной в теле функции (*см. разд. «Область видимости и время жизни» данного урока*). Поэтому любые его изменения происходят только внутри функциии и не влияют на переменные вне ее.

Передача параметра по значению эквивалентна выполнению операции присваивания `имя_параметра=выражение`. Выражение вычисляется, приводится к типу параметра (если это необходимо и возможно) и присваивается параметру.

В частности, если выражение — константа или переменная, совпадающая по типу с параметром, то значение просто копируется в параметр. Копирование значения фактического параметра в локальную переменную требует времени, поэтому способ передачи параметров по значению обычно применяется для данных встроенных типов, время копирования которых мало.

Во все стандартные математические функции из библиотеки `<cmath>` параметры передаются по значению. Все ранее написанные нами функции использовали именно такой способ передачи параметров.

В качестве примера рассмотрим простую функцию `Cube`, вычисляющую куб своего аргумента. Аргумент в ней как раз передается по значению:

```
float Cube(float x)              //--определение функции
{ return (x*x*x); }
float a{5.5}, b{1.2},
      c{Cube(2)};                // фактический-параметр-константа
float d{Cube(b)};                // фактический-параметр-переменная
float e{Cube(a/b+2.51)};         // фактический-параметр-выражение
float f{Cube(Cube(c))};          // параметр-выражение-вызов функции
```

Все примеры понятны, за исключением последнего. С математической точки зрения этот вызов означает `f(f(x))`. Поскольку `c` равно 8 (вызов `Cube(2)`), то получается следующий результат: `8*8*8` равно `512`, которое при возведении в куб даст значение `134217728`.

Наглядно продемонстрировать, что семантика передачи параметра по значению совпадает с семантикой операции присваивания, позволяет нам постфиксная операция инкремента. Пусть в программе объявлена некоторая переменная `b`, имеющая значение `1`. Тогда операторы:

```
cout << Cube(b++) <<endl;
cout << b << endl;
```

выведут на экран:

```
1
2
```

Несмотря на то, что параметр передается по значению, сначала в функцию передается значение единица, а вычисление выражения `b++` выполняется после передачи значения параметра в функцию — это в точности совпадает с выполнением операции присваивания:

```
x=b++;
```

Как правило, список фактических параметров должен соответствовать списку формальных параметров по количеству и типам. Однако в C++ возможны исключения из этого правила при определении *параметров по умолчанию*, которые будут рассмотрены далее.

Передача параметров по ссылке

Передача параметра по значению — надежнейший способ, который, как мы теперь понимаем, практически всегда применяется для математических функций. Эти функции выдают единственный результат. Однако очень часто требуется получить от подпрограммы более одного результата, например изменить значения *нескольких* переменных. Передача по значению в таких случаях не годится. Чтобы понять, в чем суть дела, рассмотрим классический пример: функцию, обменивающую значения двух целых переменных.

```
void Swap(int a, int b)
{ int t = a; a = b; b = t; }
```

Тело функции нам уже хорошо знакомо: именно такую последовательность операторов присваивания мы писали в задаче про кирпич и отверстие (см. листинг 5.5).

Такая функция не вызовет протестов компилятора, но работать не будет, т. к. параметры передаются по значению. Пусть в программе объявлены две переменные `x` и `y`, и нам необходимо обменять их значения местами.

```
int x = 5, y = 3;
Swap (x, y);
cout << "x = " << x << "y = " << y << endl;
```

Оператор cout выведет на экран x = 5 и y = 3. Таким образом, функция Swap не сделала свою работу. Вставим в функцию оператор вывода на экран.

```
void Swap (int a, int b)
{  int t = a; a = b; b = t;
   cout << "a = " << a << " b = " << b << endl;
}
```

Повторив выполнение, увидим на экране следующее:

```
a = 3 b = 5
x = 5 y = 3
```

Таким образом, внутри функции обмен происходит. Тем не менее на значения переменных x и y это никак не влияет. Это происходит именно вследствие передачи параметров по значению: внутрь функции передаются только значения переменных x и y во внутренние (локальные) переменные a и b. Обмен происходит в этих внутренних переменных функции Swap.

Именно в таких случаях и необходим способ передачи параметров по ссылке. Код определения функции Swap приведен в листинге 7.9.

Листинг 7.9. Функция обмена значений переменных

```
void Swap(int &a, int &b)          // - ссылки в заголовке
{ int t = a; a = b; b = t; }
```

Обращение к таким образом определенной функции не отличается от обращения при передаче параметров по значению:

```
Swap(x,y);
```

Выполнив приведенный выше пример, убеждаемся, что теперь все в порядке — переменные вне функции поменяли свои значения.

По стандарту языка C++ при передаче параметра по ссылке на месте фактического параметра можно указывать только переменную.

Константные параметры функций

Так же как и переменные, параметры могут быть объявлены константными. Рассмотрим несколько вариантов заголовков одной и той же функции f.

```
int f(int x);                  //--по значению
int f(int& x);                 //--по ссылке
int f(const int x);            //--константный по значению
int f(const int& x);           //--константный по ссылке
```

Первый вариант передачи параметра позволяет нам в качестве фактического параметра задавать и переменные, и любые выражения. В теле функции мы можем использовать этот параметр как дополнительную локальную переменную. Мы не только получаем значение фактического параметра, но и можем присваивать новые значения этой переменной. Как мы знаем, никаких последствий для фактического параметра это не имеет.

Во втором случае мы имеем передачу параметра по ссылке и можем в качестве фактического параметра задавать только переменную. Эта переменная похожа на глобальную — все изменения параметра внутри функции немедленно отражаются на содержимом фактического параметра.

Третий вариант — передача по значению, но внутри тела функции запрещено изменять значение параметра: компилятор не будет транслировать программу при наличии изменяющих операторов в теле такой функции (нельзя вводить и присваивать).

А вот четвертый вариант более интересен: мы передаем параметр по ссылке, но запрещаем его изменение внутри тела функции, как и в предыдущем случае. При вызове такой функции разрешается задавать в качестве аргумента как переменную, так и произвольное выражение, чего нельзя было сделать при передаче параметра просто по ссылке (без слова `const`). Таким образом, этот способ аналогичен передаче по значению, но имеет два существенных преимущества по сравнению со старым способом:

- он значительно эффективнее — не происходит копирования значения;
- он явно самодокументирован.

Константность/неконстантность, как мы увидим далее, приобретает важное значение для параметров-массивов и параметров-контейнеров стандартной библиотеки.

Область видимости и время жизни

Мы уже неоднократно упоминали о локальных переменных. Разберемся в этом вопросе более детально.

Любое имя в программе так или иначе объявляется. С момента объявления имя может использоваться в программе. *Область видимости* — часть исходного кода программы, в которой имя можно использовать в конструкциях языка. Иногда вместо термина «область видимости» употребляется термин «область действия». С моей точки зрения эти два термина эквивалентны, поэтому в дальнейшем будем пользоваться словосочетанием «область видимости».

В C++ есть различные варианты области видимости имен. Мы на данный момент имели дело только со следующими их видами:

- глобальная: область видимости, не входящая ни в одну другую область видимости;
- пространство имен: область видимости, входящая в глобальную область видимости или в другое пространство имен;
- локальная область видимости: между фигурными скобками блока `{ ... }` или в списке параметров функции;
- область видимости оператора: например, в цикле `for`.

С глобальной областью видимости мы уже немного знакомы: вспомним объявление глобальной константы `pi` вне всех функций (см. листинг 7.6). Пространство имен мы пока не объявляли (сделаем это позднее), но уже использовали: в каждой

программе написана директива `using`. Даже область видимости оператора нам уже знакома (см. листинг 6.9).

Теперь мы детально рассмотрим традиционные правила локальной области видимости, связанные с понятием функции. С областью видимости связано другое понятие — время жизни объекта. В данном случае речь идет исключительно о переменных и константах. *Время жизни* — это время существования объектов во время работы программы.

Время жизни глобальных объектов совпадает с временем работы программы. Имя глобальной переменной видно с момента объявления, т. е. в любой точке программы *после* точки объявления. Время жизни объектов в области видимости оператора определяется оператором. Например, счетчик цикла, объявленный в операторе `for`, «живет» до конца выполнения цикла.

Все переменные, объявленные в теле функции, как и параметры, являются внутренними *локальными переменными* функции. Это одно из проявлений принципа *инкапсуляции*. В данном случае это означает следующее:

1. За пределами функции имена объявленных внутри нее переменных не видны, т. е. имя переменной *нельзя использовать вне тела* функции в других местах программы. Говорят, что область видимости локальных переменных и параметров — тело функции.
2. До входа в функцию локальных переменных не существует. Они «рождаются» при входе в функцию и «умирают» при выходе из нее. Говорят, что время жизни локальных переменных ограничено временем выполнения функции. То же относится к параметрам. Такая простая дисциплина реализуется с помощью стека[1]: локальные переменные и параметры автоматически размещаются в стеке программы при входе в функцию и удаляются из стека при выходе из нее.

В связи с этим немедленно возникают три вопроса:

- можно ли внутри функции объявить переменную таким образом, чтобы она была видна вне тела функции?
- что происходит, если имя локальной переменной совпадает с именем глобальной (объявленной вне функции)?
- можно ли внутри функции объявить переменную таким образом, чтобы время ее жизни не было связано с временем выполнения функции?

Проще всего ответить на первый вопрос: внутри тела функции нельзя объявить переменную так, чтобы она была видна вне тела функции.

Сокрытие имен

Ответ на второй вопрос таков: внутренняя переменная скрывает внешнюю переменную с тем же именем внутри функции. То есть вне функции работает внешняя

[1] Стек — это динамическая структура данных, работающая по принципу Last In First Out (LIFO) — «последним пришел, первым ушел».

переменная, а при входе в функцию используется внутренняя локальная переменная с тем же именем. При выходе из функции снова работает внешняя переменная. Это правило существует практически во всех языках программирования.

Для прояснения этого вопроса напишем программу, код которой представлен в листинге 7.10.

Листинг 7.10. Сокрытие глобальной переменной

```
#include <iostream>
using namespace std;
int x = 5;                          // глобальная переменная
void f()
{ int x = 3;                        // локальная переменная
  cout << "x = "<< x <<"-внутренний" <<endl;
}
int main(){
    cout << "x = " << x <<endl;
    f();
    cout << "x = " << x <<endl;
  return 0;
}
```

При выполнении этой программы на экран выведется:

```
x = 5
x = 3-внутренний
x = 5
```

Это означает, что до вызова функции видна глобальная переменная `x`. При входе в функцию локальная переменная `x`, объявленная внутри функции `f()`, скрывает глобальную переменную — внутри функции глобальная `x` не видна. При выходе из функции локальная переменная `x` уничтожается, и опять становится видна глобальная `x`.

А если все-таки внутри функции требуется обращаться и к локальной, и к глобальной переменной, имена которых одинаковы?

> ***Совет***
>
> Никогда не называйте локальные и глобальные объекты одинаковым именем.

Тогда для доступа к глобальной переменной нужно применить операцию глобального доступа `::`. Эта операция нам уже знакома: мы не раз писали в программах имена с префиксом `std::` — тут как раз и используется эта операция. Как всякая операция в C++, она имеет приоритет — самый наивысший среди всех операций. Поэтому выполняется всегда первой.

Следующий фрагмент программы поясняет, как это делается.

```
int x = 5;                          // глобальная переменная
void f(){
```

```
   int x =3;                    // локальная переменная
   cout << "x = " << x <<"-внутренний" <<endl;
   cout << "x = " <<::x<<"-глобальный" <<endl;
}
int main(){
   cout << "x = " << x<<endl;
   f();
   cout << "x = " << x<<endl;
}
```

При выполнении этого фрагмента программы на экран выведется:

```
x = 5
x = 3-внутренний
x = 5-глобальный
x = 5
```

Интересно, что принципу блочной видимости подчиняются и прототипы функций: прототип, объявленный внутри функции, вне функции не работает — он невидим. Более того, как и в случае переменных, если мы имеем два разных прототипа с одинаковым именем, один из которых вне функции, а второй внутри, то внутренний скрывает внешний. C++ разрешает иметь функциям одинаковые имена, обеспечивая *перегрузку* функций.

Статические переменные

Ответ на третий вопрос также положительный: внутри функции можно объявить локальную переменную, время жизни которой не зависит от времени выполнения функции. Для этого переменную надо объявить *статической*. Общий вид объявления статической переменной следующий:

```
static тип имя
```

Статическую переменную, как и любую другую, можно инициализировать при объявлении. Замечательное свойство таких локальных переменных состоит в том, что они сохраняют свое значение после окончания работы функции. При очередном вызове используется оставшееся после предыдущей работы значение. В то же время локальные статические переменные защищены от изменений извне, поскольку имя переменной вне функции недоступно.

Одним из традиционных применений статических переменных является использование их в качестве счетчиков для сбора различной статистической информации о вызове функции. Например, можно посчитать количество вызовов функции. Это делается так:

```
unsigned int f(параметры)
{ static unsigned int count = 0;
// . . .
  ++count;
  return count;
}
```

Вызов делается так:

```
insigned int n = f(параметры);
```

Эта информация может понадобиться для оптимизации программы. Подобные счетчики часто реализуются в объектно-ориентированном программировании, когда нужно получать статистическую информацию о создании, поведении и уничтожении объектов в программе.

Рассмотрим применение статических переменных для вычисления среднего арифметического последовательности значений, задаваемых пользователем с клавиатуры. Мы уже писали такую программу без использования функций (см. листинги 6.8 и 6.9).

В листинге 7.11 показана функция вычисления среднего с использованием статических переменных.

Листинг 7.11. Статические переменные в функции

```
#include <iostream>
using namespace std;
double Average(double data)     // функция вычисления среднего
{ static double summa = 0;      // инициализация статических
  static int count = 0;         // переменных при первом вызове
  count++; summa += data;
  return summa/count;           // возврат нового среднего значения
}
int main(){
  double data {1}, middle;
  while(data)                           // пока не задали ноль
  { cout << "Введите число: "; cin >> data;
    middle = Average(data);             // вычисление очередного среднего
    cout << "Среднее значение: " << middle << endl;
  }
  return 0;
}
```

При первом вызове сначала статические переменные `summa` и `count` получают значение `0`. Затем к значению переменной `summa` добавляется значение параметра `data`, а переменная `count` увеличивается на 1. Выполняется возврат среднего (при первом вызове это значение переданного параметра). При втором вызове (и следующих) инициализация переменных не выполняется, а используются значения, накопленные в предыдущем вызове.

Программа заканчивает работу при вводе пользователем нуля. Таким образом, функция вычисляет среднее арифметическое последовательности чисел, среди которых нет нуля.

Автоматизация тестирования функций

Тестирование — обязательная часть работы программиста. Мы, как самые настоящие профессионалы, составляли набор *тест-кейсов* для проверки наших программ. Каждый тест-кейс обязательно включает входные данные и требуемый результат (вспоминаем табл. 3.1 и 6.1). Результат — это главное в тест-кейсе, т. к. именно различие требуемого и полученного результата показывает программисту, что в программе что-то пошло не так.

До сих пор мы проводили тестирование визуально, сравнивая то, что вывела программа, и требуемый результат в тест-кейсе. Подобное тестирование обычно называется ручным тестированием. Мы тестировали программы вручную только потому, что они пока очень небольшие, и данных вводить требуется немного.

Путь программиста — это путь избавления от ручной работы. Поэтому нам нужно подумать об автоматизации тестирования разрабатываемых программ. Наличие в программе функций открывает нам возможность сделать это.

Чтобы автоматизировать тестирование, конечно, надо написать код. Очевидно, что надо написать условные операторы, которые сравнивают результаты вызова функций и эталонные результаты, прописанные в тест-кейсах. Напишем тестирующий код для проверки функции `leapyear` (см. листинги 7.6 и 7.7). Проще всего это сделать в функции `main`. Удалим из нее все операторы и напишем следующий код, показанный в листинге 7.12.

Листинг 7.12. Тестовый код для проверки функции *leapyear*

```
#include <iostream>
using namespace std;
bool leapyear(uint16_t year){
   return(((year % 100)&&(!(year % 4)))||(!(year % 400)));
}
int main(){
    if(leapyear(2000))  cout << "leapyear(2000) прошел\n";
    if(leapyear(1996))  cout << "leapyear(1996) прошел\n";
    if(!leapyear(1900)) cout << "!leapyear(1900) прошел\n";
    if(!leapyear(1997)) cout << "!leapyear(1997) прошел\n";
    return 0;
}
```

Код понятен. Вызов функции `leapyear` непосредственно в условии оператора `if` — это как раз то, что и требуется. Один условный оператор — один тест. Так как сама функция возвращает логический результат, то сравнивать результат вызова с константами `true` или `false` не нужно. В первых двух операторах функция возвращает `true`, т. е. год 2000 и год 1996 — високосные. В консоль выводятся сообщения о том, что тест прошел. Следующие два вызова возвращают `false`, т. е. год 1900 и год 1997 — не високосные. Логическая операция отрицания конвертирует `false` в `true`, поэтому сообщения о прохождении тестов тоже выводятся.

Таким образом, при выполнении программы мы получим в консоли четыре сообщения о том, что код прошел:

```
leapyear(2000) прошел
leapyear(1996) прошел
!leapyear(1900) прошел
!leapyear(1997) прошел
```

Аналогичным способом, написав в функции `main` соответствующий код, можно проверять любые функции.

Стандартные средства тестирования

Однако гораздо удобнее выводить сообщения о не прошедших тестах — это более экономичный подход, т. к. по мере выявления и исправления ошибок количество сообщений будет уменьшаться. В конечном итоге сообщений о не прошедших тестах не должно быть ни одного.

Но переписывать код программы из листинга 7.12 мы не будем. Процесс тестирования уже давно во многом автоматизирован. Поэтому сейчас мы воспользуемся простейшим стандартным средством — макросом `assert`. Для его использования нужно подключить стандартную библиотеку `<cassert>`. Как видим, библиотека была написана еще при создании языка C.

Сначала напишем программу, а потом разберем все непонятные моменты. Протестируем функцию вычисления периметра прямоугольника (см. листинг 7.8). Воспользуемся данными из табл. 3.1 и напишем в функции `main` вызовы макроса `assert`. Код программы приведен в листинге 7.13.

Листинг 7.13. Тестирование функции вычисления периметра прямоугольника

```
#include <iostream>
#include <cassert>
using namespace std;
double Perimeter(double a, double b){
    return 2 * (a + b);
}
int main(){
    assert(Perimeter(1,1) == 4);                // 1
    assert(Perimeter(1,2) == 6);                // 2
    assert(Perimeter(2,1) == 6);                // 3
    assert(Perimeter(2,2) == 8);                // 4
    assert(Perimeter(1.1,1.1) == 4.4);          // 5
    assert(Perimeter(0.1,1) == 2.2);            // 6
    assert(Perimeter(1.1,0.9) == 4);            // 7
    cout << "Все тесты прошли!\n";
   return 0;
}
```

Рассмотрим первый оператор в теле функции `main`. Этот оператор *похож* на вызов функции. Имя `assert` — это имя макроса, а не функции.

В качестве аргумента `assert` задано логическое выражение сравнения: результат вызова функции `Perimeter` сравнивается с результатом тест-кейса из табл. 3.1, который соответствует заданным аргументам функции `Perimeter`. Макрос `assert` проверяет истинность этого утверждения. Если утверждение истинно (`true`), то выполнение программы продолжается и выполняется следующий `assert`. Если во всех операторах `assert` значение логического выражения равно `true`, в конце работы выводится сообщение, что все тесты прошли.

Если же мы сделаем ошибку, то увидим картину, показанную на рис. 7.1. Я в операторе `// 6` проставил неверное значение периметра.

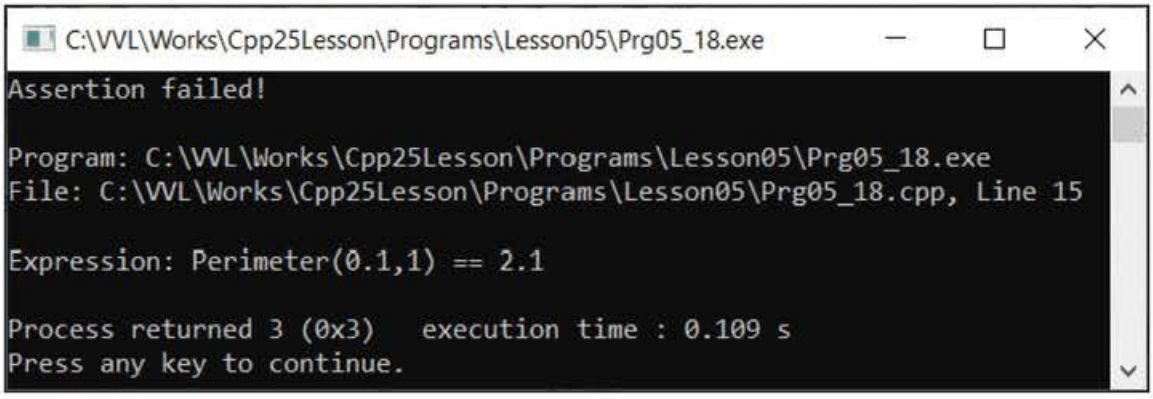


Рис. 7.1. Ошибка при вызове макроса `assert`

Сообщения, естественно, на английском, как и все, что реализовано в стандартной библиотеке. Первое сообщение, `Assertion failed!`, говорит о том, что тест не прошел. Следующая строка сообщает, в какой программе (`Program`) это произошло.

Но самые интересные сообщения — следующие два. Сначала сообщается номер строки в файле (`File`) исходного кода, где не сработал этот `assert`. И последнее сообщение показывает то логическое выражение (`Expression`), которое в этом операторе `assert` оказалось ложным (`false`). Согласитесь, это очень полезная информация при тестировании программы.

Этот пример показывает, что при написании тестового кода программист тоже может совершать ошибки. Поэтому тестирование должно выполняться предельно внимательно, и каждый не прошедший тест нужно анализировать с двух позиций: смотреть и тестовый код, и рабочий код.

Вроде бы количество работы удвоилось. Но написание тестового кода очень ускоряет проверку рабочего кода и исправление в нем совершённых ошибок.

Задачи и упражнения

1. Написать функцию, возвращающую количество цифр натурального числа.
   - Используя эту функцию, приписать к заданному числу N ($1 \le N \le 20000$) это же число.

Например:

```
1 -> 11
12 -> 1212
123 -> 123123
```

- Используя эту функцию, обменять в заданном числе N первую и последнюю цифры ($12 \le N \le 1000000000$):

```
12 -> 21
123 -> 321
1234 -> 4231
```

2. Написать функцию-предикат, проверяющую, является ли натуральное число простым. Найти все простые числа на интервале от A до B.

   Подсказка: число является простым, если делится только на себя и на 1. Число 1 простым не является, но оно не является и составным. Поэтому для проверки на простоту следует пытаться делить число на 2, затем на 3 и т. д. Если число поделилось на очередной делитель, то оно не является простым.

3. Написать функцию, возвращающую наибольший общий делитель (НОД) двух натуральных чисел.

   Подсказка: реализовать классический алгоритм Евклида: пока числа A и B не равны, из большего вычитается меньшее. Когда числа станут равны — это и есть наибольший общий делитель.

4. Даны два натуральных числа A и B, представляющие собой числитель и знаменатель некоторой дроби. Написать функцию для сокращения дроби, разделив числитель и знаменатель на их наибольший общий делитель.

   Подсказка: использовать алгоритм Евклида (задача 3), параметры (числитель и знаменатель) в функцию передать по ссылке.

5. Реализовать задачи предыдущего урока в виде функций. Самостоятельно определить, как передавать параметры и какие результаты возвращать.

6. Для всех задач *уроков 5*, *6* и *7* реализовать тестирующие функции.

- Урок 8 -

С пользой для дела

Предыдущие уроки содержали учебный материал. Хотя мы писали довольно много программ, эти программы были учебные. Они создавались для изучения той или иной конструкции C++. Но в настоящий момент мы изучили уже достаточно, чтобы начать писать реально полезные программы.

Ранее мы написали уже несколько вариантов программы, которая показывает нам таблицу умножения (см. листинги 2.9, 3.3, 5.2, 6.3, 6.4, 6.5, 6.6 и 7.3). Все они полезны, т. к. с помощью таких программ таблицу умножения можно повторить, если что-то забыл. Но лучше написать программу, которая будет не только показывать таблицу, но и тренировать школьника, чтобы он ее выучил. Этим мы и займемся.

Тренажер по таблице умножения

Основная идея состоит в том, чтобы программа задавала ученику вопросы, например: сколько будет 2*2? Ученик вводит ответ. Программа задает следующий вопрос: сколько будет 6*4? Ученик опять вводит ответ. И т. д. Такие программы называются *тренажерами*.

Для разработки программы нужно решить несколько проблем. Самая простая из них: в каком виде задавать вопрос. Можно так, как написано выше. Но ведь каждый вопрос — это задание, которое нужно решить. А задание можно задавать кратко:

```
3*4 = .
```

После знака равенства ученик вводит ответ. Решение о формате задания принимает разработчик тренажера.

Проблема посложнее состоит в том, сколько раз задавать задания и когда заканчивать тренировку. Самый простой вариант — фиксированное количество заданий. Но для тренажера это плохое решение. Гораздо лучшим вариантом является произвольное количество заданий. Пользователь сам завершит тренировку, вводя особый ответ, например ноль.

Однако в тренажерах существует самая главная проблема: задания должны быть разные при каждом запуске программы. Даже если задания повторяются (например, на первом уровне 10 заданий по таблице умножения на 2), нужно выдавать их при каждом запуске в произвольном порядке. Почему это так важно? Практика

обучения показывает, что ученик может запомнить порядок следования ответов, а не сами ответы. Ситуация примерно такая же, как при изучении иностранных слов: нужно учить их не последовательно одно за другим, а в произвольном порядке.

В решении этой проблемы нам помогут *случайные* числа. Случайные числа очень полезны при тестировании программ. Кроме того, фактически все компьютерные игры используют случайные числа. Поэтому нужно научиться их применять.

Случайные числа

Средства для работы со случайными числами определены в библиотеке `<random>`. Библиотека предоставляет несколько разных *датчиков* случайных чисел. Датчик — это такой объект, который можно «попросить» в программе: дай мне случайное число. Эту «просьбу» можно повторять в цикле, и датчик будет выдавать каждый раз новое (иногда повторяющееся с каким-нибудь предыдущим) число.

Конечно, средство, с помощью которого нужно «просить», реализовано в самом датчике. Можно сказать, что случайное число *генерируется*, и датчик можно назвать *генератором* случайных чисел (ГСЧ). Но программисты с давних времен привыкли к названию «датчик».

Чтобы использовать датчик случайных чисел, как обычно, нужно объявить в программе переменную соответствующего типа и инициализировать ее. Стандартный датчик, обычно применяемый по умолчанию, объявляется так:

```
default_random_engine dre;
```

Имя `dre` выбрано специально, чтобы не забыть, что это датчик по умолчанию. Но лучше определить синоним этого длинного имени:

```
using random_t = default_random_engine;
```

После этого можно датчик по умолчанию определять короче:

```
random_t dre;
```

В стандартной библиотеке реализовано еще несколько других датчиков, но мы будем пользоваться либо стандартным, либо специальным 64-битным, тип которого — `mt19937_64`.

Инициализация датчика случайных чисел

Датчик нужно инициализировать, и это делается разными способами:

- при объявлении;
- с помощью метода `seed()`.

Инициализация датчика случайных чисел по смыслу совершенно отличается от инициализации обычной переменной. Датчик используется, чтобы получить не одно случайное число, а *последовательность* чисел. Или, как говорят, *случайную последовательность*. Если датчик не инициализирован или инициализирован неправильно, то последовательность всегда получается одна и та же.

Например, в тренажере по таблице умножения при каждом запуске требуется выдать новую последовательность заданий. Каждое задание создается из случайных чисел. Если последовательность каждый раз будет одна и та же, ученик ее запомнит. И тренажер перестанет выполнять свою функцию: тренировать запоминание таблицы умножения.

Чтобы убедиться, что без инициализации получается одна и та же случайная последовательность, напишем простую программу, в которой заодно покажем, как получить очередное случайное число. Код — в листинге 8.1.

Листинг 8.1. Последовательность случайных чисел без инициализации

```
#include <iostream>
#include <random>
using namespace std;
using random_t = default_random_engine;
int main(){
   random_t dre;                          // датчик по умолчанию
   cout << dre() <<'\n' << dre() <<'\n' << dre() <<'\n';
   mt19937_64 gen;                        // 64-битный датчик
   cout << gen() <<'\n' << gen() <<'\n' << gen() <<'\n';
   return 0;
}
```

Как видите, для получения очередного случайного числа надо написать имя переменной-датчика с круглыми скобками после него: `gen()`. Это не вызов функции, а создание временного объекта данного типа.

> ***На заметку***
>
> С подобной записью мы столкнемся еще не раз. Такая запись означает вызов конструктора по умолчанию. Мы познакомимся поглубже с этой темой при изучении основ объектно-ориентированного программирования (см. *урок 10*).

Запустив программу дважды, увидим в консольном окне одни и те же числа, как показано на рис. 8.1.

Первые три числа генерирует датчик случайных чисел по умолчанию, и эти числа лежат в диапазоне `unsigned int`. Следующие три числа генерируются датчиком, который называется «вихрь Мерсенна», и он генерирует 64-битные числа `unsigned long long`. И сколько бы раз мы ни запускали программу, мы всегда получим одни и те же числа.

Для получения действительно случайной последовательности чисел нужно инициализировать датчик *случайным* начальным значением. Современный вариант — использование объекта типа `random_device`. Этот тип реализован в библиотеке `<random>` специально для инициализации датчиков случайных чисел. Надо объявить переменную этого типа, а потом использовать ее для инициализации датчика:

```
random_device rd;
random_t dre { rd() };          // инициализация при объявлении
```

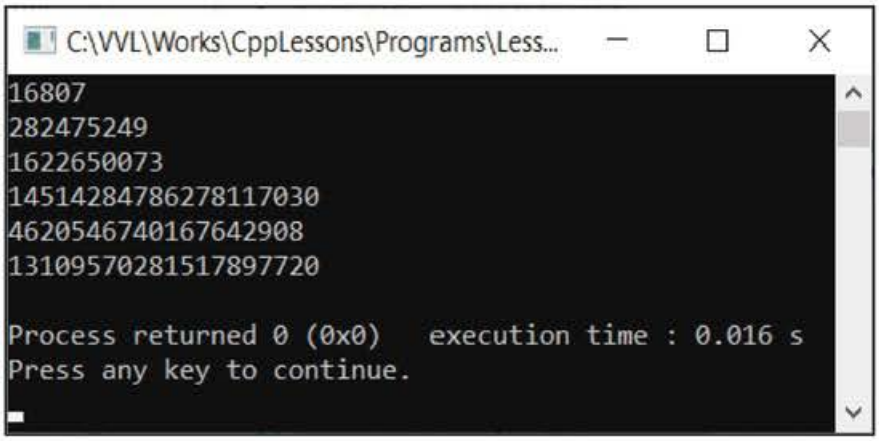


Рис. 8.1. Последовательности случайных чисел без инициализации

Как мы видим, и в данном случае инициализирующее значение создано как временный объект типа `random_device`.

Инициализация методом `seed()`:

```
auto init = rd();
gen.seed(init);
```

Инициализация датчика объектом типа `random_device` всегда корректно работает в Linux: при каждом запуске программы объект выдает новое значение, и случайная последовательность каждый раз разная. Однако в Windows надо проверять, работает ли этот механизм. Если первое значение датчика случайных чисел одно и то же при нескольких запусках программы, значит, этот механизм не работает.

Модифицируем код программы в листинге 8.1, задав инициализацию датчиков с помощью объекта `random_device`. Код представлен в листинге 8.2.

Листинг 8.2. Инициализация датчика случайных чисел (*random_device*)

```
#include <iostream>
#include <random>
using namespace std;
using random_t = default_random_engine;
int main(){
   random_device rd;                          // случайный объект
   random_t dre {rd()};                       // инициализация
   cout << dre() <<'\n' << dre() <<'\n' << dre() <<'\n';
   mt19937_64 gen {rd()};                     // инициализация
   cout << gen() <<'\n' << gen() <<'\n' << gen() <<'\n';
   return 0;
}
```

Запустив ее несколько раз в Windows, я получил одни и те же числа:

```
262406142
1476101303
1103509377
```

```
6819385613748767594
15006646477418450595
12585391993046132881
```

Как видите, эти числа отличаются от показанных на рис. 8.1. Датчик все же был инициализирован при каждом запуске, но не случайным объектом, а одним и тем же значением. Поэтому при работе в Windows нужно инициализировать датчик другим способом.

Традиционно это всегда делалось с помощью времени. При каждом запуске программы время будет новое, поэтому если мы каким-то способом получим текущее время, то его можно использовать для инициализации датчика.

Современный способ инициализации использует стандартную библиотеку для работы со временем <chrono>. Нужно сначала получить с помощью системных часов текущее время, а потом привести его к типу time_t. Переменные этого типа хранят количество секунд, прошедшее с 1 января 1970 года. Количество секунд — это обычное целое число, поэтому с данными типа time_t можно выполнять арифметические действия. Этот тип реализован в си-библиотеке <ctime>.

Давайте посмотрим, как это делается, непосредственно в коде (листинг 8.3).

Листинг 8.3. Инициализация датчика временем

```
#include <iostream>
#include <random>
#include <chrono>                                   // библиотека времени
#include <ctime>                                    // для time_t
using namespace std;
using random_t = default_random_engine;
int main(){
   auto tp = chrono::system_clock::now();           // текущий момент
   time_t moment = chrono::system_clock::to_time_t(tp);
   random_t dre { moment };                         // инициализация
   cout << dre() <<'\n' << dre() <<'\n' << dre() <<'\n';
   mt19937_64 gen { moment };                       // инициализация
   cout << gen() <<'\n' << gen() <<'\n' << gen() <<'\n';
   return 0;
}
```

Все средства библиотеки <chrono> помещены в пространство имен chrono, поэтому для сокращения можно задавать директиву using. Но я не стал этого делать, чтобы в коде были явно видны средства работы со временем.

Для получения текущего значения времени используются *часы*. В библиотеке определены три вида часов:

- system_clock — системные часы;
- steady_clock — монотонные часы;
- high_resolution_clock — часы высокого разрешения.

Для работы с системным временем используются системные часы, т. к. только для них реализован метод преобразования `to_time_t`, без применения которого системное время нельзя использовать для инициализации. Напишем функцию, которая будет выдавать текущее время типа `time_t` (листинг 8.4).

Листинг 8.4. Функция *time* для инициализации ГСЧ

```
time_t time(){
    auto tp = chrono::system_clock::now();
    time_t moment = chrono::system_clock::to_time_t(tp);
    return moment;
}
```

Я назвал функцию тем же именем `time` специально. В библиотеке `<ctime>` реализована функция `time`, но она имеет параметр-указатель, с которыми мы еще не работали. Здесь я воспользовался *перегрузкой* функций.

Инициализация датчика с помощью нашей функции делается так:

```
random_t dre { time() };
```

После этого датчик будет выдавать случайную последовательность.

Получение случайных чисел

После инициализации генератор может быть использован для получения целых случайных чисел, например:

```
unsigned long long k = gen();
```

Значение `k` зависит от типа датчика. Каждый датчик генерирует случайные числа в некотором диапазоне от минимального до максимального. Например, стандартный датчик выдает значения от 1 до 2 147 483 646 включительно. А вот минимум 64-битного датчика типа `mt19937_64` равен нулю. Максимальное значение равно `numeric_limits<uint64_t>::max()` (см. табл. 4.1). Любой датчик позволяет получить минимум и максимум своих случайных чисел:

```
cout << gen.min() << "-" << gen.max() << endl;
```

Но в программах обычно нужны числа гораздо меньшего диапазона. Нам в тренажере по таблице умножения нужны числа от 1 до 9. Это легко сделать:

```
unsigned long long k = gen()% 9 + 1;
```

Остаток от деления на 9 даст число от 0 до 8, к которому мы добавляем 1.

Минимум и максимум позволяют получать вещественные случайные числа. Для этого необходимо выполнить вычисления. Пусть нам требуется получать случайные числа из диапазона [0.0, 1.0]. Это можно сделать так:

```
double number = (double(gen())- gen.min())/(gen.max()- gen.min());
```

Если `gen()` равен `gen.min()`, то получится 0.0. А если `gen()` равен `gen.max`(), то результат равен 1.0. При всех других значениях `gen()` переменная `number` получает значение x больше нуля и меньше единицы: $0.0 \le x \le 1.0$.

Стандартная библиотека предоставляет более удобный механизм получения случайных чисел в заданном диапазоне — *распределения*. В библиотеке реализовано много разных распределений, но нас интересуют равномерные распределения целых и действительных чисел.

Для получения равномерно распределенных целых случайных чисел из заданного диапазона необходимо объявить распределение и связать с ним датчик, например:

```
std::uniform_int_distribution<> uid(0,n);
int k = uid(gen);
```

В первой строке объявлено равномерное распределение целых в диапазоне от 0 до *n* включительно. Во второй строке — получение случайного числа из диапазона с помощью ранее объявленного и инициализированного датчика `gen`. Комбинация `uid(gen)` используется везде (почти), где в программе требуется получить очередное случайное число из диапазона [0, n].

Естественно, что можно объявить столько распределений, сколько требуется, и связать с ними один или несколько датчиков. В качестве границ диапазона можно задавать переменные или выражения — границы распределения можно вычислять во время выполнения.

Равномерное распределение вещественных чисел на интервале [0.0, 1.0] задается так:

```
std::uniform_real_distribution<> urd(0,1);
```

Указанные целые границы преобразуются по умолчанию в вещественные. После этого можно получать вещественные случайные числа очень просто:

```
double R = urd(gen);
```

Чтобы не писать длинные названия типов, надо применить using-объявление.

Разработка программы

Итак, мы знаем, как получить последовательность случайных чисел. Теперь надо научиться их использовать в программах. Разработаем сначала самый простой вариант тренажера по таблице умножения.

Перед началом тренировки необходимо вывести приветствие и небольшую инструкцию о том, как будет проходить тренировка, в каком виде будут выдаваться задания и как ее завершить. Это можно реализовать в виде одной или двух функций. Например, функция приветствия спросит имя ученика и поприветствует его по имени (преобразуйте в функцию код листинга 3.2).

А инструкцию должна выводить отдельная функция. Инструкция должна содержать краткое описание процесса тренировки, что и в каком формате должен вводить ученик, как завершить тренировку.

> ***Совет***
>
> Каждая функция, реализованная в программе, должна выполнять отдельную задачу, не связанную с другими частями программы.

Замечу, что программа пишется медленно и постепенно. Начинать нужно с реализации самых простых функций (например, вывод приветствия). Причем тело функции сначала пустое. В главной функции пишется вызов функции. Такой пустой проект должен компилироваться без ошибок и выполняться.

Затем в пустую функцию прописываются необходимые операторы.

И снова не все сразу, а постепенно, по одному-два оператора. При каждом изменении кода программу нужно компилировать и запускать, чтобы убедиться в ее работоспособности.

Если вы пока не знаете, какие операторы писать в коде, напишите вместо этого комментарий — это поможет вам уяснить, что же требуется реализовать в данном месте программы.

Первая версия программы может выглядеть примерно так, как в листинге 8.5.

Листинг 8.5. Тренажер (первая версия)

```
#include <iostream>
using namespace std;
// вывод приветствия
void hello(){ cout << "Приветствую вас!\n";
}
// Вывод инструкции
void instruction(){ cout << "Инструкция\n";
}
int main(){
    hello();                              // Приветствие
    instruction();                        // Инструкция
// переменные
// основная работа
    return 0;
}
```

Теперь займемся разработкой непосредственно тренажера. Задания должны нумероваться, чтобы знать, сколько всего заданий выполнил ученик. Нужно считать количество ошибок, совершенных учеником во время тренировки. Тренировка (цикл) завершается, когда в ответе ученик задаст ноль. После завершения цикла нужно вывести общее количество заданий и количество ошибок, сделанных учеником.

Понятно, что одна итерация цикла должна содержать следующие шаги:

1. Увеличить номер задания.
2. Сгенерировать случайные сомножители от 1 до 9.
3. Вывести задание.
4. Получить ответ пользователя.
5. Если ответ неправильный, то увеличить количество ошибок.

В п. 5 надо еще проверить, не ввел ли ученик ноль, и количество ошибок увеличивается, если ответ — не ноль.

Для генерации нам потребуется датчик случайных чисел, а диапазон зададим с помощью равномерного распределения целых чисел, как это было описано выше. Инициализацию сделаем функцией `time` (см. листинг 8.4).

Переписав план итерации на C++ в виде тела цикла, получим код главной функции нашей программы (листинг 8.6).

Листинг 8.6. Главная функция тренажера

```
int main(){
   hello();                              // Приветствие
   instruction();                        // Инструкция
   random_t gen { unsigned(time()) };    // датчик
   uniform_int_distribution<> uid(1,9); // диапазон случайных чисел
   size_t count_errors = 0;              // количество ошибок
   size_t tasks = 0;                     // количество заданий
   int L,R;                              // случайные множители
   int answer {-1};                      // ответ
   while(answer){                        // пока ответ не ноль
      L = uid(gen);  R = uid(gen);       // генерация сомножителей
      ++tasks;                           // считаем задания
      cout << setw(2) << tasks << "."    // номер задания выводить
           << setw(2) << L << " *"       // первый множитель
           << setw(2) << R << " = ";     // второй множитель
      cin >> answer;                     // ответ пользователя
      if((answer)&&(L*R != answer)){     // если ответ не ноль
         ++count_errors;
         cout << "Ошибка!" << endl;
      }
   }
   cout << "Всего заданий     = " << tasks-1 << endl;
   cout << "Количество ошибок = " << count_errors << endl;
   return 0;
}
```

Для первого входа в цикл начальное значение ответа сделано равным `-1`. Ошибки считаются, если ответ не равен нулю и неправильный. Когда ученик вводит нулевой ответ, то счетчик заданий `tasks` уже был увеличен на `1`, поэтому при выводе нужно эту единицу отнять.

Запустим программу на выполнение и сделаем пару ошибок, чтобы показать работу тренажера во всех случаях (рис. 8.2).

На этом первую версию программы можно считать завершенной. Функции приветствия и вывода инструкции реализуйте самостоятельно.

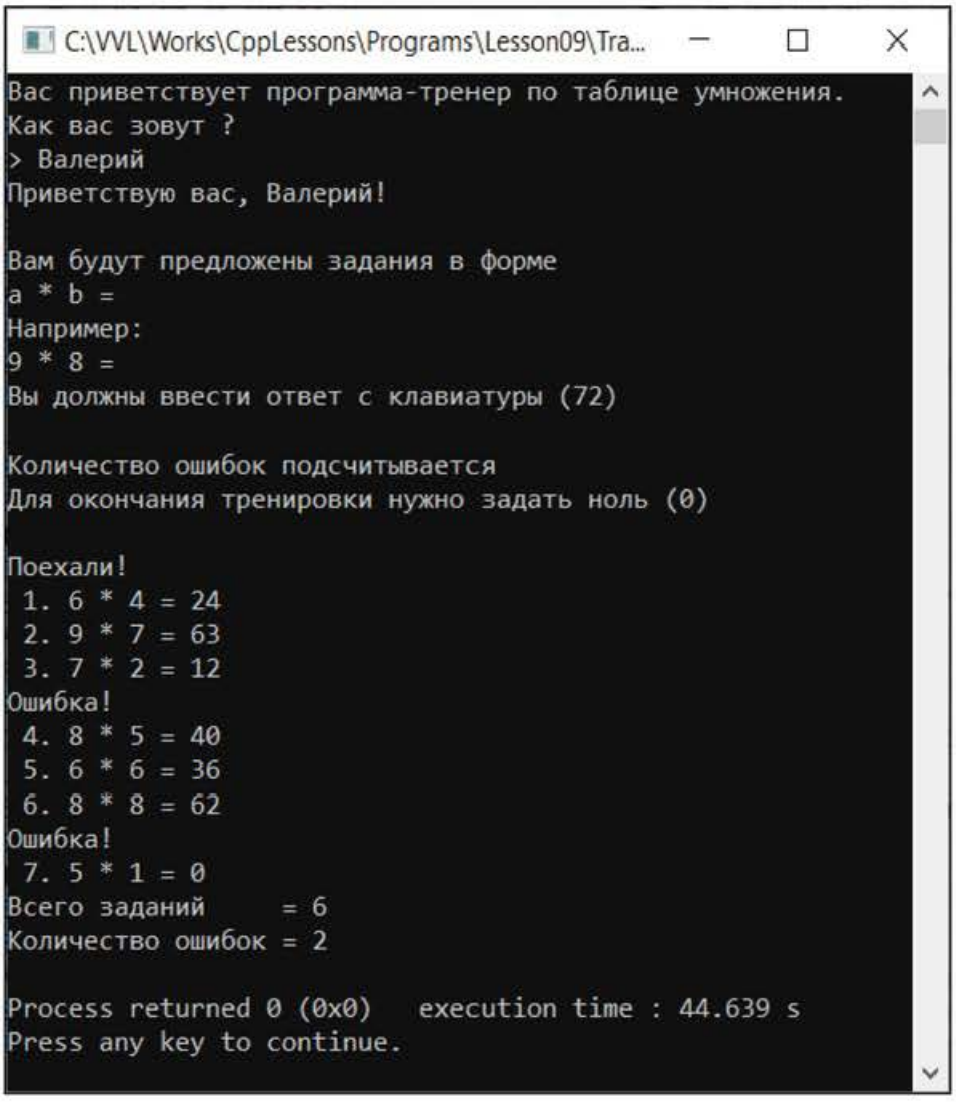


Рис. 8.2. Работа программы-тренажера

Оператор-переключатель

В C++ есть еще один оператор выбора решения — оператор-переключатель. Этот оператор очень удобен при создании меню. Например, мы уже написали три программы по таблице умножения:

- таблица умножения с конкретным множителем;
- полная таблица умножения;
- тренажер по таблице умножения.

Мы можем написать программу, которая выводит в консоль *меню*, предлагая ученику выбрать, какую из программ запустить, например:

```
Вы можете:
1. Вывести таблицу умножения с конкретным множителем
2. Вывести полную таблицу умножения
3. Запустить тренажер по таблице умножения
0. Выход
Ваш выбор:
```

Пользователь задает целое число, соответствующее одному из пунктов меню, и программа вызывает функцию, реализующую этот выбор. При вводе нуля программа завершает работу.

Выбор пункта меню в коде задается с помощью оператора-переключателя. В общем виде оператор-переключатель имеет форму:

```
switch (переключающее_выражение)
{ набор альтернатив }
```

Переключающим выражением может быть любой интегральный тип, т. е. целый, символьный и даже булевский. В общем виде одна *альтернатива* оператора имеет вид:

```
case константа: операторы;
```

Константа должна быть того же типа, что и переключающее выражение. В отличие от других операторов языка C++, в альтернативе можно задавать несколько операторов без фигурных скобок. Одна из альтернатив — особая, она начинается со слова `default`. Эту альтернативу можно не писать, но обычно она указывается последней. Все слова — зарезервированные слова языка C++. Оператор-переключатель работает следующим образом:

- вычисляется переключающее выражение;
- значение выражения поочередно сравнивается с константами в альтернативах;
- первое совпадение приводит к выполнению операторов совпавшей альтернативы;
- выполняются все остальные операторы во всех следующих альтернативах, в том числе и в `default`.

После этого выполняется следующий за `switch` оператор.

Если значение переключающего выражения не совпало ни с одной из констант, то выполняются операторы альтернативы `default`. А если ее нет, то происходит выход из блока альтернатив и выполняется следующий после `switch` оператор.

Однако обычно нужно выполнить операторы *совпавшей* альтернативы. Тогда последним оператором в этой альтернативе нужно написать известный нам оператор `break`, который и осуществит выход из блока альтернатив. В последней альтернативе оператор `break` можно не писать.

Оператор-переключатель вполне может заменить условный оператор `if`:

```
bool condition;
// вычисление переменной condition
switch(condition){
    case true:  // операторы блока «истина» ; break;
    case false: // операторы блока «ложь»   ; break;
}
```

Булевская переменная `condition` является переключающим выражением. Она как-то вычисляется перед оператором `switch`. Переключатель имеет две ветки в соответствии с возможными значениями булевской переменной. Если значение переменной `condition` равно `true`, то выполняются операторы в альтернативе `true` и происходит

выход из блока альтернатив. Вторая альтернатива работает, если значение переменной `condition` равно `false`.

На мой взгляд, такая запись гораздо понятнее условного оператора, т. к. здесь явно показаны значения логической переменной и соответствующие им последовательности операторов, которые нужно выполнять.

Система обучения арифметике

Начнем разрабатывать программу, которую будем дополнять и развивать по мере углубления наших знаний. Включим в нее наши три программы по таблице умножения. Такую большую программу можно даже назвать автоматизированной *системой* — системой обучения арифметике. Как обычно, начнем с первой простейшей версии, в которой используем оператор-переключатель. Код представлен в листинге 8.7.

Листинг 8.7. Система обучения арифметике (версия 1)

```
#include <iostream>
using namespace std;
// рабочие функции
void multiplier(){ cout << "множитель\n"; }
void table(){ cout << "полная таблица\n";}
void trainer(){ cout << "тренажер\n";}
int menu(){                                    // функция выбора пункта меню
    int number = -1;
    cout << "Вы можете:\n"
    "\t1. Вывести таблицу умножения с конкретным множителем\n"
    "\t2. Вывести полную таблицу умножения\n"
    "\t3. Запустить тренажер по таблице умножения\n"
    "\t0. Выход\n";
    cout << "Ваш выбор: "; cin >> number;
    switch(choice){
       case 1: multiplier();   break;
       case 2: table();        break;
       case 3: trainer();      break;
       case 0: break;
       default:
               cout << "Такого пункта меню нет!\n";
    }
    return number;
}
int main(){
    int choice = -1;                           // номер пункта меню
    while(choice){                             // пока не ноль
        choice = menu();                       // выбор пункта меню
    }
```

```
    cout << "Game Over!" << endl;              // сообщение о завершении
    return 0;
}
```

В этой первой версии самый важный код написан в функции `menu`, которая взаимодействует с пользователем и организует работу всех составных частей (модулей) нашей обучающей системы. Главная программа просто вызывает эту функцию в цикле.

Модуль — это отдельная часть системы, которую можно разрабатывать почти независимо от всей остальной системы. Разработку модуля обычно можно поручить отдельному программисту. В нашей программе модулями можно считать функции `multiplier`, `table` и `trainer`.

Пока наши модули находятся в одном файле, но потом мы научимся объединять несколько файлов кода в одну программу.

Функцию `multiplier` нужно создать из программы в листинге 6.3. Напомню, что мы писали функцию `tables` (см. листинг 7.3), которая получала множитель как параметр и выводила таблицу умножения с этим множителем. Можно использовать эту функцию, вызвав ее из функции `multiplier`.

Функция `table` создается из программы, которая выводит полную таблицу умножения (см. листинг 6.6). Функция `trainer` создается из написанного нами в этом уроке тренажера (см. листинг 8.6). Создайте эти функции самостоятельно.

Пишем калькулятор

Для освоения оператора-переключателя напишем упрощенную версию калькулятора. Программа вводит выражение из двух чисел и знака операции, которую требуется выполнить с этими числами. Числа можно задавать как целые, так и дробные. Знак операции — это один из символов `+`, `-`, `*`, `/`. Выражение нужно задавать без пробелов в таком виде:

```
1.1+2.67
2*3.4
78/3
4-1.5
```

Сначала напишем программу на C++, а затем разберем ее работу. Код программы-калькулятора представлен в листинге 8.8.

Листинг 8.8. Калькулятор с оператором *switch*

```
#include <iostream>
using namespace std;
int main(){
   double a, b; char op {'?'};          // операнды и операция
   cout << "Задайте выражение (без пробелов): ";
```

```
   cin >> a >> op >> b;                  // ввод выражения
   switch (op)
   { case '+':  cout << a+b; break;              // сложение
     case '-':  cout << a-b; break;              // вычитание
     case '*':  cout << a*b; break;              // умножение
     case '/':  if (b!=0) cout << a/b;           // деление
                else cout << "Делитель равен нулю!\n";
                break;
     default: cout << "Неверный знак операции!" << endl;
   }
   return 0;
}
```

Рассмотрим оператор ввода. Первое вводимое число (до знака операции) заносится в переменную `a`. Потом вводится один символ — знак операции. После знака операции вводится второе число в переменную `b`. После ввода написан оператор-переключатель, в котором переключающим выражением является символ операции (один из интегральных типов).

Обращаю ваше внимание на то, что числа могут быть как положительные, так и отрицательные. Например, вполне допустимо вводить такие выражения:

```
-1.1+2.67
2*-3.4
-78/-3
```

Все эти выражения правильно вычисляются с учетом знака.

Пишем еще один тренажер

Напишем еще один тренажер для обучения сложению положительных двузначных чисел. Используем наш первый тренажер (см. листинг 8.6), чтобы не писать все с самого начала. Основные изменения касаются оператора вывода инструкции и задания. Надо реализовать вывод задания по формату в виде отдельной функции с двумя параметрами. Параметры — это генерируемые в тренажере случайные числа. Еще нужно изменить проверку ответа, т. к. операция в новом тренажере другая. Функция показана в листинге 8.9.

Листинг 8.9. Функция вывода инструкции и функция вывода задания

```
void instruction(){
    cout << "\n Вам будут предложены задания в форме\n"
            " a \n+\n b \n---\n"              // вывод будет в столбик
            " Например:\n"
            " 18\n+\n 36\n---\n"              // вывод будет в столбик
            " Вы должны ввести ответ с клавиатуры (54)\n"
            " Можно набирать с пробелом (_54)\n"
```

```
            " Для окончания тренировки нужно задать ноль (0)\n"
            "\n Количество ошибок подсчитывается\n";
}
void task(int L, int R){
    cout << setw(3) << L << "\n+ \n" << setw(3) << R << "\n----\n";
}
```

В функции `instruction` я разбил строковый литерал на несколько частей, чтобы было проще подгонять позиции при выводе. Эти функции вызываются в главной функции (листинг 8.10).

Листинг 8.10. Главная функция тренажера сложения

```
int main(){
    instruction();                              // Инструкция
    random_t gen { unsigned(time()) };
    uniform_int_distribution<> uid(10,99);      // диапазон случ. чисел
    size_t count_errors = 0;                    // количество ошибок
    size_t tasks = 0;                           // количество заданий
    int L,R;                                    // случайные множители
    int answer {-1};                            // ответ
    while(answer){
        L = uid(gen);  R = uid(gen);            // генерация
        ++tasks;                                // номер задания
        task(L,R);                              // вывод задания
        cin >> answer;                          // ввод ответа
        if(answer)
          if(L+R != answer){                    // проверка ответа
            ++count_errors; cout << "Ошибка!\n";
          }
    }
    cout << "Всего заданий     = " << tasks-1 << endl;
    cout << "Количество ошибок = " << count_errors << endl;
    return 0;
}
```

Код улучшился, т. к. вывод задания вынесен в отдельную функцию.

Запустим программу и сначала посмотрим, как выводится инструкция (рис. 8.3).

Сопоставьте текст в консоли с кодом функции в листинге 8.9. Такой анализ очень помогает понять, как писать код для вывода в требуемом виде.

Теперь снова запустим программу и решим два-три задания. Завершение работы осуществляется вводом нуля, как и в первом тренажере.

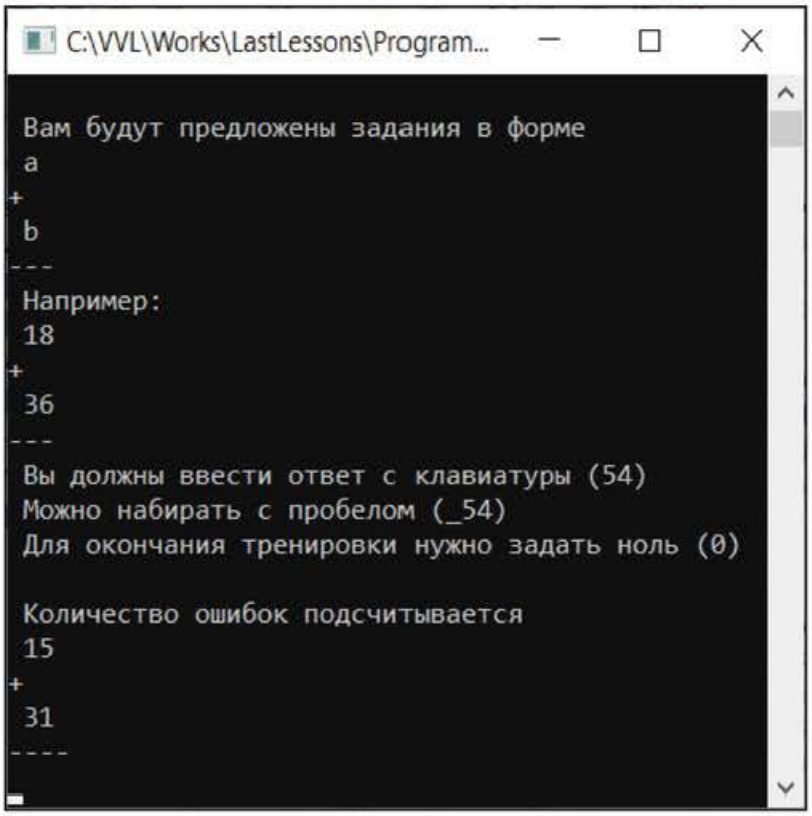


Рис. 8.3. Инструкция для пользователя

Задачи и упражнения

1. Реализуйте функции `multiplier`, `table` и `trainer` тренажера в листинге 8.7.
2. Напишите функции вывода инструкции для тренажера в листинге 8.7.
3. Реализуйте тренажер по вычислению площадей нескольких разных фигур: треугольника, круга, прямоугольника.

- ЧАСТЬ II -

РАБОТА С ДАННЫМИ В C++

УРОК 9. МАССИВЫ
УРОК 10. СОЗДАЕМ НОВЫЕ ТИПЫ
УРОК 11. ЗНАКОМСТВО С УКАЗАТЕЛЯМИ
УРОК 12. МАССИВЫ И ФУНКЦИИ
УРОК 13. СТРОКИ, СТРОКИ, СТРОКИ
УРОК 14. СНОВА О ФУНКЦИЯХ
УРОК 15. ПЕЧАЛЬНАЯ ДЕЙСТВИТЕЛЬНОСТЬ

- Урок 9 -

Массивы

Все элементы данных, которые мы рассматривали до сих пор, являлись «одиночными», или, как принято их называть, *скалярными*. Между тем большинство реальных задач должны обрабатывать группы данных, как правило, довольно большого объема. Например, ежемесячное начисление заработной платы работникам предприятия. Или ежегодное вычисление отпускных для тех же работников того же предприятия, где надо использовать ежемесячные начисления. В таких задачах мы имеем дело с группой *однородных* данных.

Группа данных может быть устроена совершенно по-разному. Однако самая простая организация данных — это последовательность. Последовательность однородных данных называется *массивом* — это название принято во всех языках программирования без исключения. В языке C++ существует два вида массивов: встроенные и реализованные. Встроенный массив — это конструкция C++, которая досталась по наследству от C. Реализованные массивы определены в стандартной библиотеке.

Встроенные массивы относятся к *составным типам* данных. Иногда говорят, что массив — это один из *агрегатов* данных. В языке C++ существуют и другие виды составных типов: структуры, объединения, классы. С некоторыми другими составными типами данных мы постепенно познакомимся.

Зачем нужны массивы

Чтобы разобраться, что такое массивы и зачем они нужны, давайте напишем программу, которая будет вычислять, какой день недели приходится на заданное число и месяц текущего года (2024). Например, 1 марта 2024 года — это пятница, а 1 мая — среда.

Очевидно, нам потребуется переменная, в которой будет храниться заданный месяц. Будем представлять месяц как целое число (номер месяца в году): январь — `1`, февраль — `2` и т. д. Нужна также и переменная, в которой будет находиться заданное число месяца. Кроме того, нам требуется 12 чисел, представляющих количество дней в месяце: в январе — 31 день, в феврале — 28 дней (29 в 2024 году) и так далее до декабря.

Алгоритм понятен:

1. Сложить количество дней всех месяцев до заданного.
2. Прибавить заданное число дней.
3. Получить остаток от деления на 7.

Кроме того, мы должны учесть, с какого дня недели начинается текущий год (с понедельника) и добавить количество дней, оставшихся в прошлом году (для 2024 года — ни одного, т. е. ноль). Это и будет искомый день недели.

Будем считать, что понедельник — это число 1, вторник — число 2 и т. д. Воскресенье — это число 0. Например, посчитаем, на какой день недели приходится 1 мая 2024 года. Вычисляем сумму дней четырех месяцев до 1 мая: 31 + 29 + 31 + 30 = = 121 день; добавляем заданное число: 121 + 1 = 122; вычисляем остаток от деления: 122 / 7 = 3. Ответ — среда.

Вроде бы все просто. Но нам требуется написать универсальную программу, которая вычисляет день недели для любого числа 2024 года. И тут нас поджидают проблемы с вычислением суммы дней месяцев до заданного — мы должны предусмотреть 12 вариантов суммы. Если месяц первый (январь), то сумма равна 0. Для каждого следующего месяца мы должны добавлять количество дней предыдущего. Напишем функцию `days` (дни), которая по номеру месяца вернет сумму дней всех месяцев до заданного. Корректность параметра не проверяем. Код представлен в листинге 9.1.

Листинг 9.1. Вычисление суммы дней по номеру месяца

```
int days(int month){
    int summa{0};
    if(month == 2)        summa = 31;       // до февраля только январь
    else if(month == 3)   summa = 31+29;
    else if(month == 4)   summa = 31+29+31;
    else if(month == 5)   summa = 31+29+31+30;
    else if(month == 6)   summa = 31+29+31+30+31;
    else if(month == 7)   summa = 31+29+31+30+31+30;
    else if(month == 8)   summa = 31+29+31+30+31+30+31;
    else if(month == 9)   summa = 31+29+31+30+31+30+31+31;
    else if(month == 10)  summa = 31+29+31+30+31+30+31+31+30;
    else if(month == 11)  summa = 31+29+31+30+31+30+31+31+30+31;
    else if(month == 12)  summa = 31+29+31+30+31+30+31+31+30+31+30;
    return summa;
}
```

Однако сразу становится понятно: мы имеем возможность использовать такую схему только потому, что вариантов мало — всего 12. Но большинство программ должны выполнять гораздо большее количество (десятки, сотни тысяч или даже миллионы) таких вычислений. Поэтому нам требуется так организовать константы, чтобы можно было как-то применить оператор цикла. Именно в таких случаях ис-

пользуется массив. Данные однородны, поскольку все они имеют один и тот же тип — в нашем случае целый.

Встроенные массивы

Встроенные массивы не всегда удобны в использовании, однако они достались C++ по наследству от C и присутствуют во многих реальных проектах. Поэтому программист обязан их знать и уметь применять.

Объявление массивов

Массив — это объект программы, имеющий имя, поэтому его надо объявить, как любую переменную:

```
тип имя [количество];
```

В этом объявлении `тип` может быть как встроенным, так и реализованным (например, `string`), а `количество` определяет, сколько элементов указанного типа массив может содержать. Традиционно количество элементов массива называют *размером* (size) или *длиной* (length). Массив с единственным размером в объявлении называется *одномерным*.

Встроенные массивы являются массивами *фиксированного* размера. Размер (количество элементов) можно задавать константным выражением. Заданный размер впоследствии нельзя никак изменить, так же как нельзя изменить имя и тип переменной.

> ***На заметку***
>
> В C++ есть динамические массивы, размер которых можно задавать во время выполнения программы. Но и в этом случае размер массива изменить нельзя. Мы рассмотрим такие массивы после изучения указателей (см. *урок 11*).

В нашей задаче нужно объявить массив из 12 элементов, соответствующих месяцам года:

```
int month[12];
```

Как и простую переменную, массив можно инициализировать, например:

```
int month[12] = { 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
```

Такая форма очень похожа на универсальную инициализацию переменных (см. *урок 3*). Она была унаследована из языка C, и использовать ее можно было только для массивов и структур. Называется эта форма инициализации *агрегатной*. Нынешний стандарт позволяет писать без знака равенства:

```
int month[12]{31,29,31,30,31,30,31,31,30,31,30,31};
```

При инициализации массива разрешается не указывать размер:

```
int month[]{31,29,31,30,31,30,31,31,30,31,30,31};
```

В этом случае количество элементов определяется количеством констант. Если задано количество, а констант задано меньше (больше задать нельзя — программа не

компилируется), то оставшиеся элементы массива заполняются нулями. Таким образом, обнулить массив с заданным количеством элементов можно очень просто:

```
int v[10] = {0};
int v[10]{0};
int v[10]{};
```

Все эти способы в настоящее время эквивалентны. Если размер массива заранее известен, то лучше его задавать, т. к. это убережет от случайной ошибки, когда количество констант окажется больше положенного. И кстати, объявление массива без размера с инициализацией:

```
int v[] = {0};
int v[]{0};
```

обозначает массив из *одного* элемента, и этот элемент равен нулю. А третий вариант объявления без нуля в фигурных скобках:

```
int v[]{};
```

является ошибкой (программа не компилируется).

Массив можно объявить константным, если значения его элементов не изменяются. Например, массив `month` вполне можно сделать константным:

```
int const month[12]{31,29,31,30,31,30,31,31,30,31,30,31};
```

Причем, размер тоже можно не указывать. И можно использовать `constexpr`:

```
constexpr int month[]{31,29,31,30,31,30,31,31,30,31,30,31};
```

Заметим, что нельзя объявлять массив с помощью `auto` — программа просто не компилируется, несмотря на наличие инициализации.

При инициализации массива нельзя задавать константы более широкого типа, чем тип элементов массива. Например, для массивов целых типов нельзя задавать дробные константы:

```
int h[] {5,4,3,2,1.1};          // ошибка компиляции
int h[5] {5,4,3,2,1.1};         // ошибка компиляции
int h[] = {5,4,3,2,1.1};                // ошибка компиляции
```

Это требует сужающих преобразований, и компилятор не пропускает такие объявления. Другой пример недопустимой инициализации:

```
int h[] = {5,4,3,2,1,7'000'000'000'000};                // ошибка компиляции
```

В этом случае тоже возникает ошибка компиляции, так как 7 триллионов не помещается в диапазон целых. А вот «плавающий» массив инициализировать целыми вполне допустимо.

Инициализация массива другим массивом невозможна:

```
int v[12] = month;              // ошибка компиляции
int v[12] (month);              // ошибка компиляции
int v[12] {month};              // ошибка компиляции
```

Компилятор выдает ошибку компиляции на все подобные объявления.

Синонимы типов для массивов

Для массивов тоже можно объявлять синонимы типов двумя способами: с помощью using-объявления и в операторе `typedef`. Оператор `typedef` для массивов имеет несколько необычный синтаксис:

```
typedef тип Имя [количество];
```

`Имя` — это имя нового типа для массивов, и оно задается между типом элементов и размером массива. Количество должно быть константным выражением. Это имя типа можно использовать для объявлений массивов указанного типа и размера, например:

```
typedef double DoubleArray [10];
DoubleArray B, D, P;
```

Все эти массивы имеют размер, объявленный в операторе `typedef`.

Размер в операторе `typedef` можно не указывать, но тогда при объявлении массива, конечно, требуется инициализация:

```
typedef double DoubleArray [];
DoubleArray B{1,2,3,4,5};
```

То же самое можно сделать с помощью using-объявления, например:

```
using uiarray = unsigned int[];
uiarray F {1,2,3,4,5};
```

Становится понятным, что тип массива включает и тип элементов, и размер. Массивы разного размера являются массивами разных типов даже при одинаковом типе элементов.

Доступ к элементам

Каждый элемент массива имеет номер, который обычно называют *индексом*. В C++ принято по наследству от C нумерацию (индексацию) элементов в массиве начинать с нуля: если в массиве содержится `n` элементов, то их индексы изменяются в диапазоне от `0` до `n-1`. Говорят, что индекс изменяется на полуинтервале [0, n). Эта запись означает, что правая граница диапазона не включается в интервал. Такая индексация стала общепринятой, и сейчас во всех языках программирования начальный индекс массивов равен нулю.

Чтобы использовать отдельный элемент массива в программе, мы должны написать имя массива и после него — индекс элемента в квадратных скобках. Таким образом, в нашем случае `month[0]` соответствует январю, `month[1]` — февралю и так далее до декабря (`month[11]`).

В общем случае индекс представляет собой выражение и может вычисляться во время работы программы. Диапазон изменения индекса говорит о том, что это выражение приводится к беззнаковому целому. Поэтому еще в языке C для индексов (и размеров массивов) в стандартной библиотеке был определен синоним `size_t`, о котором уже было упомянуто ранее (см. *урок 4*).

Размер массива

Как было сказано выше, размер массива задается константным выражением. Если же размер не задан, то бывает необходимо выяснить, сколько элементов в нем содержится и сколько места массив занимает в памяти.

В прежние времена это делалось с помощью хорошо известной нам операции `sizeof` — надо размер массива разделить на размер элемента, например:

```
cout << sizeof(month)<<' '<< sizeof(month)/sizeof(month[0]) << endl;
```

Этот оператор выведет на экран числа `48` и `12` — размер массива в памяти и количество элементов в массиве.

То же самое можно сделать, разделив размер массива на размер типа его элементов:

```
cout << sizeof(month)<<' '<<sizeof(month)/sizeof(int) << endl;
```

Этот оператор выведет на экран те же числа `48` и `12`.

Я упомянул об этом способе, т. к. такие выражения вполне могут встретиться в реальном коде. Однако в современном C++ так уже не делают, а используют стандартные средства, включенные в C++17 (см. ниже).

Операции с массивами

Никакие операции (из известных нам) со встроенными массивами выполнять нельзя. Массивы нельзя присваивать, нельзя складывать и вычитать, нельзя вводить и выводить, нельзя сравнивать и т. д. Все операции разрешено выполнять только с элементами массивов. Например, операция присваивания с массивами вызывает ошибку компиляции:

```
int v[10], w[10];
v = {1, 2, 3, 4, 5};                    // ошибка компиляции
```

Хотя такая запись очень похожа на инициализацию, компилятор «ругается» и не пропускает такую конструкцию. Нельзя также присвоить один массив другому:

`w = v;` — это тоже не компилируется.

Поэтому все, что нужно сделать с массивами, делается поэлементно в цикле. Обычно с массивами используется оператор цикла со счетчиком `for`. Например, следующий цикл выведет в консоль в столбик все константы из массива `month`:

```
for (size_t = 0; i < 12; ++i) cout << month[i] << endl;
```

Аналогично массив можно заполнить, вводя элементы массива с клавиатуры, если массив объявлен как не константный:

```
double mm[20];
for (size_t i = 0; i < 20; ++i) cin >> mm[i];
```

Точно так же можно заполнить весь массив одним и тем же значением:

```
double m[100];
for(size_t i = 0; i < 100; ++i) m[i] = 1.2;
```

Конечно, можно заполнить массив случайными числами, как описано ранее (см. *урок 8*). Если объявлен объект датчик `gen`, то его можно использовать в цикле справа от присваивания:

```
for(size_t i = 0; i < 100; ++i) m[i] = gen();
```

Естественно, можно во всех этих ситуациях использовать и другие виды циклов, например:

```
size_t i = 0; while(i < 100){ m[i] = 1.2; ++i; }
```

Можно заполнить массив возрастающими значениями, например:

```
for(size_t i = 0; i < 100; ++i) m[i] = i;
```

Точно так же поэлементно можно выполнять арифметические операции с массивами, например:

```
for (size_t i = 0; i < 10; ++i) w[i] *= 2;
```

Этот цикл удваивает все элементы массива `w`.

Скопировать (присвоить) один массив в другой можно только поэлементно, используя тот же цикл `for`:

```
for (size_t i = 0; i < 10; ++i) w[i] = v[i];
```

Естественно, размеры массивов должны быть одинаковыми. Или, по крайней мере, размер принимающего массива (массив `w`) должен быть не меньше копируемого (массив `v`). И цикл должен выполняться до размера меньшего массива (массив `v`).

Теперь мы можем написать программу вычисления дня недели. Вместо номера дня можно выводить название дня недели. Нужно объявить массив строк и инициализировать его:

```
string Days[]{"воскресенье", "понедельник", "вторник",
              "среда", "четверг", "пятница", "суббота"
             };
```

Код показан в листинге 9.2. Корректность входных данных не проверяется, чтобы не загромождать программу.

Листинг 9.2. Вычисление дня недели

```
#include <iostream>
#include <string>
using namespace std;
int main(){
   int day, m;
   constexpr int month[]{31,29,31,30,31,30,31,31,30,31,30,31};
   cout <<"Задайте день и месяц: ";  cin >> day >> m;
   int s {};
   for(int i = 0; i < m-1; ++i)
      s += month[i];                          // считаем месяцы
   s += day;                                  // добавляем дни
```

```
    const string Days[]{"воскресенье", "понедельник", "вторник",
                        "среда", "четверг", "пятница", "суббота"
                       };
    cout << Days[s%7] << endl;          // вывод названия
    return 0;
}
```

Заметим, что воскресенье специально поставлено первым элементом массива (индекс ноль), что позволило упростить вычисление остатка от деления.

Проверка нашей программы показывает, что при условии ввода корректных данных она работает правильно.

Новые массивы

Встроенные массивы C++ имеют достаточно много ограничений, и код для их обработки часто является источником ошибок. Поэтому в каждом новом стандарте в язык и библиотеку добавляются средства, повышающие корректность и безопасность кода, улучшающие читабельность.

Размер и тип массива

В стандарте C++17 в стандартную библиотеку была добавлена функция[1] `size`, с помощью которой можно получить размер (количество элементов) встроенного массива:

```
cout << size(month) << endl;
```

Функция правильно определяет размер даже при отсутствии количества элементов в объявлении массива. Использование этой функции вместо явно заданной длины делает код гораздо понятнее и, что очень важно, намного безопаснее. «Безопаснее» означает, что в коде будет меньше ошибок. И, более того, станет невозможно совершить некоторые ошибки.

Эту функцию можно использовать в объявлении массива для указания размера. Например, нам нужно, чтобы три массива всегда имели одинаковый размер, а при изменении одного из них нужно «синхронно» изменить размеры двух других. В прежние времена объявлялась константа, и она указывалась в качестве размера во всех трех массивах. Теперь это можно сделать и с помощью функции `size`:

```
double A[100];
double B[size(A)];
double D[size(A)];
```

[1] На самом деле это *шаблон* функции, но в данном случае это не существенно.

Новые средства позволяют объявить другой массив не только одинаковым по размеру с уже объявленным, но и тип элементов выбрать такой же:

```
int h[] = {5,4,3,2,1,7};                    // массив целых, размер 6
decltype(h) D;                              // массив целых, размер 6
```

Зарезервированное слово `decltype` сообщает компилятору, что тип элементов и размер массива `D` должны быть такими же, как и у массива `h`. Массив `D` можно инициализировать другими константами или обнулить.

Тип элементов массива можно позаимствовать у другой переменной или даже у литерала, например:

```
decltype(var) m[10];            // тип такой же, как у переменной var
decltype(1.0) mm[size(m)];
```

В первом объявлении тип элементов массива `m` будет такой же, как и тип переменной `var`, объявленной ранее в программе. А во втором случае объявлен массив с элементами типа `double`, т. к. литерал `1.0` имеет тип `double` по умолчанию.

Новый цикл *for*

В стандарте C++11 добавлен новый вид цикла `for`, который называется «цикл по диапазону» (range-for). Он позволяет обрабатывать все элемента массива, вообще не указывая его размера. Пусть в программе объявлен массив:

```
double V[10];
```

Он как-то был заполнен во время работы программы, и мы хотим просто вывести значения элементов в консоль через пробел. Это делается так:

```
for(auto t: V) cout << t << ' ';
```

В скобках до двоеточия объявляется переменная, а после двоеточия — имя массива, с которым выполняется работа. Цикл работает так: в переменную `t` копируется очередной (начиная с первого с индексом 0) элемент массива `V`, и выполняется оператор тела. И так до конца массива.

Однако и в этом новом операторе цикла есть свои тонкости. Например, нам нужно удвоить все элементы массива, и мы пишем код:

```
for(auto t: V) t *= 2;
```

Однако элементы массива не изменяются, в чем можно убедиться, написав цикл для вывода элементов. Фокус в том, что объявление переменной в таком цикле сделано похожим на объявление параметров функции. В примере с удвоением параметр `t` объявлен как передаваемый по значению. Чтобы изменять элементы массива, надо передать параметр `t` по ссылке:

```
for(auto &t: V) t *= 2;
```

Теперь все в порядке, все элементы массива удвоились.

Как и в функциях, вместо передачи по значению можно объявить параметр передаваемым по константной ссылке, например:

```
for(const auto &t: V) cout << t << ' ';
```

И работает это так же, как в функциях.

Массив *array*

В стандартной библиотеке C++ реализованы последовательные контейнеры, которые обеспечивают более широкие возможности, чем встроенные массивы. Еще в C++03 в библиотеку был включен контейнер `vector`, а в C++11 появился контейнер `array`, который мы и рассмотрим. Английское слово array переводится как «набор», «ряд», но в программировании оно с самого начала переводилось как «массив». Так мы и будем называть контейнер `array`. Он предназначен для замены встроенных массивов. В стандарте C++20 появилась функция `to_array`, название которой говорит о ее назначении: создание массива `array` из встроенных конструкций.

Объявление и инициализация

Далее считаем, что в программе прописана using-директива, чтобы не писать префикс `std::`. Объявление массива `array` выглядит так:

```
array<тип, размер> имя;
```

Угловые скобки показывают нам, что `array` — это класс-шаблон. Тип, как и у встроенных массивов, может быть любой. Размер, как обычно, является константным выражением.

Инициализация тоже очень похожа на инициализацию встроенных массивов:

```
array<int,5> m {1,2,3,4,5};     // тип и размер заданы явно
array d {1,2,3,4,5,6};          // тип и размер выводятся
```

В первом случае явно заданы и тип, и размер массива, а во втором компилятор выводит тип и размер из набора инициализирующих констант. Очевидно, тип элементов массива `d` — это `int`, а количество равно 6.

При объявлении массива нельзя задать только тип без размера или размер без типа. Размер, естественно, можно позаимствовать как из массива `array`, так и из встроенного массива, например:

```
array<int, size(d)> W;          // array d объявлен выше
int h[] = {5,4,3,2,1,7};
array<int, size(h)> V;
```

Можно использовать спецификатор `decltype` аналогично тому, как мы это делали выше для встроенных массивов.

В отличие от встроенных массивов, массивы `array` можно инициализировать массивом, причем как встроенным, так и другим array-массивом:

```
array t{W};             // универсальная array-массивом
array tt{h};            // универсальная встроенным массивом
array m = W;            // инициализация только array-массивом
array mm(W);            // инициализация только array-массивом
```

В C++20 объявление массива можно делать с помощью `auto`:

```
auto arr{d};
auto mmm = to_array(h);                 // преобразование в array
auto mm = to_array({1,1,1,4,6});        // преобразование в array
```

Массив array «знает» свой размер. Если он нам в программе потребуется, то его можно получить двумя способами:

- с помощью функции size;
- с помощью *метода* size.

О функции size написано выше, а метод size используется точно так же, как мы использовали метод length для получения размера строки (см. листинг 3.2). Например, можно объявить массив W (см. выше) вот так:

```
array<decltype(d), d.size()> W;
```

В общем, каждый выбирает для себя наиболее удобный способ.

Выясним размер памяти, которую занимает array-массив, с помощью все той же функции sizeof:

```
cout << sizeof(array<int,10>) << ' '
     << sizeof(array<int,20>) << ' '
     << sizeof(array<double,10>) << endl;
```

Этот оператор выведет в консоль числа:

```
40 80 80
```

Эти числа однозначно показывают, что для array-массива выделяется ровно столько же памяти, сколько и для встроенного массива. Это зависит и от типа, и от количества элементов.

Операции с массивом *array*

В отличие от встроенных массивов массивы array можно присваивать без использования циклов, как мы делали это для скалярных переменных. Тип элементов и размеры у обоих массивов должны быть одинаковыми.

Поскольку присваивание массивов работает, то их можно и обменивать друг с другом, как мы это делали со скалярными переменными в задаче о кирпиче и отверстии (см. листинг 5.5). И делается это опять двумя способами: с помощью функции swap или с помощью одноименного метода. Если A и B объявлены как array-массивы с одинаковым типом элементов и размером, то обменять их можно так:

```
std::swap(A,B);          // функция обмена
A.swap(B);                       // метод обмена
```

К сожалению, присвоить array-массиву встроенный массив нельзя, так же как и наоборот. Но с помощью функции to_array можно преобразовать встроенный массив того же типа и размера в array-массив и присвоить его:

```
mm = to_array(h);                // см. выше объявления mm и h
```

Доступ к элементам массива array делается точно так же, как и к элементам встроенного массива, — по индексу. Но еще для доступа к первому элементу (с индексом 0) реализован метод front, а для доступа к последнему (с индексом size – 1) — метод back.

Массив надо заполнять, поэтому реализован метод `fill`. Метод позволяет заполнить весь массив одним значением, не используя цикл:

```
m.fill(-1);
```

В стандартной библиотеке реализовано еще много других средств для работы с массивами и другими контейнерами. Мы с ними постепенно познакомимся.

Объекты типа *initalizer_list*

Вернемся к примерам инициализации массивов в универсальной форме. Мы выше писали примеры:

```
array d {1,2,3,4,5,6};                    // тип и размер выводятся
int h[] = {5,4,3,2,1,7};                  // array d объявлен выше
```

Инициализирующее выражение в фигурных скобках — это не просто набор каких-то элементов, это объект специального типа `initalizer_list`, который был добавлен в C++11 для того, чтобы инициализация массивов выполнялась в универсальной форме. Такие объекты можно объявлять и присваивать:

```
initializer_list rd {9,8,7,6,5,4};
initializer_list L {1,2,3};
L = rd;
```

Объект такого типа фактически является массивом констант небольшого размера. Поэтому им можно инициализировать array-массивы:

```
array g{rd};
```

Объекты типа `initalizer_list` знают свой размер и выводятся в консоль:

```
for(auto t: rd) cout << t << ' ';
for(auto t: {5,4,3,2,1}) cout << t << ' ';
```

Такие объекты можно даже объявлять параметрами функций и передавать как аргументы (см. выше объявление `auto mm`).

Решаем задачи

Так как массивы были исторически первым составным типом данных, за время развития программирования накоплен достаточно большой набор типовых решений различных задач. Обычно эти решения называются типовыми алгоритмами. Давайте напишем несколько программ, в которых реализуем типовые алгоритмы обработки *числовых* массивов:

- поиск максимума, минимума и других задаваемых значений;
- вычисление различных сумм и произведений;
- преобразование массива: выполнение действий над всеми элементами;
- сортировка (упорядочение) элементов массива по возрастанию (маленькие в начале) или по убыванию (большие в начале);
- получение из двух массивов третьего;
- удаление, вставка и замена элементов массива.

Подобных задач с массивами в профессиональном программировании не счесть. Естественно, в стандартной библиотеке C/C++ реализованы решения многих из них. Но мы же хотим стать профессионалами, поэтому должны не просто использовать эти решения, а понимать, как они работают. Лучший способ достичь понимания — самому написать решение задачи.

Поиск минимума

В задаче о дне недели 2024 года мы уже суммировали элементы массива, поэтому начнем с решения задач поиска. Выше мы демонстрировали приемы работы с массивами с помощью цикла со счетчиком, теперь для тренировки будем использовать цикл `while`. Для упрощения кода во всех программах будем задавать инициализированный массив из 10 элементов.

Напишем алгоритм на псевдокоде:

```
1. Задать начальное значение минимума.
2. i = 0;
3. Пока i < размер {если число[i] < минимум, то минимум = число[i]; ++i }
4. Вывести минимум.
```

Единственный вопрос: какое значение первоначально задать в переменной минимума. Можно задать в минимуме значение элемента `numbers[0]`. Если минимальное значение было в `numbers[0]`, то оно так и останется в минимуме. А если в процессе проверки выяснится, что в каком-то `numbers[i]` значение меньше, то оно и попадет в минимум.

Код первой версии программы поиска минимума приведен в листинге 9.3.

Листинг 9.3. Поиск минимума (версия 1)

```
#include <iostream>
#include <limits>
using namespace std;
int main(){
   int numbers[]{45,38,81,30,3,27,3,320,15,31};
   int minimum = numbers[0];                    // первое значение
   size_t i = 1;                                // сравниваем со следующим
   while(i < size(numbers)){
     if(numbers[i] < minimum) minimum = numbers[i];     // минимум
     ++i;
   }
   cout << minimum << endl;
   return 0;
}
```

Программа работает и выводит на экран значение `3`, которое действительно является минимальным в массиве `numbers`.

Однако код можно существенно упростить. Вместо цикла `while` используем новый цикл `for` — это позволит избавиться от явной индексации элементов и проверки окончания цикла. Вместо встроенного массива используем массив `array`. Результатом будет не только минимум, но и его индекс в массиве.

Код новой версии показан в листинге 9.4.

Листинг 9.4. Поиск минимума (версия 2)

```
#include <iostream>
using namespace std;
int main(){
   array numbers{45,13,81,7,38,27,7,320,15,31};
   size_t i = 0;                          // индекс по элементам
   int minimum = numbers.front();         // number[0]
   size_t imin = 0;                       // индекс минимума
   for(auto t: numbers){
     if(t < minimum){
        minimum = t;                      // минимум
        imin = i;                         // индекс минимума
     }
     ++i;
   }
   cout << imin << ": " << minimum << endl;    return 0;
}
```

Код стал значительно безопаснее — проверки индексов отсутствуют.

Код поиска максимума отличается только одним символом: в условии оператора `if` надо операцию сравнения «меньше» заменить на операцию «больше». Можете самостоятельно убедиться в этом, скопировав код из листинга 9.4 и заменив операцию сравнения.

Подсчет максимумов

В массиве бывает не один минимум/максимум, а несколько (листинг 9.5). Интересно было бы подсчитать, сколько максимумов в массиве. Самый простой алгоритм состоит в том, чтобы сначала найти максимум (что мы уже умеем), а потом еще раз «пробежаться» по массиву и подсчитать, сколько раз он встретится. Однако вопрос состоит в том, чтобы определить количество максимумов за один проход по массиву, а не за два.

Это можно сделать, если начинать считать сначала при каждой смене максимума в условном операторе. Код представлен в листинге 9.5.

Листинг 9.5. Количество максимумов в массиве

```
#include <iostream>
#include <array>
```

```
using namespace std;
int main(){
   array numbers{ 5,3,1,6,7,7,2,5,7,4 };
   int maximum = numbers.front();    // number[0]
   size_t count_m = 0;               // количество maximum
   for(auto t: numbers){
     if(t == maximum) ++count_m;
     if(t > maximum) {
       maximum = t;                  // заменяем максимум
       count_m = 1;                  // начинаем считать сначала
     }
   }
   cout << maximum << ": " << count_m << endl;
   return 0;
}
```

Программа для массива `numbers` выводит:

```
7: 3
```

Поменяйте числа в массиве и убедитесь, что программа правильно находит количество максимумов.

Поиск заданного числа

Теперь напишем программу, которая ищет заданное число в массиве. Это первый алгоритм, который изучается профессиональными программистами в дисциплине «Структуры данных и алгоритмы». Структура данных здесь массив — неупорядоченная последовательность чисел.

В этой программе искомое число нужно сравнивать со всеми элементами массива. Если число обнаружено, то цикл надо прервать и выйти из тела. Мы это уже умеем делать с помощью оператора `break`. Если число будет найдено, то пусть программа выводит его индекс и само число. А если в массиве нет такого числа, то пусть выводится минус единица.

Чтобы не загромождать код, не будем число вводить, а просто зададим его константой. Код программы показан в листинге 9.6.

Листинг 9.6. Поиск заданного числа в массиве (версия 1)

```
#include <iostream>
using namespace std;
int main(){
   int numbers[]{45,13,81,7,38,27,7,320,15,31};
   int number = 27;                                  // искомое число
   size_t index = 0;
   while(index < size(numbers)){
     if(numbers[index] == number) break;             // число найдено
     ++index;                                        // иначе продолжаем
```

```
    }
    if(index < size(numbers))                          // если найдено
        cout << index << ' ' << numbers[index] << endl;
    else
        cout << -1 << endl;
    return 0;
}
```

Обратите внимание на условный оператор после цикла. Это обязательная проверка. Если искомое число отсутствует в массиве, то после выхода из цикла `index` равен размеру массива. Если же число в массиве найдено, то поиск не дошел до конца массива и `index` меньше `size(numbers)`.

Для проверки этой программы нужно составить таблицу не менее чем из четырех тест-кейсов. Необходимо проверить, найдет ли программа первое число в массиве, последнее число в массиве и какое-нибудь число внутри массива. Четвертый тест должен проверять поведение программы при отсутствии искомого числа в массиве. Таким образом, составляем табл. 9.1.

Запустив программу четыре раза (меняя в коде искомое число), убеждаемся, что программа работает правильно. Заметим, однако, что при наличии в массиве нескольких элементов с искомым числом программа, естественно, находит первый из них, самый левый.

Таблица 9.1. *Тест-кейсы для программы поиска заданного числа в массиве*

Число	Индекс элемента	Элемент массива
45	0	45
31	9	31
27	5	27
12	-1	

При внимательном чтении кода возникает желание кое-что в нем поправить. Например, избавиться от оператора `break`. Решается эта проблема довольно просто: надо удалить оператор `if` и прописать его условие непосредственно в условии цикла. Тогда тело цикла будет содержать только оператор изменения индекса. Полный код новой программы представлен в листинге 9.7.

Листинг 9.7. Поиск заданного числа в массиве (версия 2)

```
#include <iostream>
using namespace std;
int main(){
    int numbers[]{45,13,81,7,38,27,7,320,15,31};
    int number = 31;                                   // искомое число
    size_t index = 0;
```

```
    while((index < size(numbers))&&(numbers[index]!= number)) ++index;
    if(index < size(numbers))                    // найдено
         cout << index << ' ' << numbers[index] << endl;
    else cout << -1 << endl;
    return 0;
}
```

В условии цикла написано логическое выражение, в котором проверяются оба условия окончания цикла. Первое условие всегда истинно, пока не достигнут конец массива, и проверяется второе условие. Если число найдено, то второе условие становится ложным, и цикл прекращается. Если же искомого числа в массиве нет, то в конце концов становится ложным первое условие, и цикл опять же прекращается.

Запустив эту программу четыре раза с теми же значениями искомого числа (см. табл. 9.1), убеждаемся, что программа работает точно так же, как и программа в листинге 9.6.

Задачи и упражнения

Дан массив целых чисел. Все задания реализовать в двух вариантах:

- со встроенным массивом;
- с array-массивом.

1. Найти сумму индексов элементов, значения которых больше заданного числа B.
2. Что больше: сумма четных чисел или сумма нечетных?
3. Каких чисел больше: четных или нечетных?
4. Найти индекс наибольшего отрицательного числа (если в массиве нет отрицательных чисел, то сообщить об этом).
5. Написать программу, в которой первый элемент массива обменивается с последним, второй — с предпоследним и т. д.
6. Реализовать сортировку выбором: находим наименьший элемент в массиве и меняем его с первым; затем находим наименьший элемент, начиная со второго, и меняем со вторым и т. д.
7. Проверить, является ли массив упорядоченным по неубыванию (каждое следующее число больше или равно предыдущему).

- Урок 10 -

Создаем новые типы

Мы начали изучение агрегатов данных на примере массивов (см. *урок 9*). В массивах группы данных однородны (одного типа), и они пронумерованы (индексированы). Однако в программах возникает потребность группировать группы разнородных данных в единое целое. Например, в отделе кадров для каждого работника имеется карточка, в которой записаны самые разные данные: фамилия, имя, отчество, день рождения и место рождения, место работы, должность и т. д.

Естественно, конструкция, с помощью которой можно группировать объекты в единое целое, присутствует в языках программирования. Эта конструкция в разных языках называется по-разному. Например, в языке Pascal такая конструкция называется *запись* (record). В C++ (по наследству от C) принято название *структура* (structure). Напомню, что эта конструкция является одним из составных типов данных (агрегатов). Кроме того, средства для компоновки разнородных данных в один объект реализованы в стандартной библиотеке STL. Они называются пары (pair) и кортежи (tuple).

Но в языке C++ структура является классом. Поэтому мы начнем знакомиться с основами объектно-ориентированного программирования.

Структуры (классы)

В C++ структуры и классы отличаются только одним свойством: доступом к элементам. Поэтому мы пока будем рассматривать только структуры, а в отличиях доступа разберемся чуть позже. Но нужно выяснить, зачем такие конструкции вообще нужны.

Зачем нужны структуры (классы)

Одна причина выше уже приведена: часто требуется группировать данные в единое целое. Именно так использовались структуры в ранних языках COBOL, PL-1, C и др. Однако в современных языках программирования эти конструкции почти всегда применяются для создания новых типов данных. Таких типов, которых в языке программирования изначально нет.

Конечно, новые типы возникают не потому, что программисту так просто захотелось. Типы рождаются на основе анализа задачи. Например, если нам потребуется создать тренажер для обучения действиям с обыкновенными дробями, то естественно определить новый тип данных `fraction` (дробь). Очевидно, что объект типа `fraction` должен содержать пару целых чисел: числитель и знаменатель. И должны быть определены операции с объектами типа `fraction` по правилам арифметики обыкновенных дробей.

Если задача заключается в разработке системы, обрабатывающей денежные суммы, то начинать надо с определения нового типа данных `money` (деньги). Хотя денежные суммы могут быть представлены единственным числом (например, типа `float`), с деньгами нельзя выполнять некоторые операции, которые разрешается выполнять с числами.

Деньги можно складывать и вычитать. Но умножать деньги на деньги нельзя. Можно умножать деньги на некоторое число — коэффициент перевода в другую валюту. Если же мы поделим одну сумму на другую, то результатом будет некоторое число, а совсем не денежная сумма. Таким образом, определение нового типа данных требует определения операций с объектами этого типа.

Определив новый тип данных с помощью соответствующих конструкций, мы расширяем *систему типов* языка. Определив новый тип один раз, мы потом можем его использовать во множестве программ аналогично стандартным функциям. Стандартная библиотека STL — это и есть расширение системы типов C++ (например, тип `string`).

Определение структуры (класса)

Структуры и *классы* — это объединение в единое целое группы нескольких разнородных (или однородных) данных, каждое из которых имеет свое имя.

Например, можно объединить в структуру отдельные элементы даты: год, месяц и день. На языке C++ это можно сделать так:

```
struct Date { unsigned int year, month, day; };
```

Или так:

```
struct Date {
   uint16_t year;
   uint8_t month;
   uint8_t day;
};
```

Будем говорить, что объявление структуры состоит из двух частей: заголовка с именем и тела в фигурных скобках. Имя `Date` называется *тегом структуры*. Объявления объектов в теле структуры, начиная со стандарта C++17, практически ничем не отличаются от объявлений переменных. Эти переменные внутри структуры называют *полями*. Имена полей, естественно, должны отличаться друг от друга и от тега структуры.

Типы полей могут быть любыми, лишь бы имена типов были известны в точке объявления структуры. Например, в некоторой задаче требуется работать с персонами (имя, фамилия, день рождения, пол):

```
struct Person {
   string name;                  // имя
   string surname;               // фамилия
   Date birthday;                // день рождения
   char gender;                  // пол
};
```

Для того чтобы была возможность объявить поле `birthday`, структура `Date` должна быть видима в точке объявления структуры `Person`.

> ***На заметку***
>
> Объявление объекта реализованного типа в качестве поля другой структуры (класса) называется *композицией*. Композиция — это один из методов создания новых типов данных.

В данном случае все поля заданы в отдельных объявлениях, хотя можно было бы все строковые поля указать в одном объявлении. Конечно, чтобы иметь возможность указывать тип `string`, нужно подключить библиотеку `<string>`.

Обратите внимание: после закрывающей скобки блока объявлений должна стоять точка с запятой. Ее отсутствие приводит к ошибке компиляции.

Структуру можно объявить с инициализацией полей, например:

```
struct Date {
   uint16_t year = 2024;
   uint8_t month = 5;
   uint8_t day {1};
};
```

Эта возможность появилась в стандарте C++11. Это уже совсем похоже на объявление переменных. Причем можно применять как инициализацию присваиванием, так и универсальную инициализацию в фигурных скобках. Понятно, что разрешение инициализации полей означает, что допускается точно так же объявлять поля-константы.

Кстати, разрешается объявлять структуры совсем без полей, например:

```
struct Empty {};
```

Такая структура называется пустой. Полезность такой конструкции станет понятна при изучении *исключений*.

Б. Страуструп постановил, что «структура является классом». Поэтому эти объявления структур являются определениями новых типов. Идентификатор `Date` вводит новое имя типа. Аналогично имя `Person` тоже вводит новый тип в точке объявления структуры. Основной принцип, сформулированный Б. Страуструпом, состоит в том, что новый тип должен иметь такие же «права и обязанности», как и встроенные типы:

- объекты нового типа можно включать в структуры (см. выше);

- после объявления типа можно объявлять локальные и глобальные объекты этого типа;
- объекты нового типа можно инициализировать теми же способами, что и объекты встроенных типов;
- объект нового типа можно присваивать другому объекту нового типа;
- можно объявить массив объектов нового типа;
- объекты нового типа можно передавать в качестве параметров в функции любыми разрешенными способами и возвращать как результат.

Я здесь перечислил только то, что мы уже изучили в предыдущих уроках. По мере изучения новых свойств и конструкций языка будем выяснять, какими еще правами обладает новый тип.

Объявление и инициализация объектов

После таких определений новых типов разрешается объявлять переменные несколькими разными способами, например:

```
Date today;                          // без инициализации
Date d1 = { 2021, 6, 28 };           // поэлементная инициализация
Date d = { 2000, 1 };                // неполная инициализация
Date d2 = d1;                        // инициализация другим объектом
Date d3(d2);                         // инициализация другим объектом
```

Отмечу, что при неполной инициализации (переменная `d`) компилятор выдает предупреждения об отсутствии инициализирующего выражения.

Естественно, можно применять и универсальную инициализацию:

```
Date d1 { 2023, 7, 2 };              // поэлементная инициализация
Date d { 2000, 1 };                  // неполная инициализация
Date d2 {d1};                        // инициализация другим объектом
Date td{};                           // без инициализации
```

Последний вариант не обнуляет поля, если они были инициализированы непосредственно в структуре.

Разрешается объявлять объекты в старом стиле, который был в C, например:

```
struct Date tomorrow;
```

Понятно, что из-за обратной совместимости C++ с языком C такой код будет работать, однако в новых программах так писать не следует.

Можно совместить определение типа, объявление объекта и инициализацию:

```
struct Date { unsigned int year, month, day; } date { 2003, 2, 2 };
```

Здесь объявлен объект `date` типа `Date`. После такого объявления тоже можно объявлять переменные типа `Date`. Но если вам не требуется в дальнейшем новый тип, то его можно не писать:

```
struct { unsigned int year, month, day; } date { 2013, 7, 24 };
```

Однако в C++ такой подход применяется очень редко. Стандартным стилем написания кода в C++ является объявление структуры с тегом, а затем объявление объектов нового типа.

Самое интересное, что объявлять объекты нового типа можно с помощью `auto`. Но в таком объявлении справа должен стоять литерал структурного типа. Стандарт C++17 позволяет это делать:

```
auto ddd = Date{2024,6,12};              // литерал типа Date
const auto t = Person{"Иван", "Иванов", Date{2000,1,1}, 'м'};
auto w = Person{"Иванна", "Иванова", ddd, 'ж'};
```

Таким образом, структурный литерал начинается с тега структуры и за ним в фигурных скобках задаются значения полей в том же порядке, как они были объявлены в структуре. До стандарта C++17 для подобной инициализации требовалось написать *конструктор* (инициализации) в структуре (см. ниже).

Вполне допускается объявлять объекты-константы структурного типа. В этом случае инициализация обязательна, например:

```
const Date cd{1941,06,22};
constexpr Date ced {};
```

В объявлении константы `ced` указана «пустая» инициализация, которая реально никак не изменяет поля. Однако при ее отсутствии, естественно, возникает ошибка трансляции, т. к. инициализация констант обязательна.

В C++17 появился механизм structured binding, что переводится буквально как «структурное связывание». Однако будет гораздо понятнее, если назвать это свойство как «декомпозиция при объявлении». Декомпозиция — это разделение структуры на отдельные элементы (поля). Например, можно декомпозировать структуру `Person`:

```
auto [name, surname, birthday, gender] = w;     // объект типа Person
```

Идентификаторы в квадратных скобках являются скалярными переменными и получают типы полей структуры объекта `w` (например, переменная `birthday` получает тип `Date`). Переменным присваиваются текущие значения полей объекта `w` (`birthday` получает значение `{2024,6,12}`). Имена, конечно, можно выбирать любые, допустимые в точке этого объявления, они не обязательно должны повторять имена полей.

> ***На заметку***
>
> Декомпозиция работает в любых объявлениях. В том числе, например, и в объявлении переменных, локальных в пределах оператора.

Нужно упомянуть еще об одном: структуры могут быть объявлены как глобально, так и локально. И точно так же, как любые другие имена, внешний идентификатор типа может быть скрыт внутренним (вспомним сокрытие имен переменных и имен функций в прототипах). Такое сокрытие имен типов влияет на объявление объектов. Напишем простую программу, в которой демонстрируется эта особенность. Код показан в листинге 10.1.

Листинг 10.1. Сокрытие имен типов

```
#include <iostream>
using namespace std;
struct Date { unsigned year {2024}, month {2}, day {24}; };
int main(){
   struct Date { unsigned year, month; }; // локальный тип Date
   Date today;                            // локальный тип
   Date d2 {2024, 4};                     // локальный тип
   ::Date d1{2022,2,24};                  // глобальный тип
   ::Date d5 = d1;                        // глобальный тип
   Date d6(today);                        // локальный тип
   ::Date d4{d5};                         // глобальный тип
   return 0;
}
```

Локальное объявление структуры с тегом `Date` скрывает объявление такой же структуры выше функции `main`. Несмотря на то, что имена внешнего и внутреннего типов совпадают, типы являются разными. Объекты этих типов нельзя присваивать друг другу, т. к. возникает ошибка компиляции. Как и при сокрытии имен переменных, для использования внешнего имени после объявления внутреннего нужно применять операцию доступа `::`.

Структура является классом, поэтому структура, объявленная внутри функции, называется *локальным* классом.

Операции с объектами структурного типа

Для доступа к отдельному полю объекта нового типа применяется операция «точка» — *селектор* поля структуры. Например, чтобы использовать в коде отдельные поля переменной `d`, мы должны писать `d.year`, `d.month`, `d.day`.

В отличие от массивов, объекты нового типа можно присваивать друг другу:

```
today = d2;
```

«Под капотом» происходит поэлементное присваивание полей:

```
today.year  = d2.year;
today.month = d2.month;
today.day   = d2.day;
```

Все остальные операции выполняются только поэлементно. Например, ввод и вывод данных:

```
cout << "Задайте дату (год месяц день):";
cin >> d1.year >> d1.month >> d1.day;
cout << d1.year << '.' << d1.month << '.' << d1.day;
```

Однако все необходимые операции можно реализовать, т. к. структура является классом. Мы научимся это делать чуть позже.

Размеры структур

Для всех встроенных типов и массивов мы определяли размеры с помощью операции `sizeof`. Выясним, сколько памяти выделяется для структуры по умолчанию. Код программы приведен в листинге 10.2.

Листинг 10.2. Размеры структур

```
#include <iostream>
using namespace std;
int main()
{  struct AAA { char ch; double b; };
   struct BBB { double b; char ch; };
   struct CCC { int b; char ch; };
   struct DDD { short b; char ch; };
   cout << "AAA " << sizeof(AAA) << endl;
   cout << "BBB " << sizeof(BBB) << endl;
   cout << "CCC " << sizeof(CCC) << endl;
   cout << "DDD " << sizeof(DDD) << endl;
   return 0;
}
```

При выполнении данная программа выводит на экран:

```
AAA 16
BBB 16
CCC 8
DDD 4
```

Такие цифры означают, что по умолчанию независимо от размеров полей структуры для объектов нового типа выделяется количество байтов, кратное наибольшему размеру. В примере получается для структур `AAA` и `BBB` размер `16`, т. к. размер `double` равен `8`. А для структуры `CCC` выделено `8` байт памяти, т. к. `int` занимает `4` байта. Соответственно, в структуре `DDD` поле типа `short` занимает 2 байта, и вся структура — 4 байта.

Такие размеры определяются байтовой организацией памяти, и это свойство называется *выравниванием*. Более того, в стандарте C++ указаны *требования* к выравниванию объектов. Размер `double` равен 8 байт, и выравнивание тоже должно быть равно 8. Для поля меньшего размера в той же структуре будет выделено тоже 8 байт. Но в эти 8 байт могут поместиться два или более полей, например:

```
struct FFF { double a; int b; char ch; };
```

Размер этой структуры тоже равен 16 байт, т. к. `int` и `char` занимают совместно менее 8 байт.

Однако размеры структуры по умолчанию зависят не только от размеров полей, но даже от порядка полей в структуре. Изменим порядок полей в структуре `FFF`, поставив на первое место `char ch`:

```
struct FFF { char ch; double a; int b; };
```

Размер такой структуры равен 24 байтам (выравнивание полей `char ch` и `int b` тоже равно 8). Поэтому при объявлении структур старайтесь объявлять поля по убыванию размеров: сначала большего размера, затем меньшего.

Таким образом, имеется излишний расход памяти, ведь суммарно поля имеют меньший размер, чем выделяется по умолчанию. Например, сумма размеров полей структуры `FFF` равна 13; в листинге 10.2 суммарные размеры структур составляют 9, 9, 5, 3 соответственно.

Можно заставить компилятор выделять для структур ровно столько памяти, сколько составляет сумма размеров полей. Для этого нужно использовать директиву препроцессора `#pragma`.

Выравнивание по границе байта задается конструкцией `#pragma pack(1)`. Директива работает с момента объявления. Можно задавать выравнивание по границам 2, 4, 8 и 16 байт. Любые нечетные числа или числа больше 16 вызывают предупреждение при компиляции, и директива не выполняется.

Когда нам опять потребуется выравнивание по умолчанию, нужно указать в коде директиву `#pragma pack()` без аргумента.

Модифицируем код в листинге 10.2, добавив директиву `#pragma`:

```
#pragma pack(1)
   struct AAA { char ch; double b; };
   struct BBB { double b; char ch; };
   struct CCC { int b; char ch; };
   struct DDD { short b; char ch; };
```

Получим на выходе числа:

```
AAA 9
BBB 9
CCC 5
DDD 3
```

Однако, если расход памяти не критичен, можно об этом не беспокоиться.

Интерес вызывает вопрос: выделяется ли память для структуры, если в ней нет полей? Добавим в листинг 10.2 объявление пустой структуры:

```
struct Empty {};
```

Вывод размера этой пустой структуры показывает, что ей выделена память 1 (один) байт. Таким образом, мы можем создавать объекты с типом пустой структуры. Причем мы можем объявить несколько пустых структур с разными именами и, соответственно, создавать объекты разных типов. Как уже было сказано выше, эта интересная особенность будет нам весьма полезна при изучении исключений.

Структуры, массивы, объекты и функции

Структура как конструкция объявления нового типа должна сочетаться и взаимодействовать с другими конструкциями языка C++. Например, требуют рассмотрения вопросы сочетания структур и функций, массивов и структур, классов и структур.

Соответственно и объекты нового типа тоже должны «мирно уживаться» с другими конструкциями языка. Мы умеем объявлять и инициализировать объекты нового типа (в том числе и как поле другой структуры), но нужно еще выяснить их взаимодействие с функциями и с массивами.

Массивы и структуры

Массивы и структуры прекрасно уживаются друг с другом: можно объявлять массив внутри структуры и объявить массив структур. Сначала рассмотрим встроенные массивы. Например, пусть требуется ежедневно обрабатывать набор из 10 измеряемых величин. Тогда нам потребуется такая структура:

```
struct Number {double number[10]; Date date; };
```

Объекты с такой структурой можно инициализировать обычным образом, только надо первым инициализировать массив:

```
Number N = { { 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9, 0},
             { 2023, 3, 2 } };
```

Если измерения выполнялись в течение 100 дней, то в программе можно объявить массив объектов типа `Number`:

```
Number NN[100];
```

Проинициализировать весь массив, конечно, сложно. Однако присвоение значений отдельным элементам массива вполне может потребоваться в программе. Как обычно, такое присвоение выполняется поэлементно:

```
NN[0].number[0] = 1.2;
// . . .
NN[0].number[9] = 5.7;
NN[0].date.year = 2013;
NN[0].date.month = 11;
NN[0].date.day = 22;
```

Объявление встроенного массива внутри структуры позволяет делать нам то, что нельзя делать непосредственно с массивами — присваивать один массив другому. Пусть у нас объявлена структура:

```
struct Array { double a[10]; };
Array a = { {1, 2, 3, 4, 5, 6, 7, 8, 9, 0 } };
```

При этом можно не указывать внутренние фигурные скобки:

```
Array a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };     // это легально
Array b = a;                                     // инициализация другим массивом
Array c {b};                                     // универсальная инициализация
Array d;
d = a;                                           // присваивание объектов
```

Последняя строка приводит к тому, что поле-массив объекта `a` копируется в поле-массив объекта `d` без написания оператора цикла.

Естественно, в качестве поля структуры можно указывать и массив `array` из стандартной библиотеки. Тогда структура `Number` выглядит так:

```
struct Number {array<double,10> number; Date date; };
```

Аналогично тег структуры можно задать в качестве типа в массиве:

```
array<Number,100> NN;
```

Присваивание элементов при таком объявлении не отличается от того, что показано выше.

Структуры, объекты и функции

При объявлении структуры объявляется новый тип объектов, практически равноправный со встроенными типами. Поэтому объекты структурного типа можно передавать в функцию как параметры любым способом. Разрешается возвращать такие объекты как результат. Код показан в листинге 10.3.

Листинг 10.3. Передача структурного параметра по значению

```
#include <iostream>
using namespace std;
struct Old { int a; float b; };          // объявление типа Old
Old g(Old p)                             // функция получает и возвращает объект
{  p.a = 6; p.b = 7;                     // изменения только внутри функции
   return p;                             // возврат объекта в качестве значения
}
int main ()
{  Old y{ 1,2 }, x, z{y};      // объявление переменных
   x = g(y);                   // 1 - переменная y не меняется
   cout << "x = {" << x.a <<',' << x.b <<"}\n";
   cout << "y = {" << y.a <<',' << y.b <<"}\n";
   y = g(y);                   // 2 - изменение y при присваивании
   cout << "y = {" << y.a <<',' << y.b <<"}\n";
   auto [f,s] = g(z);          // 3 - декомпозиция возвращаемого объекта
   cout << "z = {" << z.a <<',' << z.b <<"}\n";
   cout << "[f,s] = {" << f <<',' << s <<"}\n";
   return 0;
}
```

В этом примере объявляется структура типа `Old`, и определяется функция, принимающая параметр типа `Old` по значению и возвращающая результат того же типа.

Программа при выполнении выводит в консоль:

```
x = {6,7}
y = {1,2}
y = {6,7}
z = {1,2}
[f,s] = {6,7}
```

Первый вызов `x = g(y)` присваивает переменной `x` значения `x.a = 6` и `x.b = 7`; значение параметра `y` при этом не меняется, поскольку выполнялась передача по значению.

Во втором вызове `y = g(y)` переменная `y` тоже передается по значению, но возвращаемый из функции результат изменяет значения ее компонент.

Наиболее интересен вызов, помеченный комментарием `3`:

```
auto [f,s] = g(z).
```

Так как функция `g` возвращает объект структурного типа `Old`, то его можно декомпозировать, как это было описано выше. Переменные `f` и `s` далее можно использовать как обычные локальные переменные.

Передача по ссылке для объектов новых типов совершенно аналогична передаче встроенных типов, т. е. передаваемый в качестве фактического параметра объект изменяется. Код приведен в листинге 10.4.

Листинг 10.4. Передача структурного параметра по ссылке

```
#include <iostream>
using namespace std;
struct Old { int a; float b; }; // объявление типа
Old g(Old &p)                   // параметр — ссылка на объект
{  p.a = p.b + 6;               // 2+6 = 8 - см. структуру y ниже
   p.b = p.a + 7;               // 8+7 = 15
   return p;
}
int main ()
{  Old y { 1,2 }, x;            // объявление переменных
   x = g(y);                    // переменная y меняется
   cout << "x = {" << x.a <<',' << x.b <<"}\n";
   cout << "y = {" << y.a <<',' << y.b <<"}\n";
   return 0;
}
```

Вызов `x = g(y)` приводит к тому, что значения переменных `x` и `y` становятся одинаковыми. В консоль выводится:

```
x = {8,15}
y = {8,15}
```

Передача параметра по константной ссылке показана в листинге 10.5.

Листинг 10.5. Передача структурного параметра по константной ссылке

```
struct Old { int a; float b;};
Old g (const Old &p)       // внутри функции параметру присваивать нельзя
{  Old r;                  // поэтому объявляется локальная переменная
```

```
    r.a = p.b + 6;
    r.b = p.a + 7;
    return r;
}
```

Функция `main` приведена в листинге 10.3.

Использование структур позволяет нам уменьшать количество параметров в функциях. Например, нам нужно написать функцию для вычисления расстояния между двумя точками на плоскости. Точка представлена парой координат `x` и `y`. Объявим структуру `Point` (точка):

```
struct Point {double x,y;};
```

Тогда функция может быть реализована так:

```
double distance(Point a, Point b){
    rx = a.x - b.x;
    ry = a.y - b.y;
    return sqrt(rx*rx + ry*ry);
}
```

Без использования нового типа `Point` пришлось бы в заготовке писать четыре параметра:

```
double distance(double ax, double ay, double bx, double by)
```

Этот заголовок не только более длинный, но и менее понятный. Объявление типа `Point` позволяет уже по заголовку понять назначение функции. Поэтому я рекомендую применять создание структурных типов в подобных случаях.

Классы и структуры

Разница структура и класса состоит только в одном: элементы структуры открыты и доступны, а элементы класса закрыты и недоступны. Чтобы разобраться в этом вопросе, давайте напишем класс `Date` и объявим несколько объектов. Для этого достаточно заменить слово `struct` на слово `class`. Код примера представлен в листинге 10.6.

Листинг 10.6. Объявление класса вместо структуры

```
#include <iostream>
using namespace std;
class Date { unsigned year {2024}, month {2}, day {24}; };
int main(){
    Date today;                                  // работает
//    Date d1 = {2022,2,24};                     // ошибка
//    Date d2 {2022,2,24};                       // ошибка
//    Date d3 = Date{2022,2,24};                 // ошибка
    Date d5 = today;                             // работает
```

```
    Date d6(today);                    // работает
    Date d4{today};                    // работает
    d4 = d5;                           // работает
    return 0;
}
```

Компиляция этого кода показывает:

1. Инициализация полей в непосредственно классе — работает.
2. Объявление объекта без инициализации — работает.
3. Объявление объекта с инициализацией другим объектом — работает в любых форматах.
4. Присваивание объектов — работает.
5. Поэлементная инициализация полей в объявлении объекта вызывает ошибки трансляции.

Попытаемся присвоить полю объекта значение, например:

```
d4.year = 2022;
```

Этот оператор тоже приводит к ошибке трансляции:

```
error: 'unsigned int Date::year' is private within this context
```

Сообщение говорит о том, что поле `year` является собственным (личным, приватным) полем класса `Date` и недоступно в данном контексте (как будто оно не было объявлено).

Эти ошибки компиляции — следствие закрытости класса от постороннего доступа. Поля класса похожи на локальные переменные, которые невидимы за пределами тела. По умолчанию поля доступны только внутри фигурных скобок класса.

Закрытость и недоступность элементов класса — это реализация принципа *инкапсуляции*, который является одним из главных принципов объектно-ориентированного программирования. Слово «инкапсуляция» в буквальном смысле означает «поместить в капсулу». Капсулой является тело класса в фигурных скобках.

Мы уже сталкивались с понятием инкапсуляции: локальные переменные, объявленные в функции, недоступны извне. Однако в отличие от функций, доступ к элементам класса сделан управляемым. Для этого в языке C++ имеются два спецификатора доступа `public` и `private`. Эти английские слова понятны даже без перевода.

Спецификатор `public` позволяет сделать доступным (публичным) любой элемент класса или сразу группу элементов класса. Например, можно сделать класс `Date` полностью открытым, как структуру:

```
class Date {
    public:
    unsigned year {2024}, month {2}, day {24};
};
```

Или можно каждый элемент отдельно объявить публичным:

```
class Date {
    public: unsigned year {2024};
    public: unsigned month {2};
    public: unsigned day {24};
};
```

После такого объявления класса все объявления объектов, которые ранее приводили к ошибкам трансляции, не вызывают у компилятора протестов. С таким классом можно делать все, что мы делали выше со структурами.

Действие спецификатора доступа `private` противоположно: доступ извне к private-элементам класса становится невозможен. Например, мы можем сделать структуру похожей на класс:

```
struct Date {
    private:
       unsigned year {2024}, month {2}, day {24};
};
```

При таком объявлении структуры тоже будет невозможно объявить объекты типа `Date` с инициализацией, т. к. доступ к полям закрыт.

Конструкторы

Для осуществления поэлементной инициализации полей класса нужно написать *конструктор* инициализации, о котором уже было упомянуто выше. Сначала напишем конструктор, а потом разберемся, почему он написан именно так.

Код класса `Date` с конструктором представлен в листинге 10.7.

Листинг 10.7. Класс с конструктором инициализации

```
class Date {
    // поля закрыты по умолчанию
    unsigned year {2024}, month {2}, unsigned day {24};
public:                          // конструктор открыт
    // конструктор инициализации открыт
    Date(unsigned y, unsigned m, unsigned d){
        year = y; month = m; day = d;
    }
};
```

В этом коде мы видим следующее:

1. Конструктор — это специальная функция с именем класса.
2. Конструктор, как и функция, имеет параметры и тело.
3. У конструктора отсутствует тип возвращаемого значения, даже `void`.
4. Конструктор написан непосредственно в классе (можно и вне класса).

Кстати, параметры в конструкторе инициализации можно задавать в любой допустимой форме.

Конструктор должен быть открыт (`public`). Иначе при объявлении объекта получим сообщение об ошибке, описанное выше. Наличие конструктора в классе позволяет объявлять объекты с инициализацией полей в любой форме:

```
Date d1 = {2022,2,24};                  // универсальная инициализация
Date d2 {2022,2,24};                    // универсальная инициализация
Date d3 = Date{2022,2,24};              // универсальная инициализация
Date dd(2006,06,06);                    // старый формат
Date ddd = Date(2021,12,14);            // старый формат
```

Объявление объекта (с инициализацией) — это вызов в той или иной форме конструктора инициализации.

По-прежнему работает инициализация объектом того же типа:

```
Date d5 = today;                        // старый формат
Date d6(today);                         // старый формат
Date d4{today};
```

Такие объявления намекают на то, что должен существовать конструктор с параметром-объектом своего класса. Действительно, такой конструктор для класса `Date` создается без нашего участия автоматически. Если мы заглянем в стандарт, то увидим, что прототип этого конструктора такой:

```
Date(const Date &t);
```

Он называется конструктором *копирования*, и его работа заключается в том, что значения полей объекта-аргумента присваиваются полям нового объекта (как это делается в операции присваивания, см. выше).

Однако неожиданно выясняется, что объявление объекта без инициализации:

```
Date today;
```

вызывает ошибку трансляции. Пока в классе не было конструктора, такое объявление не вызывало проблем. Это означает только одно: если в классе не определен конструктор инициализации с параметрами, то автоматически создается конструктор, с помощью которого можно объявлять объекты без инициализации. Он называется конструктором *по умолчанию*. Аргументов у него нет, поэтому можно его называть конструктором без аргументов.

Явное определение конструктора инициализации отменяет автоматическое создание конструктора без аргументов. Можно написать этот конструктор самостоятельно. Однако в языке есть средство, которое позволяет указать компилятору, что нам нужен этот конструктор и надо его создать. Для этого надо написать в классе объявление:

```
Date() = default;
```

После этого объявление объектов без инициализации проблем не вызывает.

Таким образом, выясняется, что в классе обычно бывает три конструктора[1]:

1. Конструктор без аргументов (по умолчанию).
2. Конструктор инициализации.
3. Конструктор копирования.

Здесь мы снова сталкиваемся с *перегрузкой* функций, о которой уже было упомянуто ранее (см. *урок 7*).

Методы

Кроме конструкторов в классе можно писать и другие функции. Как мы уже знаем (см. *урок 3*), эти функции называются *методами*. Хотя Б. Страуструп называет их функциями-членами, в других языках программирования эти функции называются методами, и мы будем использовать этот термин.

Напишем метод для вывода полей в консоль. Код класса с этим методом представлен в листинге 10.8.

Листинг 10.8. Класс с методом *print*

```
class Date {
public:
    Date() = default;
    Date(unsigned y, unsigned m, unsigned d){
        year = y; month = m; day = d;
    }
    void print() const {                          // константный метод
       cout << year << '.' << month << '.' << day << '\n';
    }
private:
    unsigned year{2024}, month{5}, day{1};
};
```

Я написал тривиальный метод вывода с именем `print` только для того, чтобы показать некоторые особенности написания методов в классе. Во-первых, поля перенесены вниз, и к ним явно закрыт доступ спецификатором `private`. Но это запрет для доступа извне, а конструкторам и методам класса поля доступны всегда, независимо от того, где и с каким спецификатором они прописаны в классе. Во-вторых, в заголовке метода `print` после списка параметров написано слово `const`. Это слово говорит о том, что этот метод не изменяет поля класса, и это проверяется компилятором.

---

[1] Существует еще один вид конструктора — конструктор переноса, но он нам не понадобится.

Вызов метода осуществляется с помощью селектора (точка) — мы это уже делали для объектов типа `string` (см. *урок 3*):

```
today.print();
```

Все то же самое, что мы сейчас проделали с классом, можно делать и со структурой, с учетом того, что можно не указывать спецификатор `public`.

Задачи и упражнения

1. Объявить дробь, написать функцию сложения двух дробей.
2. Объявить точку, объявить отрезок (использовать прием композиции), написать функцию вычисления длины отрезка.
3. Объявить дробь, написать метод сокращения дроби.
4. Объявить точку, написать методы, возвращающие:
   - расстояние от текущей точки до начала координат;
   - значение 0, если точка лежит в начале координат; 1, 2, 3 или 4, если точка лежит в соответствующей четверти; 5, если точка лежит на оси 0x, и 6 — если на оси 0y.
5. Объявить дробь, написать функции:
   - сравнения двух дробей на равенство (возвращает `true`, если дроби равны);
   - сравнения двух дробей на больше/меньше (возвращает `true`, если одна дробь больше другой).

- Урок 11 -

Знакомство с указателями

Одним из самых трудных понятий в языке C++ для начинающих считаются указатели. Не разобравшись с ними, вы никогда не сможете программировать на C++ серьезные задачи.

С помощью указателей реализуется одна из «великих идей C++» — косвенные обращения. *Косвенное обращение* — это ситуация (в программе), когда доступ к некоторому объекту осуществляется с помощью другого объекта. Указатели являются этими промежуточными объектами. Как увидим далее, косвенность повышает гибкость кода и способствует инкапсуляции.

В языке C++ существует два вида указателей: встроенные и реализованные. Встроенный указатель — это конструкция языка C++, которая досталась по наследству от C. Они являются настолько фундаментальной конструкцией языка, что без них не было бы ни C, ни C++. Реализованные указатели, которые обычно называют интеллектуальными (хотя никакого интеллекта в реализации, естественно, нет), определены в стандартной библиотеке.

В библиотеке реализован и другой вид объектов, обеспечивающий косвенное обращение, — *итераторы*. Итераторы реализованы для всех стандартных контейнеров (в том числе для массивов `array`) и для строк. С итераторами мы тоже познакомимся и выясним их связь со встроенными указателями.

Что такое указатели

Память компьютера является массивом. Этот факт признали разработчики стандарта C++ и внесли в стандарт *модель памяти*. По стандарту память в C++ представляет собой массив *байтов*. Количество элементов стандартом не определяется, а размер одного байта должен быть достаточен, чтобы в нем помещался символ типа `char`.

Память в компьютере с процессором Intel — это тоже массив байтов. В массиве каждый элемент имеет индекс. Байты памяти тоже пронумерованы, начиная с нуля[1]

---

[1] Во всех современных языках программирования индексация массивов начинается с нуля по аналогии с адресацией памяти.

и до n – 1 (рис. 11.1). Этот номер (индекс в массиве памяти) называется *адресом*. Таким образом, адрес — это фактически беззнаковое целое число.

Когда программа попадает в память компьютера для выполнения (ее загружает операционная система), каждая переменная и каждая функция размещаются по некоторому адресу. Каков реально этот адрес — нам знать совсем не обязательно, важно понимать, что каждое имя в программе на самом деле является адресом памяти. Говорят, что по этому адресу в памяти хранятся значения соответствующей переменной.



Рис. 11.1. Модель памяти C++

Однако понятно, что по некоторому адресу может храниться не значение, а адрес другой переменной. И чтобы добраться до значения, нужно извлечь адрес этой другой переменной из памяти по первому адресу.

Язык C++ разрешает объявлять переменные, содержащие адреса других переменных. Такие переменные называются *указателями*, а доступ к значению с помощью указателя называется *косвенным обращением*. Опять же, какие это адреса — нам знать совсем не обязательно. Мы только должны указать, что нам нужен адрес именно этой переменной, а не другой.

Начнем с простейших объявлений встроенных указателей:

```
int *p;    // переменная p может содержать адрес объекта типа int
```

Я поставил звездочку рядом с именем `p`. Можно считать, что этот оператор является объявлением целой переменной `*p`. Язык C++ разрешает печатать `*p` в любом месте кода, где допускается указывать целую переменную.

С другой стороны, можно напечатать звездочку ближе к типу:

```
int* p;    // переменная p может содержать адрес объекта типа int
```

Этот оператор можно понимать именно как объявление указателя на объекты типа `int`. Например, можно объявить в одном операторе две переменные:

```
int* p, ip;              // ВНИМАНИЕ!
```

Неопытный программист намеревался объявить два указателя на целое. Однако здесь его поджидает неприятный сюрприз! Переменная `p` является указателем на объекты целого типа, однако переменная `ip` — не указатель, а обычная переменная типа `int`!

Поэтому я предпочитаю объявлять указатели первым способом, «прижимая» звездочку к имени переменной. Кроме того, многолетняя практика убедила меня, что лучше объявлять любые переменные в отдельных операторах: тогда каждую из переменных можно снабдить своим комментарием.

Две самые важные операции

Чтобы присвоить указателю адрес переменной, в языке реализована *операция получения адреса*, которая обозначается символом & (амперсанд):

```
int a;          // целая переменная
a = 3;          // присвоение значения
int *pa;        // указатель на объект целого типа
pa = &a;        // присвоение адреса
```

После выполнения этой операции присваивания переменная-указатель pa указывает на переменную a. Эта ситуация изображена на рис. 11.2.

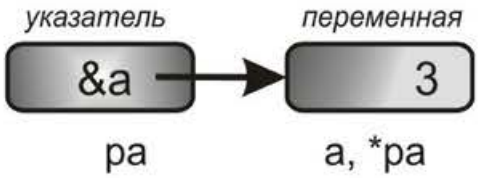


Рис. 11.2. Указатель pa указывает на целую переменную a

Большинство проблем с указателями у программиста возникает из-за одной простой вещи — непонимания, что в объявлении:

```
int *pa;
```

переменная pa является указателем и хранит адрес некоторой переменной целого типа, а конструкция *pa — это целая переменная.

Вообще-то звездочка при имени переменной — это *операция разыменования*, которая используется для доступа к объекту, на который указывает указатель. Звездочка реализует косвенное обращение, о котором мы говорили выше. Мы наблюдаем здесь *перегрузку операций*: звездочкой обозначается и другая операция — бинарная операция умножения.

Таким образом, операции & (получения адреса) и * (разыменования) являются в некотором смысле парными. Если в переменной a хранится число 3, то при выводе в консоль выражения *(&a) появится число 3. Это выражение можно писать и без скобок: *&a. С другой стороны, если в указателе pa содержится некоторый адрес, то этот же адрес выдается в консоль для выражения &(*pa), которое тоже можно писать без скобок: &*pa.

Продемонстрируем эти операции на примере простой программы (листинг 11.1).

Листинг 11.1. Простейшие операции с указателями

```
#include <iostream>
using namespace std;
int main()
{       int a = 5, b = 0;                    // - 1
        int *pa; pa = &a;                    // - 2
        cout << "a=" << a << endl;           // - 3
```

```
        cout << "*pa=" << *pa << endl;          // - 4
        cout << "pa=" << pa << endl;            // - 4.1
        b = *pa-3;                              // - 5
        cout << "b=" << b << endl;              // - 6
        if (a == *pa) *pa = b;                  // - 7
        cout << "a=" << a << endl;              // - 8
        cout << "*pa=" << *pa << endl;          // - 9
        b = 6; pa = &b;                         // - 10
        cout << "b=" << b << endl;              // - 11
        cout << "*pa=" << *pa << endl;          // - 12
        cout << "pa=" << pa << endl;            // - 12.1
        cout << "a=" << a << endl;              // - 13
        return 0;
}
```

Эта программа выведет на экран следующее:

```
a=5             -- вывод оператора 3
*pa=5           -- вывод оператора 4
pa=0x6dfe14     -- вывод оператора 4.1
b=2             -- вывод оператора 6
a=2             -- вывод оператора 8
*pa=2           -- вывод оператора 9
b=6             -- вывод оператора 11
*pa=6           -- вывод оператора 12
pa=0x6dfe10     -- вывод оператора 12.1
a=2             -- вывод оператора 13
```

Операторы в строках `1` и `2` — это объявления переменных с инициализацией. Указатель `pa` получает адрес переменной `a`.

Затем операторы `3` и `4` показывают, что `a` и `*pa` имеют одно и то же значение.

Оператор `4.1` выводит адрес переменной `a`, присвоенный указателю `pa`. Адрес выводится в *шестнадцатеричном* виде по умолчанию.

В следующем операторе (строка `5`) вычисляется значение переменной `b`. Выражение `*pa-3` не изменяет указатель `pa`. Указатель разыменуется до значения `5` переменной `a` (на нее указывает `pa`), вычитается `3`, и результат `2` заносится в переменную `b`. Оператор в строке `6` выводит это значение.

Следующий оператор `if` и операторы вывода в строках `8` и `9` показывают нам неразрывную связь `a` и `*pa`: изменение `*pa` автоматически означает изменение `a`. Оператор в строке `10` *разрывает* эту связь указателя `pa` с переменной `a` и устанавливает новую связь — с переменной `b`. Операторы вывода `11-13` демонстрируют это.

Объявление указателей

Теперь, когда мы немного попробовали указатели «на вкус», углубимся в некоторые детали. Прежде всего отметим, что в языке C++ существует три вида встроен-

ных указателей (хотя в реализации все они представляют одно и то же — переменную, значением которой является адрес памяти):

- типизированный указатель на переменную `тип *имя`. Тип может быть как встроенным, так и определенным программистом;
- бестиповый указатель `void*`;
- указатель на функцию.

Наиболее часто применяются типизированные указатели. С ними можно выполнять некоторые арифметические действия (но не все).

Бестиповый указатель от типизированного отличается тем, что с ним нельзя производить никаких арифметических операций. Бестиповые указатели — это наследство языка C, и в программах на языке C такие указатели применяются для написания универсального кода, который не зависит от типа объектов. Однако мы ими пользоваться с этой целью не будем, т. к. в современном C++ есть более безопасные средства, позволяющие писать универсальные программы.

Указатель на функцию — это вообще особый вид указателя, который применяется исключительно с одной целью — косвенного вызова функции. С таким указателем тоже нельзя выполнять никаких арифметических действий.

Типизированные указатели

Общий вид объявления типизированного указателя выглядит так:

```
тип *имя
```

Как обычно, в одном операторе можно объявлять несколько указателей, можно смешивать объявления указателей и не указателей. Но не забывайте об описанной выше ловушке и всегда пишите имя указателя со звездочкой.

Указатели можно инициализировать адресом другой переменной того же типа или другим указателем, причем в любой форме:

```
int a = 3;              // инициализация целой переменной a
int *pa = &a;           // инициализация указателя pa адресом a
int *h (&a);            // инициализация указателя h адресом a
int *ha{&a};            // универсальная инициализация
int *r = h;             // инициализация другим указателем
int *p(r);              // инициализация другим указателем
int *ra{h};             // универсальная инициализация
```

Указатель `pa` получил значение — адрес переменной `a`. Этот же адрес попал во все остальные указатели. Теперь в программе выражения `*p`, `*ha`, `*ra`, `*h`, `*pa` и `*r` представляют одно и то же значение переменной `a`, на которую указывают все указатели.

Ничто не запрещает нам объявлять указатели на указатели, например:

```
int **p;
```

Значением такой переменной должен быть адрес указателя — это двойное косвенное обращение. Звездочек может быть сколько угодно. Переменные «указатель-на указатель-на…» обычно применяются при работе с динамическими массивами.

Бестиповые указатели и преобразование типов

Мы рассмотрим бестиповый указатель, т. к. его часто можно видеть и в прототипах функций стандартной библиотеки C, унаследованной в C++, и в некоторых конструкциях C++, работающих с указателями.

Бестиповый указатель объявляется аналогично типизированному:

```
void *p;
```

Инициализировать его можно адресом переменной или другим указателем любого типа, например:

```
double d = 3.1;
void *vpa = &d;
void *vp = pa;          // pa - указатель на int (см. выше)
```

А вот инициализировать типизированный указатель бестиповым так просто не получится — требуется явное приведение к типу указателя. Мы уже знаем (см. *урок 4*), что в C++ имеется несколько вариантов явного преобразования типа. По наследству от C остались два:

```
(тип) выражение
тип (выражение)
```

Теперь понятно, что первая, несколько необычная, форма была включена в язык именно для преобразования указателей:

```
double d = 3.1;
void *vpa = &d;                 // преобразований не требуется
double *pd = (double*)vpa;      // требуется явное преобразование
```

Преобразования типизированных указателей из одного типа в другой могут выполняться таким же образом:

```
int a = 6;
double *dp = (double *) &a;     // требуется явное преобразование
int *ip = (int *) dp;           // требуется явное преобразование
```

Однако, как мы помним (см. *урок 4*), в стандарт C++ добавлена новая форма преобразования типов:

```
static_cast <тип> (выражение)
```

Поэтому можно выполнять преобразование типа так:

```
int a = 6;
int *pa = &a;
void *pointer = static_cast <void *> (pa);
```

Однако для типизированных указателей компилятор выдает ошибку:

```
double *dp = static_cast<double *>(&a);        // ошибка трансляции
int *ip = static_cast<int *>(dp);              // ошибка трансляции
```

Бестиповый указатель невозможно разыменовать непосредственно, т. к. объектов типа `void` не существует. Значит, нужно опять делать приведение указателя к целевому типу. Вот как это делается для `double`:

```
double d = 3.1;
void *p = &d;
*((double*)p) = 8.12;
```

Преобразования указателей являются чрезвычайно опасными, поэтому надо их по возможности избегать.

nullptr

Главной причиной, по которой происходит большинство «аварий» программ, является отсутствие адреса в указателе. Если мы не присвоим указателю никакого адреса, то попытки использовать такой указатель приводят, как правило, к непредсказуемому поведению программы.

Поэтому нужно иметь возможность проверить, содержит ли указатель некоторый адрес или он «пустой», т. е. адреса там нет. Создатели языка C решили, что «пустой» указатель должен содержать ноль — в этом случае легко проверить, содержит ли указатель адрес или нет. Проверка делается обычным образом с помощью логического отрицания:

```
double *p = 0;
if (!p)...
```

Это часто помогает уберечь программу от «самоликвидации».

Но ноль — это просто число. И только это число можно присвоить указателю без явного преобразования типа. Однако код с присвоением нуля указателю почти не отличается от кода с инициализацией обычной переменной и легко может «затеряться» среди объявлений других переменных. А последствия отсутствия инициализации могут быть весьма серьезные.

Поэтому в C++11 было принято решение добавить в язык специальный литерал, обозначающий отсутствие адреса. Литерал представлен ключевым словом `nullptr`, и он может быть присвоен указателю любого типа:

```
double *p = nullptr;
```

Такое объявление гораздо информативнее, его и найти в коде гораздо проще. Проверка на «пустоту» выполняется обычным образом:

```
if (!p). . .
```

Очень часто в коде C++ можно увидеть такую инициализацию указателя:

```
double *p = NULL;                    // макрос NULL
```

Это инициализация с помощью *макроса* `NULL`, который был определен еще в C как ноль. В C++11 макрос переопределен как `nullptr`.

Константность

Как и другие переменные, можно объявлять константы-указатели, однако и тут нас поджидают некоторые сюрпризы:

```
int i;
const int ci = 5;
const int *pi = &ci;
```

При такой записи указатель `pi` не является константой — он указывает на константу целого типа. Правильное объявление константы-указателя пишется следующим образом:

```
int *const pi = &ci;            // ci – константа
```

Однако такое объявление не транслируется! В этой записи указатель у нас константный, но указывать должен на переменную. Поэтому транслятор требует вместо адреса константы присвоить адрес переменной.

Вообще имеется четыре варианта объявлений указателя с константностью:

- не константный указатель на не константу `тип *` — можно изменять и указатель, и переменную;
- не константный указатель на константу `const тип *` или `тип const *` — указатель изменять можно, а переменную нельзя;
- константный указатель на не константу `тип * const` — переменную изменять можно, а указатель нельзя;
- константный указатель на константу `const тип const *` или `const тип * const` — нельзя изменять ни переменную, ни указатель.

Наверное, самое частое объявление указателя — это `const char *`. С ним и с тем, на что он указывает, мы разберемся в дальнейшем.

Размеры указателей

И, наконец, выясним, сколько памяти занимают указатели разного типа и влияет ли количество звездочек в объявлении указателя на его размер. Код программы приведен в листинге 11.2.

Листинг 11.2. Размеры указателей

```
#include <iostream>
using namespace std;
int main()
{  cout << sizeof (void *) << endl;
   cout << sizeof (bool *) << endl;
   cout << sizeof (char *) << endl;
// указатели всех остальных типов
   cout << sizeof (long double *) << endl;
```

```
// двойные указатели
   cout << sizeof (void **) << endl;
   cout << sizeof (char **) << endl;
   cout << sizeof (double **) << endl;
   return 0;
}
```

При запуске программа во всех случаях выдает число 8.

Это определяется тем, что сейчас операционные системы — 64-битные. Несколько лет назад, когда системы были 32-битные, размер указателя составлял 4 байта.

Указатели и ссылки

Мы использовали ссылки как способ передачи параметров в функциях. Но ссылки можно объявлять как переменные:

```
int name = 5;
int& new_name = name;
```

Имя объявленной переменной new_name является вторым именем (псевдоним) переменной name. Обращаю внимание: второе объявление не приводит к созданию новой переменной. Оба имени *ссылаются* на одно и то же место в памяти, т. е. имеют один адрес, в чем легко можно убедится:

```
cout << &name << ' ' << &new_name << endl;
```

На моей машине вывелся такой результат (адреса, как обычно, выводятся в шестнадцатеричном виде):

```
0x6dfe14 0x6dfe14
```

Еще одним подтверждением является изменение значения переменной при указании второго имени:

```
++new_name;
cout << name << ' ' << new_name << endl;
```

Выводится пара значений:

```
6 6
```

Изменение значения по имени new_name изменяет значение переменной name.

При объявлении ссылка должна быть инициализирована.

Встроенные массивы и указатели

Указатели тесно связаны со встроенными массивами. Пусть в программе объявлены массив и указатель:

```
int a[10];
int *pa;
```

Тогда присвоить адрес массива переменной-указателю можно одним из следующих совершенно равноправных способов:

```
pa = &a[0];
pa = &a;
pa = a;                    // не требуется ни &, ни приведения типа
```

Последнее присваивание говорит о том, что имя массива является одним из значений указателя того же типа. Ведь для присваивания не требуется ни преобразований типа, ни операции получения адреса. Более того, мы можем применить к имени массива операцию разыменования, например:

```
*a = 126;
```

Этот оператор эквивалентен стандартной записи:

```
a[0] = 126;
```

В том, что имя массива является адресом, можно убедиться, написав оператор вывода в консоль:

```
cout << a << ' ' << pa << endl;
```

В консоли будет выведено два одинаковых шестнадцатеричных значения.

Имя массива можно сравнивать с указателем на равенство или неравенство:

```
if(pa == a)...
```

После того как имя массива присвоено указателю, мы можем обращаться к элементам массива и по его исходному имени, и по имени указателя. Например, вместо `a[0]` можно писать `pa[0]`, вместо `a[i]` — `pa[i]`.

```
for(size_t i{0}; i < size(a); ++i) cout << pa[i] << ' ';
```

Это оператор цикла выведет в консоль массив, адрес которого присвоен указателю `pa`. Однако тут есть свои тонкости. Так как указатель — это все-таки не массив, то нельзя применять к нему функцию `size`: выражение `size(pa)` приводит к ошибке компиляции. Поэтому в операторе написано выражение `size(a)`.

Операции с указателями

С типизированными указателями разрешается выполнять арифметические операции:

1. К указателю можно прибавить целое число с помощью любой операции сложения.
2. От указателя можно отнять целое число с помощью любой операции вычитания.
3. Можно вычислить разность указателей.

Других арифметических операций с указателями выполнять нельзя. Например, указатели нельзя складывать друг с другом, нельзя умножать и делить ни в каких выражениях. Никаких арифметических операций нельзя выполнять с бестиповыми указателями.

Указанные выше операции практически всегда применяются в циклах при обработке массивов. Пусть в программе объявлен массив и указатель на него:

```
double a[] {10,20,30,4,5,6,7,8,9,0};
double *p{a};
```

Указатель p содержит адрес элемента a[0]. Увеличим значение указателя на единицу любым способом:

```
++p;
p += 1;
p = p + 1;
```

Указатель будет показывать на элемент массива a[1]. В этом можно легко убедиться, выведя в консоль значение *p.

Тогда вам должно быть понятно, что выражение pa + i означает адрес i-го элемента массива. Обращаю внимание, что i — это номер элемента массива, а не количество байтов. Компилятор сам правильно вычисляет адрес нужного байта, выполняя умножение i * sizeof(тип) (в данном случае — double).

Можно вывести адреса всех элементов массива в цикле:

```
for(size_t i{0}; i < size(a); ++i) cout << (p + i) << ' ';
```

Так как p = a, то можно писать непосредственно имя массива:

```
for(size_t i{0}; i < size(a); ++i) cout << (a + i) << ' ';
```

Будут выведены те же самые адреса в том же порядке.

Значение i-го элемента можно получить с помощью разыменования:

```
*(p + i)
```

Скобки поставлены, поскольку приоритет операции разыменования * больше приоритета операции сложения +. Эта запись эквивалентна записи p[i] и имеет смысл a[i]. Понятно, что выражение *(p + i) можно использовать и слева, и справа от знака присваивания.

Показанный выше цикл вывода элементов массива можно написать так:

```
for(size_t i{0}; i < size(a); ++i) cout << *(p + i) << ' ';
```

Вместо переменной указателя можно писать непосредственно имя массива:

```
for(size_t i{0}; i < size(a); ++i) cout << *(a + i) << ' ';
```

Замечу, что выражение *p + i имеет совершенно другой смысл: сначала извлекается значение по адресу, хранящемуся в p, и складывается с целым i, т. е. имеем выражение a[0] + i.

Вообще со скобками в выражениях с указателями надо обращаться очень аккуратно. Особенно это касается операций инкремента и декремента. Если указатель p указывает на элемент массива, то операция инкремента p++ или ++p «передвинет» указатель на следующий элемент массива: адрес в указателе увеличивается на sizeof(тип) байтов. Однако в сочетании с операцией разыменования (да еще и со скобками) могут получиться неочевидные результаты. Чтобы разобраться в этих нюансах, напишем небольшую программу, код которой представлен в листинге 11.3.

Листинг 11.3. Операции инкремента с указателями

```
#include <iostream>
using namespace std;
```

```
int main(){
    int a[] {10,20,30,4,5,6,7,8,9,0};
    int *pa{a};
    cout << pa << ' ' << *pa << endl;                          // 1
    cout << ++*pa << ' ' << pa << ';' << *pa << '\n';          // 2
    pa = a; a[0]=10;          // восстановление
    cout << ++(*pa) << ' ' << pa << ';' << *pa << '\n';        // 3
    pa = a; a[0]=10;
    cout << *++pa << ' ' << pa << ';' << *pa << '\n';          // 4
    pa = a; a[0]=10;
    cout << *(++pa) << ' ' << pa << ';' << *pa << '\n';        // 5
    pa = a; a[0]=10;
    cout << *pa++ << ' ' << pa << ';' << *pa << '\n';          // 6
    pa = a; a[0]=10;
    cout << *(pa++) << ' ' << pa << ';' << *pa << '\n';        // 7
    pa = a; a[0]=10;
    cout << (*pa)++ << ' ' << pa << ';' << *pa << '\n';        // 8
   return 0;
}
```

Все операторы вывода прокомментированы номерами. Оператор `// 1` выводит первую строку в консоли: адрес массива и значение элемента `a[0]`. Напоминаю, что адреса выводятся в шестнадцатеричном виде.

В операторе `// 2` выводится значение выражения `++*p`, а затем получившийся адрес после выполнения этого выражения и значение по этому адресу. Выведенные значения говорят нам о том, что это выражение не увеличивает указатель, а изменяет значение по первоначальному адресу. Затем состояние массива и указателя восстанавливается.

Оператор с пометкой `// 3` выводит третью строку, и этот вывод показывает нам, что скобки в выражении `++(*p)` никак не влияют на результаты.

Операторы с пометками `// 4` и `// 5` демонстрируют, что выражения `*++pa` и `*(++pa)` работают одинаково: сначала увеличивается адрес, а потом по этому адресу извлекается значение.

Операторы `// 6` и `// 7` показывают отличие постфиксного инкремента от префиксного: сначала выводится значение `*p` по исходному адресу, потом увеличивается адрес. Оператор `// 7` демонстрирует, что скобки в выражении `*(p++)` не изменяют результат.

И наконец, в операторе с пометкой `// 8` выясняется, что в выражении `(*p)++` скобки приводят к изменению вычислений: сначала выводится исходный результат, адрес не увеличивается, а изменяется значение по этому адресу, т. е. значение `a[0]` увеличивается на единицу.

Запустим программу и получим результаты в консоли (рис. 11.3).

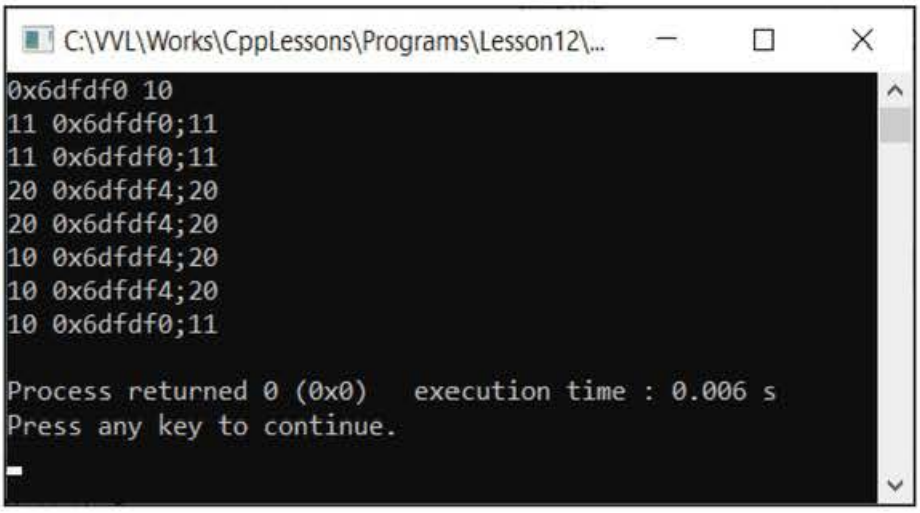


Рис. 11.3. Результат работы программы

Понятно, что то же самое можно проделать с операциями декремента — сделайте это самостоятельно.

Теперь рассмотрим операцию вычитания указателей. Разность указателей — это важная величина. Пусть у нас в программе есть следующие объявления:

```
int a[10];
int *p1 = &a[1];                    // начало
int *p2 = &a[8];                    // конец
```

Тогда оператор вывода:

```
cout << (p2 - p1) << endl;
```

выведет на экран число `7` — количество элементов массива от `a[1]` до `a[8]`.

Обращаю внимание, что правый элемент не включается в это количество, т. е. мы имеем полуинтервал адресов [начало, конец). Напомню, что индексы полного массива изменяются на полуинтервале [0, n).

Указатели, итераторы и контейнеры

До сих пор все циклы по массивам мы писали, изменяя индексы. Даже цикл с использованием выражения `*(p + i)` с указателем написан с изменением индекса. Но к элементам массива можно обращаться с помощью указателя. Для этого надо задать начальное и конечное значения указателя — адреса.

Определим эти значения в виде полуинтервала аналогично полуинтервалу индексов. Начальное значение — это адрес элемента массива с индексом `0`. Конечное значение указателя должно быть таким, чтобы разница между конечным и начальным адресами равнялась количеству элементов массива. Это легко сделать — посмотрите код в листинге 11.4.

Листинг 11.4. Указатель как итератор встроенного массива

```
int arr[] {10,20,30,4,5,6,7,8,9,0};
auto *it{arr};                      // указатель = итератор (начало)
```

```
auto *finish{arr + size(arr)};          // конечный адрес
while(it != finish){                    // цикл по итератору
    cout << *it << ' ';                 // доступ по итератору
    ++it;                               // изменение итератора
}
```

В коде объявлены указатель для перемещения по массиву (`it`) и указатель конечного адреса (`finish`). В начале они определяют полуинтервал [`arr, finish`), как и индексы. Конечное значение `finish` — это адрес после последнего элемента массива. По определению `finish` разность (`finish - arr`) равна `size(arr)`.

Указатель `it`, передвигаясь по массиву, сравнивается с финишным адресом. Сравнение делается с помощью неравенства — это обычная практика в подобных циклах.

Поздравляю вас, читатель! Вы только что осуществили обработку массива с помощью итератора! *Итератор* — это объект, обеспечивающий косвенный доступ к элементам контейнера. *Контейнер* (в C++) — это набор однотипных элементов. Массив является встроенным контейнером в C++. Встроенный указатель является встроенным итератором по встроенному массиву.

Практически то же самое можно проделать и с array-массивом (листинг 11.5).

Листинг 11.5. Указатель как итератор *array*

```
array m {1,2,3,4,5,6,7,8,9,0};          // массив
auto it = m.data();                     // указатель = итератор (начало)
auto endm{it+size(m)};                  // конец
while(it != endm){                      // цикл по итератору
    cout << *it << ' ';                 // доступ по итератору
    ++it;                               // изменение итератора
}
```

Единственная разница заключается в присвоении начального адреса (вторая строка в коде): начальный адрес выдает метод `data`. Можно использовать для той же цели функцию `data`:

```
auto it = data(m);                      // функция data
```

Это более безопасный способ, чем явное присвоение адреса:

```
auto it{&m[0]};                 // итератор = указатель (начало)
```

Мы использовали указатели для array-массива. Но такой массив является одним из стандартных контейнеров STL, поэтому для него, как и для всех контейнеров, реализованы «настоящие» итераторы. Итераторы в стандартной библиотеке «замаскированы» под встроенные указатели, т. е. операции с итераторами не отличаются от операций с указателями. Отсутствует только одна операция — операция получения адреса. Для итераторов эта операция не нужна, т. к. итераторы — не указатели. Вместо этой операции реализованы две функции для установки начального и конечного значения.

Для array-массива m код с итераторами представлен в листинге 11.6.

Листинг 11.6. Итераторы array-массива

```
auto it{begin(m)};                    // итератор - начало
while(it != end(m)){                  // цикл по итератору
    cout << *it << ' ';               // доступ по итератору
    ++it;                             // изменение итератора
}
```

В этом фрагменте кода цикл фактически не отличается от предыдущего варианта. Здесь использованы стандартные функции begin и end. Имена у функций «говорящие», поэтому первая устанавливает итератор на начальный элемент array-массива, а вторая выдает такое значение, чтобы равенство:

```
end(m) - begin(m) = size(m)
```

выполнялось.

Таким образом, значения этих функций для любого массива m определяют полуинтервал [begin(m), end(m)). В стандартной библиотеке полуинтервал итераторов принято обозначать как [first, last).

Операция инкремента передвигает итератор к следующему элементу точно так же, как передвигался указатель в коде листингов 11.4 и 11.5. А звездочка является операцией разыменования, как и для указателя.

Поскольку array-массив реализован в виде класса-шаблона, то обращение к его итераторам можно записывать в объектно-ориентированном стиле, как вызов метода, например: m.begin(), m.end().

Самое интересное, что мы можем использовать эти функции для работы и со встроенными массивами. Код обработки встроенного массива отличается от кода обработки array-массива m только именем массива.

Совершенно аналогично можно обрабатывать массивы в обратном порядке. Сначала напишем код со встроенным массивом и указателями (листинг 11.7).

Листинг 11.7. Обработка массива в обратном порядке (указатели)

```
auto start = arr;                     // начало
auto it = start + size(arr);          // итератор (конец)
do{ --it;                             // сначала двигаем
    cout << *it <<' ';                // потом обрабатываем
} while (it != start);                // потом сравниваем
```

В начале, как обычно, создаются начальный и конечный адреса. Изменяемый итератор (указатель) инициализируется конечным значением — адрес после последнего элемента массива. Поэтому в цикле сначала двигается итератор (указатель), а потом выполняется обработка. Когда операция декремента --it передвинет итера-

тор (указатель) на начальный элемент массива, то условие цикла станет ложным и цикл завершится.

Обработка массива в обратном порядке с помощью итераторов делается с помощью *обратных* итераторов (reverse iterators):

```
for(auto it{rbegin(m)}; it != rend(m); ++it) cout << *it << ' ';
```

Имена функций `rbegin` и `rend` говорят о том, что обработка выполняется в обратном (reverse) порядке. Однако изменение итератора выполняется той же операцией инкремента. Инкремент в реализации *перегружен* таким способом, чтобы итератор перемещался от последнего элемента к первому.

Для обратных итераторов тоже выполняется известное нам соотношение:

```
rend(m) - rbegin(m) = size(m)
```

В противовес обратному итератору обычный итератор называют прямым.

Это программистский жаргон (сленг), но нужно помнить, что:

- прямые итераторы передвигаются от начала к концу массива;
- обратные итераторы передвигаются от конца массива к его началу.

В обоих случаях изменение итератора осуществляет операция инкремента, что и показано на рис. 11.4.

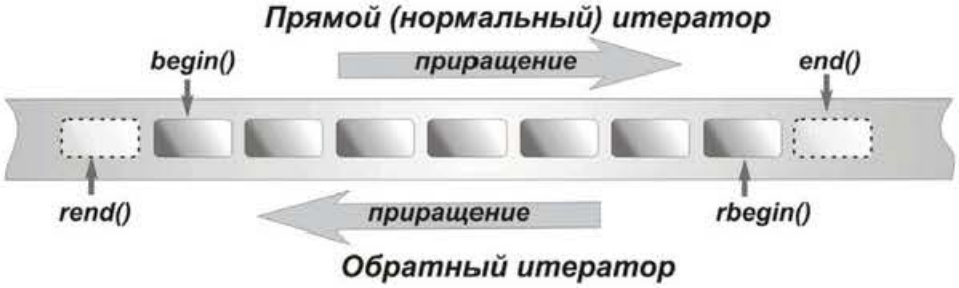


Рис. 11.4. Обычные и обратные итераторы

Итераторы (и прямые, и обратные) могут быть константными. Это отражено в именах функций:

- прямые константные называются `cbegin` и `cend`;
- обратные константные — `crbegin` и `crend`.

Но для итератора константность означает запрет изменения элементов массива, по которому он передвигается, а не самого итератора.

Функция *distance*

Разность конечного и начального указателей дает нам количество элементов массива. То же самое можно сделать и с итераторами. Для array-массива можно применять разность, однако с другими контейнерами стандартной библиотеки надежнее

использовать для этого реализованную в библиотеке функцию `distance` (нужно подключить библиотеку `<iterator>`).

Если у нас в программе объявлен array-массив `m`, то все операторы вывода:

```
cout << (end(m)-begin(m)) << endl;
cout << distance(begin(m), end(m)) << endl;
cout << (rend(m)-rbegin(m)) << endl;
cout << distance(rbegin(m), rend(m)) << endl;
```

выведут в консоль одно и то же значение, равное `size(m)`. То же самое будет выведено и для константных итераторов.

При вычислении разности итераторов из конечного вычитается начальный.

Но в функции аргументы задаются в том же виде, как и в полуинтервале [`first`, `last`). В таком порядке итераторы задаются во всех стандартных функциях STL.

Итераторы ввода/вывода

Вам, наверное, уже очень надоело писать циклы для вывода массивов. Даже цикл для диапазонов — это все равно цикл. Стандартными средствами можно написать вывод в одну строку. Нам поможет стандартная функция `copy` и *потоковые* итераторы. Любой массив `m` может быть выведен так:

```
copy(begin(m), end(m), ostream_iterator<int>(cout, ", "));
```

Третий аргумент функции называется потоковым итератором вывода. Чтобы его правильно написать, нужно сделать следующее:

- указать в угловых скобках тип выводимых элементов;
- указать объект `cout`;
- задать разделитель между элементами.

В примере указан тип элементов `<int>`, разделителем является строка из двух символов `", "`. Это приведет к выводу элементов массива через запятую в одну строку. Если надо вывести в столбик, задайте символ `"\n"`.

Итератор можно объявить как переменную с инициализацией:

```
ostream_iterator<int> out_iter {cout, ", "};
```

Потом этот объект можно задать в вызове функции `copy`:

```
copy(begin(m), end(m), out_iter);
```

Аналогично можно вводить массив, используя итератор потока ввода:

```
array<int, 10> d;
copy(istream_iterator<int>(cin), istream_iterator<int>(), begin(d));
```

Это позволит заполнить массив `d`, вводя значения с клавиатуры. После каждого значения можно нажимать либо пробел, либо клавишу <Enter>. Завершается ввод нажатием комбинации клавиш <Ctrl>+<Z> (<Ctrl>+<D> в Linux) и последующим нажатием клавиши <Enter>.

Задачи и упражнения

1. Напишите программу, аналогичную представленной в листинге 11.3 для операции декремента.
2. Перепишите программы *урока 9* (листинги 9.4, 9.7, 9.5) с итераторами.
3. Перепишите те же алгоритмы поиска с обратными итераторами.

- Урок 12 -

Массивы и функции

До сих пор мы не писали функций с параметрами-массивами только по одной причине: сначала надо было разобраться с указателями и понять, как связаны указатели и массивы. Теперь мы будем писать такие функции регулярно.

Заполнение, преобразования элементов, вычисление некоторых величин, сортировки, поиск — это типичная работа с массивами. Практически любая обработка массивов пишется в виде функций. Огромное количество таких функций реализовано в стандартной библиотеке, и мы с ними вскоре познакомимся. Напомню, что массив — это встроенный последовательный контейнер, а array-массив — один из реализованных в STL последовательных контейнеров. Мы обязательно познакомимся и с другими последовательными контейнерами, а также с другими видами контейнеров.

Массивы и функции

Реализуем задачу вычисления суммы элементов вещественного массива в виде функции, параметром которой будет массив. Напомню, что мы уже писали функцию, в которой суммировали множество чисел, используя статическую переменную (см. листинг 7.11). Теперь вычислим сумму более традиционным способом — как сумму элементов массива. Сначала рассмотрим варианты для встроенного массива.

Мы, как настоящие программисты, будем писать универсальную функцию, которая может работать с массивами любого размера. Понятно, что главный параметр — это массив. Первая версия функции:

```
double Summa(const double m[]){
    double summa = 0.0;
    for(int i {0}; i < size(m); ++i){          // ошибка компиляции
        summa += m[i];
    }
    return summa;
}
```

Однако такой код даже не компилируется — ошибка возникает в операторе `for` из-за функции `size`. Оказывается, параметр `m` не является массивом! И это действи-

тельно так: вспомним, что имя массива является адресом. Поэтому параметром в данном случае является указатель, а не массив. Естественно, компилятор «ругается» по поводу функции `size` — она не работает с указателями.

Функция, получающая массив в качестве параметра, «не знает», сколько элементов в этом массиве. Нужно задавать второй параметр — количество элементов массива. Код функции и ее вызов показаны в листинге 12.1.

Листинг 12.1. Функция суммирования массива (версия 1)

```
#include <iostream>
using namespace std;
double Summa(const double m[], int n){
    double summa = 0.0;
    for(int i {0}; i < n; ++i){
        summa += m[i];
    }
    return summa;
}
int main(){
   double arr[]{1,1,1,1,1,1,1,1,1,1};          // тестовый массив
   double s = Summa(arr, size(arr));           // вызов
   cout << s << endl;                          // сумма равна 10
   return 0;
}
```

Массив передается константным, т. к. функция не собирается изменять элементы массива — это обычная практика при написании таких функций.

Можно при вызове использовать начальный итератор:

```
double s = Summa(begin(arr), size(arr));              // вызов с итератором
```

Вместо обычного итератора можно применить константный итератор:

```
double s = Summa(cbegin(arr), size(arr));             // вызов с итератором
```

Встроенный массив невозможно передать в функцию по значению, т. к. фактически вместо массива имеем указатель в качестве параметра. Поэтому можно указатель в заголовке написать явно:

```
double Summa(const double *m, int n)
```

Работа программы от этого никак не изменится.

Передача массива по ссылке

Как и любые другие параметры, встроенный массив может быть передан по ссылке или по константной ссылке. Однако форма записи параметра в этом случае несколько необычна (посмотрите код в листинге 12.2).

Листинг 12.2. Функция суммирования массива (версия 2)

```
#include <iostream>
using namespace std;
double Summa(const double (&m) [10]){          // ссылка
    double summa = 0.0;
    auto i = 0ull;                             // тип size_t
    while(i < size(m)){
        summa += m[i];
        ++i;
    }
    return summa;
}
int main(){
   double arr[]{1,1,1,1,1,1,1,1,1,1};          // тестовый массив
   double s = Summa(arr);                              // вызов
   cout << s << endl;                                  // сумма равна 10
   return 0;
}
```

Массив передается по константной ссылке, т. к. не требуется изменять элементы массива. Ссылка записана в скобках `(&m)`, т. к. без них возникает ошибка компиляции. Размер массива после ссылки тоже требуется задавать, т. к. без конкретного размера функция `size` не может его вычислить, и возникает ошибка компиляции.

Запись параметра-массива по ссылке в общем виде выглядит так:

```
тип (&имя) [размер]
```

Так же пишутся синонимы для массива в операторе `typedef` (см. *урок 9*).

Отметим, что эта версия функции не очень универсальна, т. к. требуется явно задавать конкретный размер. С массивом другого размера эта функция работать не будет, и придется писать другую. Это не очень хорошо, т. к. фактически один и тот же алгоритм (в данном случае суммирование массива) приходится дублировать для массивов разных размеров.

Справедливости ради отметим, что передача массива по ссылке позволяет суммировать массив с помощью самого безопасного цикла — по диапазону:

```
double Summa(const double (&m) [10]){
    double summa = 0.0;
    for(auto t: m){
         summa += t;
    }
    return summa;
}
```

Очевидно, что такой цикл намного проще традиционных.

И обращаю ваше внимание на то, что встроенный массив невозможно вернуть из функции, т. к. имя массива — указатель. Только поместив массив в структуру, можно получить эту структуру в качестве результата с массивом внутри.

Перегрузка функций

Функция версии 1 прекрасно работает и даже позволяет вычислить сумму любой части массива от начала. Например, вызов:

```
double s = Summa(arr, 5);                // вызов
```

вычисляет сумму первых пяти элементов массива (индексы `[0,5)`). А для вычисления суммы всех элементов без последнего можно написать вызов:

```
double s = Summa(arr, size(arr)-1);      // вызов
```

Функция позволяет вычислить сумму некоторой части массива, которая начинается не с первого элемента (с индексом 0), например:

```
double s = Summa(arr+2, size(arr)-2);    // вызов
```

Этот вызов позволяет вычислить сумму элементов, начиная с `arr[2]` и до конца массива. Однако в таких вызовах надо внимательно следить, чтобы не «вылететь за границу» — за последний элемент массива. Последствия, как правило, не предсказуемы. Непредсказуемое поведение программы в мире C++ называется *неопределенным поведением*.

> ***На заметку***
>
> Термин «неопределенное поведение» (undefined behavior, UB) определен в стандарте как поведение программы, не регламентированное стандартом C++.

Нужно как-то изменить список параметров, чтобы возможных ошибок было поменьше. Давайте определим в заголовке функции три параметра: массив, индекс начала и индекс окончания (с учетом полуинтервала индексов) последовательности. По собственному опыту знаю, что с индексами обычно работать немного проще, чем с размером массива.

Имя функции, естественно, оставим прежним, т. к. она выполняет ту же работу — суммирование. Но тогда у нас в программе получится две функции с одинаковыми именами. К счастью, C++ позволяет это делать, предоставляя механизм перегрузки функций, о чем мы уже не один раз упоминали.

Перегрузка функций — это механизм C++, который позволяет писать функции с одинаковыми именами в одной программе. Этот механизм используется тогда, когда ряд функций должен выполнять однотипную работу над объектами разных типов. Например, все функции вывода разнообразных объектов в консоль естественно назвать одним именем `print`.

Имена функций одинаковые, поэтому компилятор должен разобраться, какую из функций в каком месте нужно вызвать. Он это делает по спискам параметров — у перегруженных функций должны быть разные списки параметров по количеству и/или по типам.

Код обеих функций представлен в листинге 12.3.

Листинг 12.3. Перегрузка функции суммирования (версия 3)

```
#include <iostream>
using namespace std;
double Summa(const double m[], int n){          // список параметров
    double summa = 0.0;                         // накапливаемая сумма
    for(int i {0}; i < n; ++i){
        summa += m[i];
    }
    return summa;
}
// перегруженная функция с другим списком параметров
double Summa(const double m[], int start, int finish){
    double summa = 0.0;
    int i = start;                          // первый элемент последовательности
    while(i < finish){          // полуинтервал [start,finish)
        summa += m[i];
        ++i;
    }
    return summa;
}
int main(){
   double arr[]{1,1,1,1,1,1,1,1,1,1};   // тестовый массив
   double s = Summa(arr, size(arr));    // вызов первой функции
   cout << s << endl;                   // сумма равна 10
   s = Summa(arr, 0, size(arr));                // вызов второй функции
   cout << s << endl;                   // сумма равна 10
   return 0;
}
```

При запуске программы обе функции выдают одинаковый результат.

Существует много тонкостей в определении перегружаемых функций, с которыми мы пока разбираться не будем. Подумаем, как еще можно написать список параметров для снижения потенциального количества ошибок.

Вспомним про итераторы. Можно написать функцию с двумя итераторами, которые позволят суммировать произвольную последовательность элементов в массиве любого размера. Естественно, функция тоже имеет имя `Summa`. Код ее показан в листинге 12.4.

Листинг 12.4. Функция суммирования (версия 4)

```
#include <iostream>
using namespace std;
// параметры-итераторы
```

```
double Summa(const double *start, const double *finish){
    double summa = 0.0;
    auto it = start;                    // начальный итератор
    while(it != finish){                // полуинтервал [start, finish)
        summa += *it;                   // доступ по итератору
        ++it;                           // продвижение итератора
    }
    return summa;
}
int main(){
   double arr[]{1,1,1,1,1,1,1,1,1,1};          // тестовый массив
   double s = Summa(begin(arr), end(arr));     // вызов
   cout << s << endl;                          // сумма равна 10
   return 0;
}
```

Эта функция написана в той же программе, что и предыдущие версии — перегрузка нам позволяет это делать. Модификатор `const` при итераторах (в данном случае это указатели) задан по той же причине, что и прежде: не требуется изменять элементы массива.

И при запуске функции выводится все тот же результат `10`.

Эта последняя функция лучше предыдущих версий по двум причинам:

- она самая универсальная — позволяет вычислить сумму любого отрезка массива, и размер массива не имеет значения;
- у функции меньше всех параметров (как у первой версии, но здесь параметры одного типа).

Хотя функция с передачей массива по ссылке (см. листинг 12.2) имеет всего один параметр, она, как мы отмечали выше, совершенно не универсальна, т. к. требуется писать такие же функции для каждого нового размера массива.

Про массив *array*

Теперь посмотрим, как работать в функциях с array-массивом. Так как такой массив представляет собой полноценный объект, то его передают в функцию и по значению, и по ссылке. При этом нет необходимости передавать размер отдельным параметром, т. к. его всегда можно получить с помощью функции (или метода) `size`. Но даже этого не требуется, т. к. полная информация о типе и размере позволяет использовать цикл по диапазону (код в листинге 12.5).

Листинг 12.5. Функция суммирования с параметром *array*

```
#include <iostream>
#include <array>
using namespace std;
```

```
double Summa(const array<double,10> &m){
    double summa = 0.0;
    for(auto t: m){                          // цикл по диапазону
        summa += t;
    }
    return summa;
}
int main(){
   array<double,10> d {1,1,1,1,1,2,2,2,2,2};
   double s = Summa(d);
   cout << s << endl;
   return 0;
}
```

Обратите внимание: использован цикл по диапазону так же, как мы это сделали выше для функции с передачей встроенного массива по ссылке. И так же, как и та функция, эта тоже не обладает универсальностью: для массивов с другим размером придется писать новую функцию.

Опять вспомним об итераторах. Напомню, что для array-массива, как обычно, определены четыре типа итераторов:

- `iterator` — прямой обычный итератор;
- `const_iterator` — прямой константный итератор;
- `reverse_iterator` — обратный итератор;
- `const_reverse_iterator` — обратный константный итератор.

Таким образом, можно перегрузить четыре варианта функции с параметрами-итераторами. Однако «родные» итераторы для array-массива практически не используются, т. к. тип итератора array-массива очень длинный:

```
array<double,0>::const_iterator start
```

Именно в таких объявлениях стало использоваться слово `auto`. Но все равно нужно знать, каким образом объявлять итераторы с полным наименованием типа. Поэтому рассмотрим составные части этого типа.

Тип объекта-итератора (для любого контейнера) состоит из двух частей: префикса и самого типа. Префикс `array<double,0>::` показывает, для какого array-массива объявляется итератор. В качестве размера задан ноль.

> ***На заметку***
>
> Вместо нуля можно написать какое угодно число.

В общем случае итераторы никак не зависят от размеров контейнера, и для других видов контейнеров размеры никогда не указываются. Но для массива типа `array` мы вынуждены задавать хотя бы какое-нибудь число, иначе возникает ошибка при компиляции. И это мешает при чтении программы.

Вместо полного типа итераторов для array-массивов практически всегда пишут auto. Стандарт C++20 поддерживает такую возможность даже для параметров функций. Так можно писать даже в C++17, если задать режим компиляции -fconcepts.

В листинге 12.6 показан код такой функции.

Листинг 12.6. Обобщенная функция суммирования

```
auto Summa(auto start, auto finish){
    auto summa = 0.0;
    auto it = start;
    while(it != finish){
        summa += *it++;          // сначала разыменование
    }
    return summa;
}
```

В теле цикла я написал уже знакомую нам конструкцию *it++. Вычисляется это выражение точно так же, как было описано для указателей (см. листинг 11.3).

Вызов функции можно писать и с константными, и с обычными итераторами:

```
array<double,10> d {1,1,1,1,1,2,2,2,2,2};
double s = Summa(cbegin(d), cend(d));
```

Функция получилась универсальной (обобщенной) и позволяет суммировать массивы с любым типом элементов:

```
array<int,10> d {1,1,1,1,1,2,2,2,2,2};
int s = Summa(cbegin(d), cend(d));
```

Компилятор самостоятельно выводит тип на основании типа задаваемых при вызове аргументов.

Шаблоны функций

Возможность написать обобщенную функцию суммирования (см. листинг 12.6) с помощью auto-параметров появилась только в C++20. Однако потребность в подобных обобщенных функциях возникла еще в достандартную эру C++. Перегрузка функций — полезное свойство, позволяющее программисту запоминать меньше разных имен. Однако перегрузка приводит к слишком большому количеству практически одинаковых функций.

Вообще-то подобных типовых функций очень много, и они не обязательно обрабатывают массивы. Например, функция обмена двух объектов должна уметь обменять объекты любых типов. А ведь в C++ только встроенных типов более десяти. Но ведь еще есть реализованные типы, например строки. Поэтому нужно было иметь в языке такое средство, с помощью которого можно написать функцию один раз и использовать ее потом для любых объектов, не заботясь о типах.

Б. Страуструп — тоже программист и не любит писать одно и то же, поэтому он добавил в C++ *шаблоны* функций задолго до появления стандарта. Общий вид шаблона функции выглядит следующим образом:

```
template <список параметров шаблона>
заголовок функции
тело функции
```

Слово template так и переводится — «шаблон».

Давайте сначала напишем конкретный шаблон простой функции обмена, а потом разберем его особенности (листинг 12.7).

Листинг 12.7. Шаблон функции обмена

```
template <class T>
void swap (T& x, T& y){
   T z = x; x = y; y = z;
}
```

В списке параметров шаблона определен параметр `T`, который использован как имя типа в заголовке и в теле функции. Имея такой шаблон, мы можем больше не заботиться о конкретных типах параметров функции обмена.

Компилятор «не обращает внимания» на шаблон до появления в коде вызова функции с конкретными типами. Например, в программе есть такие строки:

```
long k = 7, d = 12;
swap(k, d);
float p = 62.44, q = 68.3;
swap(p, q);
```

Компилятор сформирует два определения функции:

```
void swap (long& x, long& y){
   long z = x; x = y; y = z;
}
void swap(float& x, float& y){
   float z = x; x = y; y = z;
}
```

Процесс конкретизации (указание конкретных типов) шаблона называется *инстанцированием.* Вобще-то инстанцирование можно задать явно:

```
swap<long>(k, d);
swap<float>(p, q);
```

Но в этом нет необходимости, т. к. компилятор самостоятельно выводит тип аргументов аналогично тому, как он это делает для объявления типа `auto`.

Точно так же мы можем написать шаблоны разных вариантов функции суммирования. Их должно быть несколько, т. к. списки параметров у функций разные. Например, код функции-шаблона с параметрами-указателями выглядит, как показано в листинге 12.8.

Листинг 12.8. Шаблон функции суммирования с указателями

```
template<typename T>
T Summa(const T *start, const T *finish){
    T summa = 0.0;                              // тип как у указателей
    auto it = start;
    // цикл суммирования
    return summa;
}
```

В списке параметров шаблона функции обмена написано слово `class`, а в новом шаблоне использовано слово `typename`, которое так и переводится — «имя типа»; пишите то, которое вам больше нравится.

Теперь мы сможем суммировать не только `double`-массивы, но и массивы с любым другим типом элементов, например:

```
int iar[]{-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
double s = Summa(iar, iar+size(iar));  // явные указатели на int
cout << s << endl;
```

Заметим, что данный вызов для `int`-массива просуммирует целые в переменную типа `int`, но при присваивании произойдет неявное приведение целого в вещественное.

Можно вызвать эту же шаблонную функцию и с итераторами:

```
double s = Summa(begin(iar), end(iar));        // итераторы на int
cout << s << endl;
```

Результат будет тот же.

Теперь напишем шаблон функции суммирования с параметром-ссылкой. Самое интересное, что в функции параметр единственный, но нам нужно вызывать функцию для массивов и разных размеров, и с разными типами элементов. То есть требуется два параметра, причем один параметр — это тип элементов массива, а второй — его размер. Однако в функции только один параметр-ссылка, поэтому эти два параметра должны быть параметрами шаблона. Стандарт C++ позволяет задавать в шаблоне параметры, не являющиеся типами. Код функции-шаблона представлен в листинге 12.9.

Листинг 12.9. Шаблон функции суммирования с параметром-ссылкой

```
template<typename T, size_t N>          // параметр - размер
T Summa(const T (&m) [N]){
    T summa = 0.0;
    for(auto t: m){ summa += t; }
    return summa;
}
```

В этом шаблоне два параметра: тип `T` и размер `N`.

Вызов этой функции с массивом `iar` (см. выше) выглядит так:

```
double s = Summa(iar);          // массив целого типа
```

Тип и размер известны компилятору из объявления массива. При трансляции этого вызова они подставляются компилятором на место параметров в заголовке и теле шаблона.

Как вы понимаете, точно такой же шаблон можно написать для функции с параметром array-массивом. Напишите этот шаблон самостоятельно.

В этом разделе мы использовали только простейшие возможности механизма шаблонов функций.

Стандартные алгоритмы

До этого момента все массивы в наших программах были маленькие, и мы их инициализировали при объявлении. Но в реальной программисткой жизни массивы обычно большие, и объявить массив с явной инициализацией просто физически невозможно. Если быть точным, то мы можем лишь обнулить весь массив при объявлении (да и то один-единственный раз), а заполнить его, скажем, единицами, — не можем.

> ***История из жизни***
>
> Однажды я занимался разработкой программы для моделирования процесса перколяции. В этой программе нужно было не менее тысячи раз очищать массив данных и заполнять его потом по определенному алгоритму. Самый маленький массив, который нужно было обрабатывать, включал не менее миллиона элементов. Этот маленький массив использовался для первичной проверки алгоритма заполнения. А при последующих прогонах программы массив надо было увеличить до 100 миллионов элементов и даже до полутора миллиардов элементов.

Конечно, мы можем написать функцию, заполняющую массив нужными значениями. Можем даже написать шаблон функции, чтобы не писать сто раз одно и то же для разных типов и размеров. Однако такая работа уже сделана разработчиками стандартной библиотеки. Нам нужно только правильно использовать предоставленные средства в наших программах.

В стандартной библиотеке шаблоны всех подобных функций называются алгоритмами, т. к. они работают независимо от размеров массива и от типа его элементов. Они реализованы в максимально универсальном виде и позволяют работать с любыми контейнерами STL. Для их использования нужно подключить библиотеку **<algorithm>**.

Заполнение массива

Заполнение массива (и вообще любого контейнера) некоторой константой — наиболее часто встречающаяся операция. Все алгоритмы работают с контейнерами, поэтому первые два параметра большинства алгоритмов — начальный и конечный итераторы.

Стандартный алгоритм `fill` имеет прототип:

```
template<class Iterator, class Type>
void fill(Iterator first, Iterator last, const Type& value);
```

Третий параметр — это значение, которым заполняется массив. В качестве параметра можно задавать любое выражение, которое приводится к типу элементов контейнера (массива). Вызов этой функции выполняется так:

```
fill(begin(m), end(m), -1);
```

Таким образом, весь массив заполняется константной `-1`.

Другая функция заполнения имеет первые два параметра такие же, как и в нашей первой функции суммирования (см. листинг 12.1): начальный итератор и размер. Она имеет прототип:

```
template<class Iterator, class Size, class Type>
void fill_n(Iterator first, Size n, const Type& value);
```

Вызывается функция так:

```
fill_n(begin(m), N, -1);                    // заполнение
```

Иногда нужно заполнить массив возрастающими значениями. Для такой операции в STL реализована функция `iota`, имеющая следующий прототип:

```
template< class ForwardIt, class T >
void iota( ForwardIt first, ForwardIt last, T value );
```

Третий параметр задает стартовое значение последовательности. Пусть в программе есть массив, например:

```
array<double,size(arr)> ma;
```

Мы можем заполнить его возрастающими значениями:

```
iota(begin(ma), end(ma), -4);   // заполняем возрастающими
```

Если размер `size(arr)` равен 10, то вывод массива в консоль покажет числа:

```
-4 -3 -2 -1 0 1 2 3 4 5
```

Можно заполнять новый массив, скопировав элементы из уже заполненного. Первый вариант функции имеет такой прототип:

```
template< class InputIt, class OutputIt >
OutputIt copy( InputIt first, InputIt last, OutputIt d_first );
```

Первые два параметра — это итераторы массива, из которого копируются значения. Третий параметр — итератор принимающего массива, обычно это `begin`. Возвращается тоже итератор принимающего массива, который устанавливается за последним скопированным элементом. Таким образом, эта функция вполне позволяет копировать меньший массив в больший, или копировать часть большого массива в меньший.

Выходной итератор можно не сохранять, поэтому функцию можно вызывать двумя способами: с присвоением выходного итератора и без присвоения. Копирование без присвоения обычно выполняют в тех случаях, когда нужно скопировать весь массив:

```
copy(begin(ma), end(ma), begin(arr));
```

Естественно, размер массива `ma` должен быть не больше размера массива `arr`.

Если размер принимающего массива значительно больше исходного, то обычно надо знать, на каком элементе завершилось копирование, и тогда функция вызывается следующим способом:

```
auto it = copy(begin(ma), end(ma), begin(arr));
```

Итератор `it` установлен на первый незаполненный элемент принимающего массива (`arr`).

Заполнение случайными числами

Довольно часто для тестирования требуется заполнить массив (контейнер) случайными числами. Мы уже использовали случайные числа, когда писали программы-тренажеры (см. *урок 8*). Поэтому будем считать, что объявлен датчик с именем `gen` и распределение с именем `uid`.

Первый способ — перемешать массив, заполненный функцией `iota`:

```
shuffle(begin(ma), end(ma), gen);
```

Массив `ma`, заполненный значениями от `-4` до `5`, может стать таким:

```
2 -4 -3 3 4 -1 1 5 0 -2
```

Второй способ заполнения — использовать стандартные алгоритмы `generate` и `generate_n`. Первые два параметра у них такие же, как и у показанных выше алгоритмов `fill`. Третьим параметром должен быть генератор случайных чисел. Однако не все так просто. Вызов:

```
generate(begin(ma), end(ma), gen);
```

заполнит массив (контейнер) числами от `gen.min()` до `gen.max()`, что, как правило, нас не устраивает. Можно, конечно, написать на месте третьего параметра выражение (как мы уже писали ранее — см. *урок 8*), но это тоже не всегда удобно.

> ***На заметку***
>
> Вообще говоря, на месте третьего параметра в алгоритмах `generate` можно написать любую функцию. В том числе и функцию, выдающую случайное число как результат. Но мы пока не умеем задавать функции в качестве параметров. Эту важную тему мы изучим далее (см. *урок 17*).

Вспомним про распределение и попробуем на месте третьего параметра написать `uid(gen)`:

```
generate(begin(ma), end(ma), uid(gen));          // ошибки трансляции
```

Однако такой вызов приводит к ошибкам трансляции. Правильный вызов должен быть таким:

```
auto Gen = bind(uid, gen);                       // связывание
generate(begin(ma), end(ma), Gen);               // ошибок нет
```

Первый оператор — это связывание распределения и генератора. Чтобы этот оператор работал, необходимо подключить библиотеку `<functional>`.

Поиск, подсчет, замена

В стандартной библиотеке реализовано множество типовых алгоритмов обработки массивов. Имена функций, как правило, «говорящие», например:

- `count` — подсчет элементов;
- `find`, `search` — поиск элементов;
- `replace` — замена элементов;
- `reverse`, `rotate` — перестановки элементов.

Внимательно изучив прототипы и параметры функций заполнения, вы без труда освоите эти и все другие алгоритмы стандартной библиотеки.

Рассмотрим еще алгоритмы поиска минимума и максимума (мы писали цикл поиска — см. *урок 9*). Прототип у этих функций совершенно типичный:

```
template< class ForwardIt >
ForwardIt min_element( ForwardIt first, ForwardIt last );
```

Поиск максимума отличается только префиксом `max_` вместо `min_`. Вызов:

```
auto result = min_element(begin(ma), end(ma));
```

Результат выполнения — итератор, показывающий на минимум массива. Так как `result` — это итератор, то сам минимум мы получаем разыменованием, а индекс в массиве вычисляется функцией `distance` (см. *урок 11*):

```
cout << "min = " << *result << "; "                    // значение
     << "ind = " << distance(begin(ma), result);       // индекс
```

Возврат итератора позволяет вызвать эту функцию слева от присваивания:

```
*min_element(begin(ma), end(ma)) = 100500;             // замена минимума
```

В результате минимальный элемент массива будет заменен на число `100500`. Точно так же работает и функция `max_element`.

Если вы напишете некую функцию для обработки массива, и она будет возвращать указатель на элемент массива, то эту функцию тоже можно будет использовать для присвоения нового значения этому элементу.

В библиотеке реализована функция, которая позволяет найти в массиве сразу и минимум, и максимум. Прототип функции следующий:

```
template< class ForwardIt >
std::pair<ForwardIt, ForwardIt>                  // результат
minmax_element( ForwardIt first, ForwardIt last );
```

Функция интересна тем, что возвращает пару итераторов. Пара (pair) — это реализованный в стандартной библиотеке класс-шаблон. Напомню, что пары чисел встречаются повсеместно (см. *урок 10*): числитель и знаменатель дроби, координаты x и y, часы и минуты, широта и долгота и т. п. Поэтому естественно, что в STL реализован такой класс-шаблон.

При вызове этой функции можно применять декомпозицию результата:

```
auto [mini, maxi] = minmax_element(begin(ma), end(ma));
```

Так как `mini` и `maxi` — итераторы, то для получения значений надо их разыменовать.

Числовые алгоритмы

В стандартной библиотеке есть и алгоритмы, которые предназначены для работы с числовыми массивами (контейнерами). Выше мы писали много разных вариантов функции суммирования. Естественно, в стандартной библиотеке реализован алгоритм для суммирования `accumulate`.

Прототип функции следующий:

```
template< class InputIt, class T >
T accumulate( InputIt first, InputIt last, T init );
```

Функция возвращает значение, поэтому вызывается точно так же, как и версия нашей функции `Summa` с итераторами (см. листинг 12.4):

```
double summa = accumulate(begin(ma), end(ma),0);
```

Среднее арифметическое массива (контейнера) тоже легко посчитать:

```
double average = accumulate(begin(ma), end(ma),0)/size(ma);
```

В олимпиадном программировании при работе с массивами встречается такой прием, как «префиксные суммы»: для каждого элемента массива считается сумма от начала до этого элемента включительно.

Пусть префиксные суммы массива `ma` сохраняются в массиве `p`. Тогда:

```
p[0] = ma[0]
p[1] = p[0] + ma[1]
p[2] = p[1] + ma[2]
…
p[i] = p[i-1] + ma[i]
…
p[n-1] = p[n-2] + ma[n-1]
```

Такие суммы позволяют быстро получать суммы произвольных отрезков массива без использования циклов — простым вычитанием.

Прототип функции, которая позволяет их вычислять, совершенно типичный:

```
template< class InputIt, class OutputIt >
OutputIt partial_sum(InputIt first, InputIt last, OutputIt d_first);
```

Первые два параметра — итераторы входного массива (или контейнера), третий — итератор начала массива префиксных сумм. Результат — итератор `end` выходного массива (контейнера), который можно не сохранять. Вызов:

```
partial_sum(begin(ma), end(ma), begin(p));
```

Выходным массивом может быть исходный массив. Тогда элементы массива будут заменены префиксными суммами.

Задачи и упражнения

1. Напишите шаблон с двумя параметрами для функции вычисления суммы массива array, передаваемого по ссылке.

2. Реализовать задания 1–4 *урока 9* как функции, которые обрабатывают встроенный массив. Перегрузить функции со списками параметров:
   - массив, размер;
   - указатель, указатель;
   - auto, auto.

   Остальные параметры задаются в зависимости от задачи. Исходный массив заполнить случайными числами в зависимости от задачи.

   Функции должны вернуть следующие результаты:

   1. Сумма.
   2. `-1`, если сумма четных меньше; `+1`, если сумма четных больше; `0`, если суммы равны.
   3. `-1`, если четных меньше; `+1`, если четных больше; `0`, если равно.
   4. Индекс элемента или `-1` при его отсутствии.
3. Реализовать в виде функции, обрабатывающей любой массив, задание 5 *урока 9*. Исходный массив заполнить случайными числами и вывести до и после вызова функции в главной программе.
4. Реализовать как функцию сортировки задание 6 *урока 9*. Функция должна быть реализована в двух вариантах: без применения стандартных алгоритмов и с применением подходящих. Исходный массив заполнить случайными числами и вывести до и после вызова функции в главной программе.
5. Реализовать как функцию-предикат задание 7 *урока 9*.

- Урок 13 -

Строки, строки, строки

Мы уже не раз встречались со строками. Строки — наиболее часто встречаемая в самых различных программах структура данных. Например, при входе на многие сайты мы вводим логин и пароль — это строки. Исходный код программы на языке C++ — это строки. Поэтому программист обязан уметь работать со строками.

Строки типа `string`, с одной стороны, являются обычными скалярными объектами, с которыми можно работать почти так же, как с объектами встроенных типов: можно объявлять и инициализировать переменные типа `string` любыми способами, их можно присваивать, передавать в функции любыми способами и возвращать как результат, из них можно создавать массивы, их можно включать в качестве полей в структуры и классы и т. п.

Но с другой стороны, строка является последовательностью символов, и с этой точки зрения представляет собой массив символов. Поэтому для строки фактически должны работать все средства, которые вы изучали для массивов.

Однако язык C++ унаследовал массивы символов и все средства работы с ними из языка C. В мире написаны (и до сих пор пишутся) миллионы строк кода с массивами символов. Поэтому программист обязан знать о них и уметь с ними работать. И надо разобраться, в чем похожи и чем отличаются массивы символов и реализованные строки.

Однако при обработке строк посимвольно возникает «засада» — кодировка символов. Это не самая простая проблема даже для профи — поэтому программы этого урока создавались и тестировались в Windows с кодировкой cp-1251. Программы будут правильно работать в Linux с кодировкой UTF-8, но только с цифрами и английским алфавитом.

Строки как массивы символов

Для демонстрации возможностей работы со строками реализуем простую функцию, которая определяет, является ли строка палиндромом. Напомню, палиндром — это последовательность символов, которая читается одинаково слева направо и обратно. Например, слово «потоп» является палиндромом. Существует масса решений этой простой задачи.

Напишем функцию-предикат, принимающую строку (листинг 13.1).

Листинг 13.1. Функция-предикат «Палиндром» (версия 1)

```
bool isPalindrom(const string &s){
    bool yes = false;
    auto left = 0u;                    // индекс первого символа
    auto right= size(s)-1;             // индекс последнего символа
    while((s[left] == s[right])&&(right > left)){
        ++left;                        // сдвигаемся к следующему
        --right;                       // сдвигаемся к предыдущему
    }
    if(!(right > left)) yes = true;
    return yes;
}
```

Так как строку в процессе решения задачи изменять не требуется, то строка в эту функцию передается по константной ссылке так же, как и array-массив (см. листинг 12.5). А внутри функции осуществляется доступ по индексу к символам. Цикл выполняется до середины строки. Если в строке нечетное количество символов (слово «12321»), то в конце концов индекс `left` станет равным `right`, и цикл закончится. Если же количество символов четное, то `left` станет больше `right`, и цикл тоже закончится.

В этом случае в условном операторе условие будет истинным, и функция вернет `true`. В случае, когда проверяемая строка не является палиндромом, выход произойдет по первому сравнению в условии: `s[left] != s[right]`. В этом случае `right > left` и возвращается `false`.

Итераторы для строки

Как и для любого массива, для прохода по строке можно применять итераторы. Конечно, надо использовать константные итераторы, т. к. строку изменять не требуется. Решение с итераторами показано в листинге 13.2.

Листинг 13.2. Функция-предикат «Палиндром» с итераторами (версия 2)

```
bool isPalindrom(string::const_iterator start,
                 string::const_iterator finish)
{   while((*start == *finish)&&(distance(start, finish) > 1)){
        ++start;                // двигаем вправо
        --finish;               // двигаем влево
    }
    return(!(distance(start, finish) > 1));     // true или false
}
```

Код получился чуть короче, чем первая версия. Мы избавились от булевской переменной `yes`, т. к. просто возвращаем результат сравнения итераторов.

Вызов такой функции такой же, как и для массива:

```
string p {"12321"};
cout << isPalindrom(begin(p), end(p)-1) << endl;
```

Второй итератор установлен на последний символ строки. В консоли получите константу `1`, которая означает, что функция выдала результат `true`.

Если вы работаете в C++20, то заголовок можно сделать короче за счет `auto`:

```
bool isPalindrom(auto start, auto finish)
```

Код тела функции остается прежним. Напомню, что так можно писать и в C++17, если задать режим компиляции `-fconcepts`.

Стандартные средства

Стандартные алгоритмы позволяют написать код гораздо короче. Ведь к строке, как к массиву, применимы стандартные алгоритмы. Идея проста: исходную строку копируем в новую и реверсируем ее. Реверс строки — это перестановка символов в обратном порядке. Если исходная строка является палиндромом, то новая строка должна совпадать с исходной (листинг 13.3).

Листинг 13.3. Функция-предикат «Палиндром» короткая (версия 3)

```
bool isPalindrom(const string &s){
    string rs = s;                          // новая строка
    reverse(begin(rs), end(rs));            // реверс строки
    return (s == rs);                       // true для палиндрома
}
```

Заметим, что в коде из трех строчек ошибиться просто негде. Единственная проблема, которая теоретически может возникнуть, — это скорость работы этой функции для очень длинных строк. Но если строка не содержит «стотыщпятьсотмиллионов» символов, то я предпочту эту версию.

Строка — массив переменного размера

А теперь внимательно почитаем коды функций для обработки строк и сравним их с аналогичными кодами функций для массивов. Я специально вынес этот абзац под отдельный заголовок, чтобы обратить ваше внимание на одну важную деталь. Для работы со встроенными массивами в параметрах почти всегда нужно задавать размер, т. к. функция про размер массива ничего не знает. С array-массивами иногда тоже требовалось задавать размер.

В отличие от встроенных и array-массивов, для строк мы ни в одной функции размер в явном виде не указывали. Строка знает свой размер. В операторах объявле-

ния объектов-строк конкретный размер никогда не задавался. Более того, размер строки изменяется во время работы программы. Мы уже применяли операцию сцепления для строк (см. листинг 3.2). Давайте сделаем это снова:

```
string ss = "0123456789";        // size(ss) = 10
ss = ss + "0123456789";          // операция сцепления, size(ss) = 20
```

Это оператор похож на оператор присваивания вроде `x = x + 1`. Вывод:

```
cout << ss << "; " << size(ss) << endl;
```

покажет, что строка `ss` содержит 20 символов `"01234567890123456789"`, и `size(ss)`, конечно, выдает `20`. Это означает, что строка является массивом символов, который изменяет свой размер во время работы программы. Но напомню, что строка — это реализованный массив. Можно даже называть строку стандартным контейнером символов.

Встроенные символьные массивы

Символьный массив объявляется так же, как и числовой массив любого типа:

```
char ss[10];
unsigned char ch[20];
```

Вопросы возникают при объявлении массива с инициализацией без указания размера. По общим правилам нужно писать инициализацию массива так:

```
char sm[] = {'0', '1', '2', '3', '4', '5', '6', '7', '8', '9'};
```

Но такая запись, конечно, вызывает «протест» программистов. Поэтому для массивов типа `char` (и только для них) разрешается «сокращенная» запись инициализации строковым литералом:

```
char ss[] = "0123456789";
```

Символьный массив можно выводить без цикла:

```
cout << ss << endl;
```

Символьный массив можно вводить без цикла:

```
cin >> ss;
```

Но, как и при вводе строк, ввод массива символов осуществляется до пробела (см. *урок 3*). Кроме того, надо помнить о размере массива, т. к. можно нечаянно задать при вводе больше символов, чем указано в размере. В этом случае программа работает как ни в чем не бывало (лишние символы заносятся в память после массива). Но последствия непредсказуемы, и в стандарте такие ситуации называются *неопределенным поведением* (см. *урок 15*).

Как обычно, в таком массиве доступ к отдельному символу осуществляется по индексу. Можно заменить любой символ другим, например:

```
ss[1] = 'a';
```

Однако массиву нельзя присвоить другой строковый литерал:

```
ss = "123";                      // ошибка компиляции
ss = "9876543210";               // ошибка компиляции
```

В первом случае компилятор жалуется на размер строкового литерала, а во втором случае (когда размер тот же самый) — на сам оператор присваивания.

Мы помним о связи указателей и массивов. Для символьных массивов эта связь была явно определена в языке C. Например, мы можем объявить массив символов вот таким образом:

```
const char *ps = "9876543210";
```

Здесь указатель «инициализирован» строковым литералом явно. Доступ к отдельному символу осуществляется обычным способом с помощью индекса. Например, `ps[1]` — это символ `'8'`.

И оператор вывода указателя выводит в консоль литерал, а не его адрес:

```
cout << ps << endl;
```

Обратите внимание, что указатель `ps` является указателем на константу. Без слова `const` объявить такой указатель невозможно. Константность означает, что строковый литерал нельзя изменять — при попытке заменить какой-нибудь символ возникает ошибка компиляции. И вводить таким образом объявленный символьный массив нельзя — он же константный. Но можно указателю присвоить другой строковый литерал, например:

```
ps = "012";
cout << ps << endl;
```

И мы, конечно, помним, что параметр-массив преобразуется в указатель. Это относится и к символьным массивам (см. ниже).

Размер встроенного символьного массива

Вычислим теперь размеры массивов. Мы помним, что `sizeof(char)` равно 1 по стандарту (см. *урок 4*). Поэтому размер показанного выше массива `sm` (см. выше, посимвольная инициализация), выдаваемый функцией `size(sm)` и операцией `sizeof(sm)`, должен быть один и тот же и равен 10:

```
cout << size(sm) << ' ' << sizeof(sm) << endl;
```

Получаем в консоли вывод:

```
10 10
```

Но для массива `ss` (см. выше, инициализация литералом) такой же оператор:

```
cout << size(ss) << ' ' << sizeof(ss)<< endl;
```

выдает:

```
11 11
```

Можно строковый литерал написать вместо имени массива в функции `size`:

```
cout << size("0123456789") << ' ' << sizeof("0123456789")<< endl;
```

И в этом случае получаем в консоли два числа: `11 11`. Получается, что символьный массив имеет ровно столько элементов, сколько указано при инициализации, а к строковому литералу добавляется еще один какой-то символ (его размер равен 1).

Дело в том, что еще в языке C было принято решение, что строковые литералы должны заканчиваться специальным символом, который является признаком окончания строки. Этот символ добавляется к литералу при компиляции, увеличивая размер на единицу.

На моем компьютере единицей памяти является байт, поэтому символ — это байт. Его значение равно нулю, поэтому назовем этот байт нуль-байтом. Нуль-байт не является символом `'0'`, он является специальным символом `'\0'`. Выведем числовое значение этих символов в консоль:

```
cout << int('0') << ' ' << int('\0') << endl;
```

Получаем в консоли значения `48` и `0`. Первое число, 48 — это *код* символа `'0'`, второе число, 0 — это нуль-байт. При задании размера символьного массива нужно учитывать наличие нуль-байта и задавать размер на единицу больше:

```
char dd[11] = "0123456789";
```

Хотя визуально в литерале 10 символов, размер массива должен быть равен 11, иначе возникает ошибка трансляции. Таким образом, функция `size` для символьных массивов, представленных строковыми литералами, должна применяться с учетом того, что она выдает размер на единицу больше количества видимых символов. Аналогично разность:

```
end(s) - begin(s) = size(s)
```

выдает результат на единицу больше. Для указателя на символы:

```
const char *ps = "9876543210";
```

функция `size(ps)` не компилируется, а `sizeof(ps)` выдает размер указателя.

Обработка символьных массивов

Так как с символьными массивами функция `size` работает не совсем так, как ожидается, нам требуется написать свою функцию, которая вычисляет размер встроенного символьного массива.

Встроенный символьный массив заканчивается нуль-байтом, поэтому для функций обработки таких массивов не требуется задавать длину массива, т. к. достаточно посчитать количество символов от начала строки до нуль-байта. Поэтому написать функцию, вычисляющую длину строки, очень легко (код приведен в листинге 13.4).

Листинг 13.4. Функция вычисления длины символьного массива (версия 1)

```
unsigned int Length(const char s[]){
   auto L = 0;                          // счетчик символов
   while (s[L])++L;
   return L;
}
```

Функция не изменяет массив, поэтому мы задали константность. Цикл будет выполняться до тех пор, пока очередной символ `s[L]` не станет равен нуль-байту. Как только функция «натыкается» на него, цикл заканчивается.

С постфиксным декрементом можно написать цикл короче:

```
while (s[L++]);
```

Так как при передаче параметра-массива происходит неявное преобразование в указатель, то можно цикл написать с изменением указателя:

```
while (*s++) ++L;
```

Сам указатель не является константным, поэтому его можно изменять. Функция будет работать абсолютно точно так же. Можно написать функцию с параметром-указателем (листинг 13.5).

Листинг 13.5. Функция вычисления длины символьного массива (версия 2)

```
unsigned int Length(const char *s)
{ auto L = 0;
  while (*s++)++L;
  return L;
}
```

Обе функции выдают один и тот же размер как для символьного массива, так и для строкового литерала:

```
char ch[] = "0123456789";
cout << Length("1234567890") << '\n';
cout << Length(ch) << '\n';
```

В обоих случаях выводится размер, равный 10.

Конечно, нам не было необходимости писать самим функцию вычисления размера символьного массива. Язык C++ унаследовал от языка C функции для работы с такими массивами и литералами. Чтобы работать с этими функциями, нужно подключить библиотеку `<cstring>`.

Название библиотеки прямо говорит нам о том, что это библиотека для обработки строк языка C (которые являются символьными массивами). Эти строки мы в дальнейшем будем называть *си-строками*, а строки `string` — стандартными строками, строками STL или просто строками.

Напишем функцию копирования си-строк, чтобы лучше понять, как с ними работать. Так как си-строки — это массивы, то присваивать их нельзя, а можно только посимвольно копировать в цикле (листинг 13.6).

Листинг 13.6. Функция копирования си-строк

```
char* copystr(char dst[], const char *src){
    char *rs = dst;                    // адрес новой си-строки
    while(*dst++ = *src++);            // цикл копирования
    return rs;
}
```

Функция копирует си-строку `src` в си-строку `dst`. Целевая си-строка `dst` в параметрах представлена как массив только для иллюстрации. Обратите внимание на цикл:

вся работа по копированию делается прямо в условии. Заканчивается копирование в тот момент, когда будет скопирован нуль-байт.

Функция возвращает адрес этого массива как результат для того, чтобы можно было вызвать эту функцию в операторе вывода, например:

```
char ch[] = "0123456789";
char dst[size(ch)];                              // размер не меньше исходного
cout << copystr(dst, ch) << endl;
```

Опять заметим, что писать эту функцию нам не было необходимости, т. к. она присутствует в библиотеке <cstring> с именем strcpy.

Универсальный параметр *string_view*

Напишем новую версию функции isPalindrom для си-строк (листинг 13.7).

Листинг 13.7. Функция-предикат «Палиндром» (версия 4)

```
bool isPalindrom(const char *s){
    auto left = 0u;                      // индекс первого символа
    auto right= strlen(s)-1;             // индекс последнего символа
    while((s[left] == s[right])&&(right > left)){
        ++left;                          // сдвигаемся к следующему
        --right;                         // сдвигаемся к предыдущему
    }
    returrn(!(right > left));            // истина, если палиндром
}
```

Этот вариант функции фактически повторяет первую версию. Отличается только вычисление индекса последнего элемента: функция strlen выдает размер си-строки. Эта версия позволяет работать со стандартными строками:

```
string ss{"12321"};
cout << isPalindrom(data(ss)) << endl;
```

Стандартная функция data работает точно так же, как и для array-массивов, — выдает адрес первого элемента массива символов внутри строки.

Можно написать функцию с параметрами-итераторами, но мы не будем этого делать. У нас получается много разных функций, делающих одно и то же с данными разных типов. Хочется написать функцию, аналогичную версии 3 (см. листинг 13.3), но такую, чтобы она могла работать и с си-строками, и со стандартными строками, и с символьными массивами.

И это можно сделать: в C++17 добавлен новый строковый тип string_view как раз для подобных задач (нужно подключить библиотеку <string_view>). Переменные этого типа — только для чтения, изменять их нельзя. Поэтому этот тип фактически применяется только для задания строковых параметров. Код новой функции показан в листинге 13.8.

Листинг 13.8. Функция-предикат «Палиндром» (версия 5, последняя)

```
bool isPalindrom(string_view s){                    // универсальный параметр
    string rs {s};                                  // преобразование
    reverse(begin(rs), end(rs));                    // реверс строки
    return (s == rs);                               // преобразования не нужно
}
```

Вызов этой функции возможен с самыми разными вариантами аргументов:

```
cout << isPalindrom("12321") << endl;           // си-строка
char ch[]="123454321";
cout << isPalindrom(ch) << endl;                // символьный массив
cout << isPalindrom("123454321"s) << endl;      // string-литерал
string s{"098767890"};                          // инициализация си-строкой
cout << isPalindrom(s) << endl;                 // стандартная строка
string_view sv{"123454321"};                    // инициализация си-строкой
cout << isPalindrom(sv) << endl;                // sv-строка
cout << isPalindrom("123454321"sv) << endl;     // sv-литерал
```

В этом коде показаны вызовы последней версии функции `isPalindrom` со всеми возможными в C++17 типами строк. В коде мы видим сразу несколько интересных нюансов:

- литерал `"123454321"s` — литерал типа `string`;
- литерал `"123454321"sv` — литерал типа `string_view`;
- можно объявлять строковые переменные типа `string_view`;
- строки любого типа неявно преобразуются в `string_view`.

Строковые литералы с суффиксами реализованы в стандартной библиотеке с помощью специальной конструкции user-defined-literal, которую включили в C++11. Суффикс `s` означает, что этот литерал имеет тип `string`, а суффикс `sv` говорит о том, что это литерал типа `string_view`. Поэтому в дальнейшем будем строковые литералы использовать строго по типам строк:

- без суффикса — для си-строк;
- с суффиксом `s` — для стандартных строк;
- с суффиксом `sv` — для sv-строк.

Хотя я регулярно забываю, что литерал без суффикса — это встроенный символьный массив и при его использовании для стандартных строк всегда выполняется преобразование типа. Это не доставляет проблем, если нам не требуется обрабатывать большие массивы строк.

Размеры строк STL

Выше мы разобрались, сколько памяти занимает символьный литерал (который является си-строкой): его размер на единицу больше количества видимых симво-

лов. И размер символьного массива должен быть на единицу больше хранимой в нем си-строки.

А вот размер реализованных строк и соответствующих литералов никак не связан с количеством символов. Например, следующие операторы выдают один и тот же размер памяти, но разное количество символов:

```
cout << size(""sv) << ' ' << sizeof(""sv) << '\n';
cout << size("1"sv) << ' ' << sizeof("1"sv) << '\n';
cout << size("2222"sv)<< ' ' << sizeof("2222"sv) << '\n';
cout << size("333333333333333333"sv) <<' '
     << sizeof("333333333333333333"sv) << '\n';
```

Получаем вывод в консоли:

```
0 16
1 16
4 16
20 16
```

Первое число — это количество символов в sv-литерале, а второе — размер выделенной памяти в байтах. Количество символов в литералах различно, но памяти выделено одинаково. Те же 16 байт выделяется и для sv-переменных независимо от длины хранимой строки.

Для стандартных строк выдаются похожие результаты:

```
0 32
1 32
4 32
20 32
```

Объект стандартной строки занимает в памяти 32 байт независимо от количества символов в строке. Я не привел пример с литералом длиннее 32 символов, но можете поверить, что и в этом случае второе число равно 32.

Налицо некоторое противоречие: короткие последовательности символов занимают столько же памяти, сколько и длинные. Более того, даже если количество символов больше 16, то `sizeof` все равно выдает размер 16. Аналогичная картина и для стандартных строк, только `sizeof` выдает 32.

Получается, что сами последовательности символов хранятся не в объектах строкового типа, а где-то в другом месте. Раскрою секрет: сами строки хранятся в *динамической памяти*, которую программа может получить во время выполнения. Естественно, тут не обходится без указателей. Мы научимся работать с динамической памятью — это очень важная часть C++ и стандартной библиотеки.

Числа — прописью

Напишем программу, которая еще недавно была чрезвычайно востребована в реальной жизни: вывод числа прописью. Числа прописью пишутся в каждом платежном документе банка.

Мы не будем писать совсем универсальный вариант: ограничим наши числа положительными целыми в диапазоне от 0 до 999'999'999'999'999 — такие числа помещаются в переменную типа `uint64_t`.

Число разбито на тройки цифр, т. к. числительные от 1 до 999 пишутся одинаково, независимо от местоположения. Вся разница — в слове после очередной тройки: самая старшая — триллионы, следующая — миллиарды, средняя — миллионы, предпоследняя — это тысячи, а младшая — рубли. Поэтому естественно написать функцию, которая получает число от 1 до 999 и выдает результат типа `string`. В этом случае мы сможем инкапсулировать все массивы числительных в этой функции.

Однако миллионы (миллиарды, триллионы) и рубли у нас мужского рода, а тысячи — женского. Поэтому вместо «один» и «два» надо писать «одна» и «две». Это — одна из тех мелочей, которые на каждом шагу встречаются в реальном программировании, из-за которых невозможно написать короткую универсальную функцию. В остальном числительные полностью совпадают. Рассмотрим на примере миллионов, как надо писать это слово после числа:

- 1 — миллион;
- 2, 3, 4 — миллиона;
- 5 – 9, 10, 11 – 19, 20 – 90 — миллионов.

Совершенно точно так же записываются миллиарды и триллионы.

Очевидно, что можно написать короткую функцию, которая будет добавлять к слову «миллион» (миллиард, триллион) нужное окончание.

Теперь рассмотрим, как пишутся тысячи:

- 1 — тысяча;
- 2, 3, 4 — тысячи;
- 5 – 9, 10, 11 – 19, 20 – 90 — тысяч.

Тысячи в числе встречаются один раз — их придется обрабатывать отдельно.

Таким образом, требуется грамотно распределить работу между основной функцией и функцией, обрабатывающей очередную тройку цифр. Назовем эту функцию `toTread`. Основная функция должна разделить число на пять троек и пять раз вызвать функцию `toTread`, передавая ей очередную тройку и слово после тройки. После пятого вызова основная функция добавит слово «рубли» в нужной форме. Код основной функции показан в листинге 13.9.

Листинг 13.9. Основная функция «Числа — прописью»

```
string Numeral(uint64_t number)
{  uint64_t t = number;                          // просто переменная
    if(t == 0) return "ноль рублей";             // сразу выходим
    uint16_t treads[5];                          // вычисление троек числа
    treads[0] = t % 1000;                        // единицы
    treads[1] = t / 1000 % 1000;                 // тысячи
    treads[2] = t / 1000'000 % 1000;             // миллионы
```

```
    treads[3] = t / 1000'000'000 % 1000;        // миллиарды
    treads[4] = t / 1000'000'000'000;           // триллионы
    string result;              // формируем строку прописью
    if(treads[4] > 0) result =  toTread(treads[4], "триллион");
    if(treads[3] > 0) result += toTread(treads[3], "миллиард");
    if(treads[2] > 0) result += toTread(treads[2], "миллион");
    if(treads[1] > 0) result += toTread(treads[1], "тысяч");
    if(treads[0] > 0) result += toTread(treads[0], "");        //рубли
                                                               // добавляем рубли
    if (treads[0] / 10 % 10 == 1) result += "рублей";
    else switch (treads[0] % 10)
    {  case 0: case 5: case 6:
       case 7: case 8: case 9: result += "рублей"; break;
       case 2: case 3: case 4: result += "рубля";  break;
       case 1: result += "рубль";                  break;
    }
    return result;
}
```

Обратите внимание на фрагмент кода, в котором к результату добавляются рубли — похожим образом должны добавляться и миллионы (миллиарды, триллионы, тысячи). Напишем эту функцию (код показан в листинге 13.10).

Листинг 13.10. Добавление окончания к словам

```
string Word(uint16_t tread, string word){
    if (tread / 10 % 10 == 1) word += "ов";
    else switch (tread % 10)
    {  case 0: case 5: case 6:
       case 7: case 8: case 9: word += "ов"; break;
       case 2: case 3: case 4: word += "а";  break;
       // 1 - ничего не делаем --
    }
    return (word + ' ');
}
```

Ну и наконец, рассмотрим функцию `toTread`, которая превращает тройку цифр в число прописью. В ней инкапсулированы массивы числительных:

```
string_view units[] = {"","один", "два",…, "восемь", "девять"
};
string_view tens[] = {"","","двадцать", "тридцать",…,"девяносто"
};
string_view hundreds[] = {"","сто", "двести", "триста",…,"девятьсот"
};
string_view secondten[] = {"десять", "одиннадцать",…,"девятнадцать"
};
string_view unitt[] = {"","одна", "две",…,"восемь", "девять"};
```

Массивы объявлены как sv-строки только из-за меньшего объема памяти. С другой стороны, последний массив повторяет первый, кроме первых двух элементов. Я так сделал сознательно, чтобы не добавлять в код функции дополнительных проверок. Код функции показан в листинге 13.11.

Листинг 13.11. Функция формирования числительного для тройки цифр

```
string toTread(uint16_t number, string_view word)
{ // массивы числительных
    string result;                    // пустая строка-результат
    uint8_t digits[3]{};              // цифры числа
    digits[0] = number % 10;          // младшая цифра
    digits[1] = number / 10 % 10;     // средняя цифра
    digits[2] = number / 100;         // старшая цифра
    if (digits[2] > 0)                          // сотни
        result += string(hundreds[digits[2]])+' ';
    if (digits[1] == 1){                        // десятки = 1,
        result += string(secondten[digits[0]])+' ';
    } else if(digits[1] > 1){                   // десятки!= 1
        result += string(tens[digits[1]])+' ';
    }
    if ((digits[1] != 1)&&(digits[0] > 0)){     // единицы
        if(word == "тысяч"sv)
            result += string(unitt[digits[0]])+' ';    // тысячи
        else
            result += string(units[digits[0]])+' ';    // не тысячи
    }
    if(size(word) > 0){                         // если есть второй аргумент
        string cword{word};
        if(cword != "тысяч"s)
            result += Word(number, cword);
        else{ switch (number % 10)
              { case 2: case 3: case 4: cword += 'и'; break;
                case 1: cword += 'а';                 break;
              };
              result += (cword + ' ');
        }
    }
    return result;
}
```

В целом получилась относительно небольшая программа, в которой каждая функция занимается своим делом.

Тестирование программы

Теперь нужно протестировать программу. Тестирование нужно выполнять и для функции `toTread`, и для основной функции `Numeral`. Делать это должна вызывающая

функция, и для этой роли подойдет, как обычно, функция `main`. Но еще лучше написать специальную тестирующую функцию, которую и вызвать в функции `main`. Так как проверять надо две реализованные функции, то и тестирующих функций должно быть две: `testTread` для тестирования `toTread` и `testNumeral` для тестирования `Numeral`.

Сначала нужно составить набор тест-кейсов для тестирования `toTread`. И отдельный набор тест-кейсов нужен для тестирования `Numeral`. Составьте тест-кейсы самостоятельно и напишите тестирующие функции. Напомню, что нужно использовать макрос `assert`.

Длинные-длинные числа

В языке программирования Python целые числа не ограничиваются никаким диапазоном. В языке Java тоже есть длинные целые числа `BigInteger`. В C++ таких чисел нет, но их можно реализовать. Реализация таких чисел — это, в первую очередь, реализация операций с ними. Давайте реализуем функцию сложения беззнаковых целых произвольной длины.

Первый вопрос, который требует ответа: в каком виде можно представлять эти числа? В C++ можно представить их в виде строки символов; результат, естественно, тоже представляется как строка. Определим синоним типа:

```
using biginteger = std::string;
```

Тогда прототип функции можно сделать таким:

```
biginteger add(const biginteger &a, const biginteger &b);
```

Теперь рассмотрим некоторые особенности операции сложения. Хотя мы все с детства привыкли складывать два числа столбиком, реализация этого алгоритма требует внимательности. И новый вопрос, который приходит в голову: а как же складывать символы?

Символы — это числа

Естественно, сами символы складывать нельзя. Но вспомним, как мы выше выводили числовые значения символов `'0'` и `'\0'`:

```
cout << int('0') << ' ' << int('\0') << endl;
```

Слагаемые в функции `add` — это последовательности символов-цифр типа `char`. Выведем коды символов-цифр в консоль:

```
for(auto ch:{'0','1','2','3','4','5','6','7','8','9'})
    cout << int(ch) << ' ';
```

В операторе `for` в качестве контейнера задан объект типа `initilizer_list`. Получим последовательность чисел:

```
48 49 50 51 52 53 54 55 56 57
```

Если мы от кода каждого символа-цифры отнимем код нуля `48`, то получим набор десятичных цифр:

```
0 1 2 3 4 5 6 7 8 9
```

Вычесть код нуля из символа язык C++ позволяет, т. к. тип `char` еще со времен языка C относится к интегральным типам. Это делается так:

```
for(auto ch:{'0','1','2','3','4','5','6','7','8','9'})
    cout << int(ch -'0') << ' ';
```

Выражение `(ch - '0')` представляет собой целую цифру, если в переменной `ch` находился символ-цифра. Для того чтобы получить из цифры символ, нужно к цифре прибавить тот же символ `'0'`.

Функция сложения

Прежде чем писать функцию сложения длинных чисел, надо указать на некоторые тонкости. Во-первых, сложение столбиком двух длинных чисел — это цикл по символам-цифрам. В этом цикле удобнее к большему числу (более длинное) прибавлять меньшее (короткое).

Во-вторых, когда две цифры складываются, то результат может быть больше 10, и возникает перенос в следующий разряд числа. Поэтому при сложении цифр нужно прибавлять и перенос. Для самых младших цифр исходный перенос равен нулю, а потом он вычисляется при каждом сложении. Цикл по большему числу удобнее из-за переноса, который может «пробежать» до старшей цифры большего числа и может добавить одну старшую цифру (единицу) к результату.

В-третьих, строку-число мы пишем слева направо, а складывать нужно с младшей цифры, т. е. справа налево.

Теперь мы можем написать функцию сложения. Не будем проверять корректность входных параметров, чтобы не отвлекаться от главной задачи. Код функции представлен в листинге 13.12.

Листинг 13.12. Функция сложения длинных чисел

```
biginteger add(const biginteger &a, const biginteger &b){
    uint8_t cf = 0;                     // перенос
    uint8_t digit;                      // цифра короткого
    biginteger lesser = (size(a) < size(b))?a :b; // короткое
    biginteger larger = (size(a) < size(b))?b :a; // длинное
    string result = larger;                       // результат
    auto next_lesser = rbegin(lesser);            // справа налево
    auto next_result = rbegin(result);            // справа налево
    while(next_result != rend(result)){           // по длинному числу
       if(next_lesser != rend(lesser)){           // короткое еще есть
           digit = (*next_lesser -'0');           // цифра короткого
           ++next_lesser;                         // к следующей цифре
       } else digit = 0;                          // цифр короткого нет
       uint8_t ds = (*next_result - '0') + digit + cf;
       cf = ds / 10;                              // текущий перенос
       *next_result = (ds % 10) + '0';            // цифра результата
       ++next_result;                             // к следующей цифре
```

```
    }                                          // конец цикла
    if(cf > 0) {                               // перенос из старшего разряда
       digit = (cf + '0');                     // дополнительная цифра
       result = char(digit) + result;
    }
    return result;
}
```

Функция относительно небольшая и в общем понятная. Работа по строкам, как было указано выше, выполняется с помощью обратных итераторов. При обработке каждого символа он преобразуется в цифру, а для размещения цифры в строке результата помещения она опять преобразуется в символ. Если закончились цифры в коротком числе, то складывается ноль. После окончания цикла проверяется перенос и добавляется старшая единица, если перенос не нулевой.

Очевидно, что для тестирования функции сложения нужно писать функцию тестирования с использованием макроса `assert`. Набор тест-кейсов должен включать пары чисел, от нулей до очень больших, которые не помещаются в стандартный целый тип `uint64_t`.

В качестве примера тестов приведем несколько проверок:

```
assert(add("0"s,"0"s) == "0"s);
assert(add("0"s,"10"s) == "10"s);
assert(add("2"s,"3"s) == "5"s);
assert(add("9"s,"2"s) == "11"s);
assert(add("17"s,"944"s) == "961"s);
assert(add("999"s,"99"s) == "1098"s);
assert(add("2"s,"1000000000009"s) == "1000000000011"s);
```

Все тесты проходят без ошибок. Можно написать числа еще длиннее (больше 20 цифр), но такой код не поместится по ширине страницы книги.

Задачи и упражнения

1. Напишите тестовые функции `testTread` и `testNumeral`.
2. Напишите тестовую функцию для функции сложения длинных чисел.
3. Измените функцию сложения длинных чисел так, чтобы параметрами были sv-строки.
4. Напишите функцию с параметрами даты (день, месяц, год), которая возвращает дату в виде строки. Например:

   2, 5, 2024 — второе мая две тысячи двадцать четвертого года.
5. Напишите функцию, которая получает параметры (часы, минуты) и возвращает время как строку. Например:

   23, 31 — двадцать три часа тридцать одна минута.

- Урок 14 -

Снова о функциях

Мы уже умеем писать функции (см. *урок 7*). И даже пробовали писать конструкторы и методы в классе (см. *урок 10*). Однако нам предстоит еще многому научиться. Средства определения функций в C++ очень обширны, и нам еще предстоит изучить лямбда-выражения и лямбда-функции, функции с параметрами по умолчанию, constexpr-функции и т. д. Интересно, что несмотря на прямой запрет иметь вложенные функции, в C++ существует несколько способов обойти это ограничение. И этому мы тоже научимся.

Однако сначала нужно разобраться с более простыми вопросами: что такое рекурсивные функции, и как взаимодействуют функции с массивами.

Рекурсивные функции

Язык C++, как практически все языки программирования, предоставляет возможность написания рекурсивных функций. Однако только программист решает, применять рекурсию или нет. Как правило, рекурсивные алгоритмы возникают там, где имеется явное рекурсивное определение данных.

Например, наибольший общий делитель (НОД) традиционно вычисляется посредством реализации алгоритма Эвклида, который определяется так:

- $НОД(a,0) = a$ (14.1)
- $НОД(a,b) = НОД(b, a \bmod b)$

Выражение `a mod b` означает «по модулю». Такое определение называется *рекуррентной* формулой. Подобных формул в математике достаточно много. Например, существует математическое определение факториала, в котором используется рекуррентная формула:

- $0! = 1$ (14.2)
- $\forall n > 0\ n! = n \times (n - 1)!$

Если обозначить функцию факториала не восклицательным знаком, а буквой *F*, как это обычно принято в математике, то определение будет выглядеть так:

- $F(0) = 1$ (14.3)
- $\forall n > 0\ F(n) = n \times F(n - 1)$

На заметку

Значок ∀ (буква А перевернутая) называется «функтор общности» и в этой формуле читается «для любого n».

Эта формула означает: для любого целого положительного числа `n` функция факториала вычисляется как произведение `n` и значения той же функции, вычисленной для `(n-1)`. Понятно, что для вычисления значения факториала для `(n - 1)` надо умножить `(n - 1)` на значение функции от `(n - 2)`. И т. д. А на каком-то очередном шаге получится, что надо вычислить значение функции от `0`, и это значение равно `1`.

Таким образом, при вычислении факториала числа `n` вычисляется целая последовательность предыдущих факториалов. При этом для вычисления любого `k!` нужно вычислить предыдущий член последовательности `(k-1)!`. Подобные формулы вычисления последовательностей, в которых для вычисления k-го элемента используется один или несколько предыдущих элементов, и называются рекуррентными формулами.

По определению (формулы 14.2, 14.3) функция для вычисления факториала пишется прямо «в лоб» (листинг 14.1).

Листинг 14.1. Рекурсивная функция вычисления факториала

```
unsigned long Fact(unsigned long k)
{  if (k == 0) return 1u;                    // первая строка формулы
   return (k * Fact(k - 1));                 // рекурсивный вызов
}
```

В теле функции вызывается сама эта функция со значением аргумента, на единицу меньше исходного, — в полном соответствии с рекуррентным соотношением в формуле. Такой вызов называется рекурсивным, а функция, в которой присутствует подобный вызов, является *рекурсивной*. Рекурсия реализует вторую строку в определении, т. е. рекуррентную формулу. Но в функции присутствует и первая строка определения как проверка условия в условном операторе.

Рекурсивной функцией называется функция, вызывающая саму себя в своем теле. Необходимо подчеркнуть, что «самовызов» будет рекурсивным только в том случае, если находится в теле функции.

Обычно различают прямую и косвенную рекурсию. Если в теле функции явно используется вызов той же самой функции, то имеет место *прямая рекурсия*, как в приведенных примерах. Если две или более функций взаимно вызывают друг друга, то имеет место *косвенная рекурсия*. Это более сложный вариант рекурсии, который часто возникает при реализации системных программ.

По рекуррентной формуле 14.1 тоже легко написать функцию вычисления наибольшего общего делителя. Код представлен в листинге 14.2.

Листинг 14.2. Рекурсивная функция НОД

```
uint64_t nod(uint64_t a, uint64_t b){
    return (b == 0)? a : nod(b, a % b);
}
```

Как видите, эта функция вообще состоит из одного оператора возврата. Но первая строка рекуррентной формулы 14.1 все равно присутствует в виде условной операции. Аналогично в одну строку можно написать и функцию вычисления факториала:

```
uint64_t fact(uint64_t n){
    return (n == 0)? 1 : n*fact(n-1);
}
```

Мы знаем, что в программировании факториал вычисляют как произведение `n` целых чисел:

```
n! = 1 * 2 * 3 * ... * n
```

Такое произведение можно записать с помощью цикла `for`, например:

```
for (f = 1, i = 1; i < n; i++) f *= i;
```

Такое решение называется *итеративным*. Для факториала эта конструкция выглядит проще, чем рекуррентная формула. Однако, как правило, рекуррентное соотношение проще, чем какие-то другие формулы. Например, попробуйте написать цикл для вычисления НОД — не так-то просто это сделать по сравнению с написанием функции по формуле 14.1 (листинг 14.3).

Листинг 14.3. Итеративная функция НОД

```
uint64_t nod(uint64_t a, uint64_t b){
    while(b != 0){
        uint64_t r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

Даже без детального анализа видно, что код этой функции сложнее, чем рекурсивной.

Довольно часто рекуррентное определение просто хорошо известно, а другие формулы — нет. Например, все прекрасно помнят рекуррентную формулу для вычисления чисел Фибоначчи:

- $F(0) = 0$
- $F(1) = 1$ (14.4)
- $\forall\ n > 2\ F(n) = F(n-1) + F(n-2)$

А прямая формула Бине для вычисления чисел Фибоначчи известна очень немногим. Да и написать функцию по формуле Бине не проще, а даже сложнее, чем по

рекуррентной. Рекурсивная функция пишется прямо по формуле 14.4 (код приведен в листинге 14.4).

Листинг 14.4. Рекурсивная функция вычисления чисел Фибоначчи

```
uint64_t Fibo(uint64_t k)
{  if (k == 0) return 0;
   if (k == 1) return 1;
   return (Fibo(k - 1) + Fibo(k - 2));     // рекурсивный вызов
}
```

В такой функции ошибиться просто невозможно. Единственное, о чем нужно подумать, — это тип возвращаемого результата (и, конечно, параметра). Числа Фибоначчи растут очень быстро, поэтому тип `int` слишком мал.

Осваиваем рекурсию

Есть большое количество традиционных «игрушечных» задач (ханойские башни, расстановка ферзей и т. п.), которые анализируются в литературе для демонстрации рекурсии. Мы не будем их рассматривать (можете сами почитать про них в Интернете и книжках). Гораздо интереснее реализовать рекурсивным способом типичные итеративные алгоритмы. В принципе, любой цикл можно написать в виде рекурсивной функции, которая делает ровно то же самое. Обратное тоже верно: любую рекурсивную функцию можно представить в виде цикла (но иногда это довольно сложно сделать).

В качестве первого примера покажем рекурсивную реализацию функции вычисления длины си-строки (см. листинги 13.4 и 13.5). Код функции показан в листинге 14.5.

Листинг 14.5. Рекурсивная функция вычисления длины си-строки (версия 1)

```
unsigned int Length (const char *s)
{  if (*s == 0) return 0;                    // условие окончания
   else return 1 + Length(s + 1);            // рекурсивный вызов
}
```

Вызов такой функции абсолютно ничем не отличается от вызова аналогичной итеративной, например:

```
cout << Length("1234567890");
```

На экран совершенно правильно выводится `10`.

Работа рекурсивных функций обычно имеет несколько «мистический» оттенок, поэтому не будем пока в деталях разбирать работу этой функции, но обратим внимание на несколько важных моментов:

- цикла в функции нет — вместо него у нас имеется рекурсивный вызов. Таким образом, явное повторение заменяется неявным — рекурсивным;

- параметр-указатель в рекурсивном вызове увеличивается, перемещаясь к следующему символу;
- чтобы такое увеличение не происходило до бесконечности, в операторе `if` проверяется достижение конца строки (проверка завершающего нуля), и в этом операторе отсутствует рекурсивный вызов.

Такое условие является условием *завершения* рекурсии. Посмотрите коды функций в листингах 14.1, 14.2 и 14.3 — там первым оператором всегда задан оператор `if`, в котором проверяется условие завершения рекурсии. И в рекуррентных формулах 14.1–14.4 первой строкой заданы условия завершения вычисления последовательности чисел.

Но функцию вычисления длины можно написать иначе (листинг 14.6).

Листинг 14.6. Рекурсивная функция вычисления длины си-строки (версия 2)

```
unsigned int Length (const char *s)
{ if(*s)                                    // условие продолжения
     return 1 + Length(s + 1);              // продолжение рекурсии
  else
     return 0;                              // завершение рекурсии
}
```

В этом коде мы видим те же аналогичные предыдущей версии особенности реализации: вместо цикла — рекурсивный вызов; параметр в рекурсивном вызове изменяется. Однако рекурсивный вызов задан в условном операторе: если условие истинно, то рекурсия продолжается — функция вызывает сама себя, если очередной символ — не ноль (не является нуль-байтом). Можно сказать, что условие завершения рекурсии заменено условием *продолжения*.

Аналогичным образом можно переписать и другие функции, написанные ранее. Например, функцию вычисления факториала можно написать так (листинг 14.7).

Листинг 14.7. Рекурсивная функция вычисления факториала (версия 2)

```
uint64_t Fact(uint64_t k)
{  if (k > 0)                               // условие продолжения
      return (k * Fact(k - 1));             // рекурсивный вызов
   else
      return 1;
}
```

Эти примеры показывают, что рекурсивные функции, выполняющие одну и ту же работу, могут быть написаны в разных формах. В каком виде их писать, зависит от задачи и предпочтений программиста.

Формы рекурсивных функций

В общем случае любая рекурсивная функция `Func` включает в себя некоторое множество операторов `S` и один или несколько операторов рекурсивного вызова `Func`. Как мы видели на примерах, рекурсивный вызов обязательно сопровождался некоторым условием: либо условием окончания рекурсии, либо условием продолжения. Это обязательно, т. к. безусловные рекурсивные вызовы сродни бесконечному циклу. Вспомните детский стишок «У попа была собака…» — это как раз случай бесконечной рекурсии. Если написать вывод этого стишка как рекурсивную функцию, то она может выглядеть так:

```
void PriestAndDog()
{  cout << "У попа была собака, он ее любил.\n";
   cout << "Она съела кусок мяса, он ее убил,\n";
   cout << "вырыл яму, закопал и на камне написал:\n";
   PriestAndDog();
}
```

Теоретически работа этой функции никогда не завершится.

> ***Примечание***
>
> На самом деле программа закончится аварийно по причине переполнения стека (можете проверить). О стеке мы уже упоминали при изучении функций (см. *урок 7*).

Следовательно, главное требование к рекурсивным функциям заключается в том, что вызов рекурсивной функции должен выполняться по условию. Это условие указывается в операторе `if` и является либо условием продолжения, либо условием окончания рекурсии. В первом случае в операторе `if` пишется рекурсивный вызов, во втором — оператор возврата. Функция должна во время работы как-то изменять данные, по которым вычисляется условие. Если она этого не делает, то условие остается неизменным и рекурсия становится бесконечной.

Структура рекурсивной функции может принимать три разных формы (используем условие продолжения):

1. Форма с выполнением действий до рекурсивного вызова (на рекурсивном *спуске*): `void Rec() { S; if (условие) Rec(); }`.
2. Форма с выполнением действий после рекурсивного вызова (на рекурсивном *подъеме*): `void Rec() { if (условие) Rec(); S; }`.
3. Форма с выполнением действий как до, так и после рекурсивного вызова: `void Rec() { S1; if (условие) Rec(); S2; }`.

В образцах указан самый простой заголовок:

```
void Rec()
```

поскольку в данном случае нам важно было продемонстрировать именно формы рекурсивных функций, не отвлекаясь на другие детали. Кроме того, мы не писали оператор возврата, предполагая, что он неявно присутствует.

Сочетания «рекурсивный спуск» и «рекурсивный подъем» связаны с понятием *глубины* рекурсии. В описании рекурсивных функций подобные выражения являются устойчивыми идиомами: функция (при вызове) «спускается» на глубину рекурсии и «поднимается» (возвращается) оттуда.

Глубина рекурсии — это число рекурсивных вызовов функции до первого возврата. Можно сказать по-другому: глубина рекурсии — это максимальная степень вложенности рекурсивных вызовов. В общем случае глубина будет зависеть от входных данных (параметров), т. к. именно от них зависит условие продолжения (завершения) рекурсии.

Для полноты картины напишем те же формы рекурсивных функций с условием завершения:

1. `void Rec() { S; if (условие) return; else Rec(); }`
2. `void Rec() { if (условие) return; else Rec(); S; }`
3. `void Rec() { S1; if (условие) return; else Rec(); S2; }`.

Здесь уже оператор возврата прописан в явном виде, т. к. именно он завершает рекурсивный «спуск».

Выполнение рекурсивных функций

Несмотря на то, что запись рекурсивных функций бывает очень короткая, они часто «страдают» неэффективностью. Вспомним, что при каждом вызове в стеке должны быть размещены все параметры и локальные переменные (см. *урок 7*). На эту работу расходуется время — параметры надо скопировать в стек, а после выполнения работы стек нужно освободить.

Однако эта неэффективность не очень существенна по сравнению с другой — расходом памяти под стек. Поскольку рекурсивный вызов выполняется до окончания выполнения функции — размер стека увеличивается при каждом вызове до тех пор, пока не будет достигнута точка, где выполняется возврат. Размер стека пропорционален глубине рекурсии. В принципе, это может привести к *переполнению* стека и аварийному завершению программы. Поэтому при разработке рекурсивных функций необходимо минимизировать количество и размеры локальных переменных и параметров.

Рассмотрим пример, который наглядно покажет нам, почему появилось такое выражение, как «глубина рекурсии». Добавим оператор вывода адреса параметра в функцию вычисления факториала (листинг 14.8).

Листинг 14.8. Функция вычисления факториала (с выводом адресов)

```
uint64_t fact(uint64_t n){
    cout << n << ' ' << &n << endl;            // вывод адреса
    return (n == 0)? 1 : n*fact(n-1);
}
```

При запуске этой функции при `n = 3`:

```
cout << fact(3);
```

покажет в консоли картину, представленную на рис. 14.1.



Рис. 14.1. Вывод адресов параметра в стеке функции факториала

Адреса, как мы уже знаем, выводятся в шестнадцатеричном виде. Мы видим, что три младшие цифры адресов перед выводом результата уменьшаются (`940`, `910`, `8e0`, `8b0`). Происходит как бы «спуск» в «глубину».

Таким образом, при выполнении рекурсивных вызовов все аргументы попадают в стек при каждом вызове. Если аргументов много и рекурсия глубокая, то можно нарваться на ошибку переполнения стека. С размером памяти для стека мы разберемся при изучении динамической памяти.

Рекурсивные функции и массивы

Мы привыкли обрабатывать массивы в цикле. Но рекурсивные функции открывают нам новые интересные возможности. Давайте напишем функцию вывода в консоль массива строк в рекурсивном виде. Пусть у нас есть массив:

```
string numbers[]{"один", "два", "три", "четыре", "пять"};
```

Параметры функции — два итератора-указателя (листинг 14.9).

Листинг 14.9. Рекурсивная функция вывода массива

```
void Print(const string *first, const string *last){
   if(first != last){
      cout << *first << " ";          // действия на спуске
      Print(first+1, last);           // двигаемся дальше
   }
}
```

Функция написана с условием продолжения в первой форме: действия выполняются перед рекурсивным вызовом, т. е. на рекурсивном спуске. Согласитесь, что код такой функции проще, чем с циклом.

Вызов этой функции самый обычный:

```
Print(begin(numbers), end(numbers));
```

В консоль будет выведен массив `numbers` (рис. 14.2).



Рис. 14.2. Вывод массива в консоль рекурсивной функцией `Print`

Допустим, нам потребовалось, чтобы массив выводился в обратном порядке. Требуется модифицировать код нашей функции. Первая мысль, возникающая в голове программиста, который обычно пишет циклы: нужно выводить элементы массива, начиная с последнего, двигаясь в начало. Эту мысль можно реализовать и в рекурсивной функции:

```
void Print(const string *first, const string *last){
    cout << *last << " ";                    // выводим последний
    if(first != last){
        Print(first, last-1);                // двигаемся к началу
    }
}
```

И вызов тоже придется изменить:

```
Print(begin(numbers), end(numbers)-1);
```

Вроде бы изменения небольшие, но у меня получилось все правильно сделать только с третьего раза. Начинающим это сделать еще сложнее. Поэтому мы пойдем другим путем.

Рекурсия позволяет сделать нужные изменения очень просто и без проблем: надо просто поменять местами операторы в теле `if`. Тем самым мы поменяем первую форму рекурсивной функции на вторую: действия будут выполняться на рекурсивном подъеме. Код функции представлен в листинге 14.10.

Листинг 14.10. Функция вывода массива в обратном порядке

```
void Print(const string *first, const string *last){
    if(first != last){
        Print(first+1, last);          // рекурсивный спуск
        cout << *first << " ";         // действия на подъеме
    }
}
```

Вызов функции при этом изменять не требуется! Результат выполнения показан на рис. 14.3.

Выводы из этого примера простые: понимание рекурсии иногда очень упрощает программирование.



Рис. 14.3. Вывод массива в обратном порядке

Теперь займемся более серьезной работой: напишем рекурсивный вариант функции поиска минимума массива. Начнем с массивов целых чисел. Код первой версии может быть такой (листинг 14.11).

Листинг 14.11. Рекурсивная функция поиска минимума (версия 1)

```
int mini(int a[], int left, int right)          // параметры-индексы
{  if (left == right) return a[left];           // условие окончания
   int x = mini(a, left+1, right);              // идем дальше
   if (x < a[left]) return x;                   // заменяем минимум
   else             return a[left];             // не заменяем минимум
}
```

Функция имеет вторую форму — действия выполняются после вызова (на рекурсивном подъеме). Я специально написал такие параметры — это упростит мне еще одну демонстрацию глубины рекурсии.

Вызов этой функции выглядит так:

```
int numbers[]{3, 2, 4, -2, 6, -1};
auto x = mini(numbers, 0, size(numbers)-1);
cout << x << endl;
```

Функция совершенно правильно находит минимум в массиве, равный `-2`.

Мы можем посчитать количество и вложенность рекурсивных вызовов, объявив в функции `mini` статическую переменную (см. *урок 7*). Увеличивая ее при каждом вызове и уменьшая при каждом возврате, можно получить полную картину рекурсивных вызовов и возвратов. Напомню, что статическая переменная инициализируется только один раз. При следующих вызовах она сохраняет свое значение, полученное во время работы в предыдущем вызове.

Эту переменную можно передать как параметр отдельной специальной функции, которая будет выводить в консоль строку точек, длина которой равна значению этой статической переменной. Эту функцию нужно вызывать в функции поиска минимума массива перед и после рекурсивного вызова.

Такие функции, которые пишутся для исследования рабочих функций, называются *отладочными* функциями. А выполнение программы вместе с запускаемыми в рабочем коде отладочными функциями называется *трассировкой* программы. Код обеих функций представлен в листинге 14.12.

Листинг 14.12. Трассировка рекурсивной функции поиска минимума

```
void pp(int& tt, int l, int r)
{  for(int i = 0; i < tt; i++) cout << ".";
   cout << "mini(" << l << ',' << r << ')' << endl;
}
int mini(int a[], int left, int right)
{  static int tt = 0;            // счетчик уровня вложенности
   if (left == right) return a[left];
   pp(++tt, left, right);        // вложенность увеличивается
   int x = mini(a, left+1, right);
   pp(--tt, left, right);        // вложенность уменьшается
   if (x < a[left]) return x; else return a[left];
}
```

Вызов функции `mini` показан выше. Результат видим на рис. 14.4.

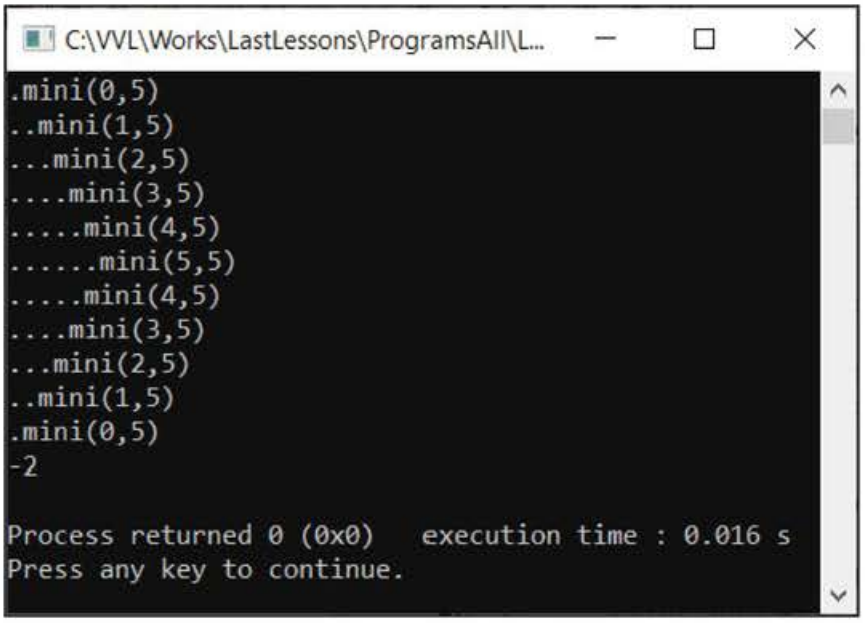


Рис. 14.4. Трассировка рекурсивной функции поиска минимума

Количество вложенных рекурсивных вызовов равно количеству элементов массива `numbers` (см. выше). Глубина рекурсии тоже равна количеству элементов. Сначала выполняется только рекурсивный спуск до последнего элемента (индекс 5). Когда достигнут последний элемент — выполняется первый рекурсивный возврат, и переменная `x` впервые получает значение. На первом шаге рекурсивного подъема переменная `x` сравнивается сама с собой и, конечно, выполняется оператор:

```
else return a[left];
```

На рекурсивном подъеме при каждом возврате переменная `x` сравнивается с предыдущим элементом массива. По количеству выведенных точек и по индексам хорошо видно, что функция выполняет ровно столько вложенных рекурсивных вызовов и сравнений, сколько у нас имеется элементов в массиве `a`. Функция эффективна по количеству операций.

Но за это приходится расплачиваться памятью — размером стека. Ведь при каждом рекурсивном вызове в стек попадает три переменных: адрес массива, индекс левого элемента и индекс правого элемента. В сумме это не менее 16 байт (адрес 8 байт и два целых по 4 байта). Для нашего короткого массива это безопасно. А если массив размером 1 000 000 (миллион) элементов, то стек займет не менее 16 миллионов байт. Это уже может привести к аварийному завершению программы.

> ***На заметку***
>
> Операционная система ограничивает размер стека при выполнении программ. Мы выясним этот параметр в одном из следующих уроков, когда будем изучать динамическую память.

Поэтому в подобных случаях иногда приходится думать, как уменьшить глубину рекурсии. Традиционным решением является деление работы на части. Например, один рекурсивный вызов обрабатывает левую половину массива (меньшие индексы), а другой — правую (большие индексы). Такой прием разделения работы на части в программировании используется часто и даже получил название «разделяй и властвуй».

Так и реализуем новую версию (код показан в листинге 14.13).

Листинг 14.13. Рекурсивная функция поиска минимума (версия 2)

```
int mini(const int a[], int left, int right)
{  if (left == right) return a[left];
   int m = (left + right) / 2;          // середина
   int x = min(a, left, m);             // левая половина массива
   int y = min(a, m+1, right);          // правая половина массива
   if (x < y) return x; else return y;
}
```

Функция имеет третью форму, выполняя действия как на рекурсивном спуске (до рекурсивного вызова), так и на рекурсивном возврате (после вызова).

Сам код, я надеюсь, понятен: вычисляется индекс середины массива, потом функция рекурсивно вызывается для левой половины и правой половины. В каждом вызове опять вычисляется середина соответствующей половины и снова вызывается функция для левой половины и правой половины. И так до тех пор, пока не выполнится сравнение и не произойдет возврат значения.

Трассировка этой функции показывает, что глубина рекурсии значительно меньше (выигрываем в памяти), но зато выполняемых операций гораздо больше (проигрываем в скорости).

Пример трассировки для массива из шести элементов показан на рис. 14.5.

Я поэкспериментировал с разными массивами и убедился, что глубина рекурсии в этой функции с увеличением размера массива растет очень медленно. Например, для массива из 30 элементов глубина рекурсии равна 6, а общее количество рекурсивных вызовов в 2 раза больше размера массива. Таким образом, функция эффек-

тивна по памяти (размер стека небольшой) и неэффективна по количеству выполняемых операций.

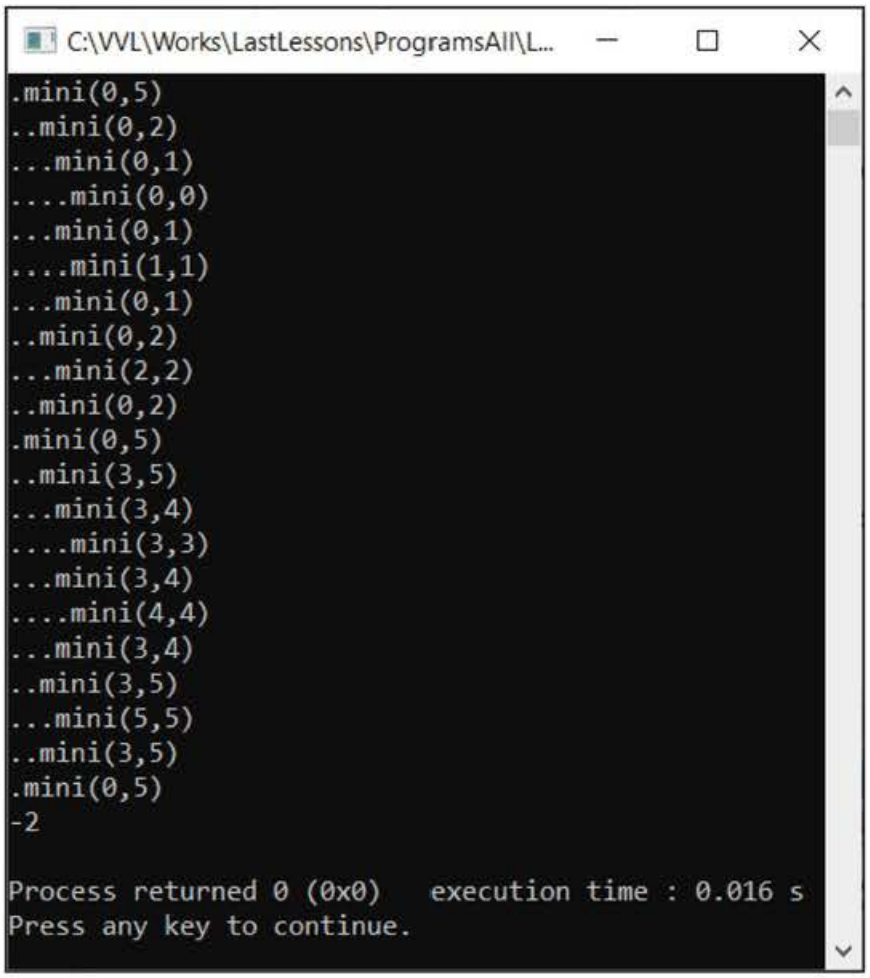


Рис. 14.5. Трассировка функции поиска минимума (версия 2)

Функции времени компиляции

Мы уже не раз использовали слово `constexpr` для объявления констант. Однако в программах с числом π (см. листинги 3.5 и 7.5) эта математическая константа объявлена с ключевым словом `const`:

```
double const pi {acos(-1.0)};                    // число π
```

Попытки объявить ее с помощью `constexpr` приводят к ошибкам во время компиляции. Получается, что константы const и константы `constexpr` — это разные виды констант. Так оно и есть на самом деле — в C++ имеется два вида констант:

- константы времени выполнения `const`;
- константы времени компиляции `constexpr`.

Первый вид констант инициализируется при выполнении программы. Именно так происходит инициализация константы π: вычисляется функция `acos` для аргумента -

1.0, и это значение присваивается объекту с именем pi. После этого объект pi изменять нельзя.

Замечу, что инициализацию во время выполнения можно осуществить даже с помощью ввода значения. Например, если у нас реализована некая функция read, то константу времени выполнения разрешается инициализировать так:

```
const auto t = read();
```

При каждом запуске программы константа t будет иметь разные значения. Более того, она может иметь даже разные типы в зависимости от того, как реализован возврат из функции read.

Константы вида constexpr должны быть вычислены во время компиляции. Поэтому они не могут быть инициализированы посредством вызова обычных функций. Константы constexpr появились в стандарте C++11, тогда же в стандарте появились и функции constexpr. Это означает, что в C++ сейчас есть функции, которые *вычисляются во время компиляции*.

Такие конструкции в C++ были включены для повышения эффективности выполняемых программ. Ведь вычисления при компиляции выполняются один раз, а вычисления во время выполнения программы обычно делаются неоднократно. Поэтому в STL многие стандартные алгоритмы постепенно переписывают в виде constexpr-функций.

Напишем constexpr-функции, чтобы познакомиться с этой возможностью C++. Мы писали программу, в которой реализовали простой алгоритм вычисления x^n (см. *урок 6*, листинг 6.7). Давайте оформим этот алгоритм в виде constexpr-функции. Код функции — в листинге 14.14.

Листинг 14.14. Функция возведения в степень (*constexpr*)

```
constexpr
double pow(double x, int n){
    if(n == 0) return 1.0;                    // любое x в степени 0
    if(x != 0){
        if(n == 1)      return x;             // любое x в степени 1
        else if(n == 2) return x*x;
        else{ double v = x;                   // общий случай
              while(--n){ v *= x; }
              return v;
        }
    }
    else if(n > 0) return 0.0;                // 0 в любой степени
}
```

Функция немного оптимизирована для отдельных случаев (для n = 0,1,2 и x = 0), чтобы не было необходимости выполнять цикл. Функция объявлена constexpr и может вызываться для инициализации constexpr-констант:

```
constexpr auto d = pow(1.5, 2);
```

В этом случае компилятор вычисляет значение, возвращаемое функцией, и присваивает константе d значение 2.25. Отмечу, что в constexpr-контекстах вместо имени d компилятор будет подставлять число 2.25. А если контекст такой, что его нельзя вычислить при компиляции, то значение d будет использоваться при выполнении.

Так же и функцию pow можно вызывать как обычную функцию, которая будет работать при выполнении программы:

```
auto dd = 1.5;
cout << pow(dd, 2) << endl;
```

Для constexpr-функций в стандарте C++11 было указано много ограничений: нельзя было писать не только циклы, но и условные операторы. Функция должна быть рекурсивной и содержать единственный оператор return (см. листинг 14.2).

Стандарт C++14 эти ограничения значительно ослабил. В частности, стало возможно писать обычный цикл, чем мы и воспользовались. Напишем шаблон рекурсивной constexpr-функции вычисления степени (листинг 14.15).

Листинг 14.15. Шаблон constexpr-функции

```
template<class T>
constexpr
T pow(T x, int n){
    if(n == 0)    return 1;
    if(x != 0){
       if(n == 1) return x;
       else if(n == 2) return x*x;
       else if(n > 0)  return x*pow(x,n-1);     // рекурсивный вызов
    }
    else if(n > 0) return 0;
}
```

Эта функция выглядит короче только из-за рекурсивного вызова. Свою работу она выполняет точно так же, как и функция в листинге 14.14. Если эти две функции написаны в одной программе, то мы имеем перегрузку шаблона и функции. Функция в листинге 14.14 работает только для аргументов типа double, а шаблон — для всех других типов, в первую очередь, целых.

А как же убедиться, что вычисления выполняются во время компиляции? Для этого в стандарт включен оператор static_accert. Напишем тесты для проверки шаблона constexpr-функции pow:

```
static_assert(pow(0.0,0)== 1, "Ошибка: pow(0.0,0)== 1");
static_assert(pow(4.0,0)== 1, "Ошибка: pow(4.0,0)== 1");
static_assert(pow(15,1) == 15,"Ошибка: pow(15,1) == 15");
static_assert(pow(5,2)  == 25,"Ошибка: pow(5,2)  == 25");
```

Первым аргументом оператора задается логическое выражение, аналогичное выражению в макросе assert (см. *урок 7*). Второй аргумент — сообщение, выдаваемое

при компиляции среди сообщений об ошибках, если логическое выражение равно `false`. Например, мы нечаянно сделали ошибку и написали такой код:

```
static_assert(pow(4.0,1)== 1, "Ошибка: pow(4.0,0)== 1");
```

Тогда при компиляции получим ошибку компиляции:

```
error: static assertion failed: Ошибка: pow(4.0,0)== 1|
```

Конечно, в полном сообщении будет указан и номер строки, в которой написан неверный `static_assert`.

Вывод подобного сообщения во время компиляции совершенно определенно показывает, что компилятор вычислил значение функции `pow(4.0,1)`, потом вычислил логическое выражение, получил `false` и вывел сообщение об ошибке. Если логическое выражение было бы равно `true` (как во всех других операторах), то никаких сообщений об ошибке не появилось бы.

Задачи и упражнения

1. Реализуйте трассировку функции поиска минимума, представленную в листинге 14.13.
2. Напишите рекурсивную функцию `isPalindrom` (см. *урок 13*).
3. Реализовать в виде рекурсивной функции задание 5 *урока 9*. Исходный массив заполнить случайными числами и вывести до и после вызова функции в главной программе.
4. Реализовать функцию суммирования массива как рекурсивную в двух вариантах:
   - с параметрами-итераторами;
   - с массивом и размером.

- Урок 15 -

Печальная действительность

До сих пор все наши программы прекрасно работали. Я специально писал исключительно корректные программы, которые выдавали правильные результаты, — чтобы не отвлекаться на решение проблем, не относящихся к делу. Но программирование полно подводных камней, о которых начинающий программист не подозревает. Как говорится, правильных программ не бывает, есть недостаточно оттестированные. Но и тестирование не может показать отсутствие ошибок, а показывает только их наличие. Поэтому программист даже начального уровня должен знать о возможных проблемах, которые могут возникнуть в его коде.

А проблем, и очень серьезных, может возникнуть столько, что становится просто интересно: а как же нам удается писать программы, которые не просто как-то работают, но еще и правильные результаты вычисляют?! В этом нам, конечно, очень помогает компилятор, выдавая предупреждения. К сожалению, начинающие программисты никогда не читают предупреждения компилятора. Только с опытом приходит понимание, что в них содержится очень важная информация о коде.

Однако и компилятор сообщает не обо всех проблемах, поэтому мы пройдемся по «подводным» камням и вытащим их из-под «воды», «покопаемся» во внутреннем устройстве чисел и научимся писать код, который меньше подвержен ошибкам.

Проблемы с целыми числами

Откроем табл. 4.1 (см. *урок 4*) и еще раз рассмотрим диапазоны целых чисел. Как уже было сказано, сумма абсолютных величин минимума и максимума знаковых целых равна максимуму беззнаковых того же размера. А если максимум и минимум знаковых просто сложить, то сумма равна -1. И подобных интересных свойств у целых довольно много.

Поэкспериментируем с целыми, чтобы разобраться в тонкостях. Начнем с самых безобидных операций: присвоим отрицательное число беззнаковой переменной и наоборот. Напишем простейший код:

```
int8_t   sbyte;                // знаковое: от -128 до +127, размер 1
uint8_t  ubyte;                // беззнаковое: от 0 до 255,  размер 1
sbyte = -1;
ubyte = -1;
cout << int(sbyte) << ' ' << int(ubyte) <<'\n' ;
```

```
sbyte = -128;
ubyte = -128;
cout << int(sbyte) << ' ' << int(ubyte) <<'\n' ;
```

В консоль выводится:

```
-1 255
-128 128
```

Мы видим очень интересное преобразование целой константы `-1` в целое беззнаковое `255`, которое является максимумом для типа `uint8_t`. Второе преобразование еще интереснее: просто исчезает знак числа `128`.

Если вы думаете, что здесь большую роль имеет размер переменных, то вы ошибаетесь. То же самое происходит и с обычными целыми типа `int32_t` и `uint32_t`. И даже с длинными типа `long long` и `unsigned long long`. Покажем это на примере обычных целых:

```
unsigned uint;
uint = numeric_limits<int>::min();
cout << uint <<'\n';
```

Этот код выведет число 2147483648, что аналогично превращению -128 в 128.

При сложении пределов диапазона знаковых целых получается `-1`, что является правильным результатом. Но присваивая `-1` беззнаковому целому, получаем, как обычно, максимум беззнакового целого, равный `4294967295`.

Справедливости ради отметим, что для универсальной инициализации компилятор выдает предупреждение о сужающем преобразовании. Однако стоит нам присвоить число `-1` каким-нибудь другим способом, и никакие предупреждения не выдаются. Преобразование типа выполняется с теми же результатами: целое число `-1` переводится в максимум соответствующего диапазона беззнаковых.

Поэтому дальше будем работать только с числами типа `short` — те же самые «фокусы» работают и со всеми другими целыми типами в пределах их диапазонов. Правда, при выводе придется каждый раз писать преобразование типа в `unsigned short`, т. к. иначе выражения расширяются до целых.

Начнем с переполнений. *Переполнение* (для целых) — это ситуация, когда результат вычислений не помещается в размеры данного типа. Какие результаты при этом получаются, зависит от размеров и диапазонов типов.

Напишем простой код:

```
int16_t sshort;                                  // знаковое
uint16_t ushort;                                 // беззнаковое
ushort = numeric_limits<uint16_t>::max();
cout << uint16_t(ushort+1) << '\n';
sshort = numeric_limits<int16_t>::max();
cout << int16_t(sshort+1) << '\n';
ushort = numeric_limits<uint16_t>::min();
cout << uint16_t(ushort-1) << '\n';
sshort = numeric_limits<int16_t>::min();
cout << int16_t(sshort-1) << '\n';
```

Этот код демонстрирует, что добавление единицы к максимальному числу (правая граница диапазона) дает в результате минимум диапазона. При вычитании единицы из минимального числа получается максимум диапазона:

```
0              // беззнаковые max+1
-32768         // знаковые    max+1
65535          // беззнаковые min-1
32767          // знаковые    min-1
```

Никаких сообщений компилятора не выводится.

Если мы просто сложим или умножим два беззнаковых числа, каждое из которых не превышает максимального, то получим интересный результат:

```
ushort = 40000+40000;
cout << ushort << '\n';
ushort = 40000*2;
cout << ushort << '\n';
```

В обоих случаях получаем результат `14464`. Но компилятор предупреждает, что в выражениях получится переполнение.

Ничего удивительного в таком результате нет, т. к. в таких выражениях для коротких беззнаковых целых результат вычисляется по модулю `65536` (остаток от деления на `65536`). В этом легко убедиться, написав оператор:

```
cout << (40000*2)%65536 << '\n';
```

В консоли появится то же самое число `14464`.

Это объясняет результаты экспериментов с границами диапазона. Выражение для коротких целых при добавлении единицы:

```
(65635+1)%65536 = 0
```

С вычитанием единицы из левой границы диапазона тоже все понятно, т. к. `0 - 1 = -1`, и результат преобразуется в правую границу диапазона. Однако нужно разобраться, что в принципе происходит при вычитании в ситуации, когда вычитаемое больше уменьшаемого.

```
ushort = 10 - 45;
cout << ushort << '\n';
```

Результат выполнения этого кода — число `65501`. Эксперименты с похожими выражениями позволяют понять, что вычисления делаются по формуле:

```
cout << (10-45)%65536+65536 << '\n';
```

Выводится то же самое число `65501`. Подставив в формулу значения `0` и `1`, получим объяснение преобразования `-1` в максимум беззнакового диапазона:

```
cout << (0-1)%65536+65536 << '\n';
```

Вспомним, как работает операция деления по модулю % для отрицательного делимого (см. листинг 4.3). Выражение `(0-1)%65536` дает в результате `-1`, что дает окончательный результат (верхняя граница беззнаковых целых).

Разберемся, что происходит в таких же ситуациях со знаковыми целыми:

```
sshort = 30000+30000;
cout << sshort << '\n';
```

В этом случае выводится число -5536. Компилятор, как и в предыдущих примерах, выводит предупреждение. Но вызывает интерес то, что получилось отрицательное число при сложении положительных. Эксперимент позволяет понять, что результат получается по такой формуле:

```
cout << (30000+30000)%65536-65536 << '\n';
```

Этот оператор тоже выводит число -5536.

При работе с отрицательными целыми мы можем получить положительное значение, например:

```
sshort = -30000-10000;
cout << sshort << '\n';
```

Результат этого выражения равен 25536. Тот же результат получаем вот так:

```
cout << (-30000-10000)%65536+65536 << '\n';
```

Это фактически та же формула, что и для беззнаковых целых.

Еще одна интересная ситуация возникает, если попытаться изменить знак у минимального знакового целого (-32768):

```
cout << int16_t(-numeric_limits<int16_t>::min()) << '\n';
```

Это число так и остается отрицательным: -32768.

Все то же самое можно наблюдать для целых типов int и long, только модуль равен 4294967296. Для типа long long модуль, соответственно, равен 18446744073709551616. Как можно понять, модуль на единицу больше, чем беззнаковый максимум того же размера.

Все рассмотренные выше ситуации не приводят к аварии программы.

Если программист не обратил внимания на возможность возникновения подобной ситуации, то программа, на первый взгляд, будет работать, но результаты будут неправильными.

На ноль делить нельзя

Ну и надо еще посмотреть, что же происходит при делении на ноль для целых. Напишем в коде самый простой оператор:

```
cout << (1/0) << endl;
```

Компилятор выдал предупреждение:

```
warning: division by zero [-Wdiv-by-zero]
```

Предупреждение прямо говорит о том, что произойдет деление на ноль. Запустим программу (рис. 15.1).

Никаких результатов не выводится — это уже вызывает подозрение, что программа завершилась как-то не совсем нормально. Это подтверждает и сообщение в консоли. В нашей программе был написан оператор возврата return 0;. Однако в сообщении говорится, что процесс вернул не ноль, а число -1073741676.

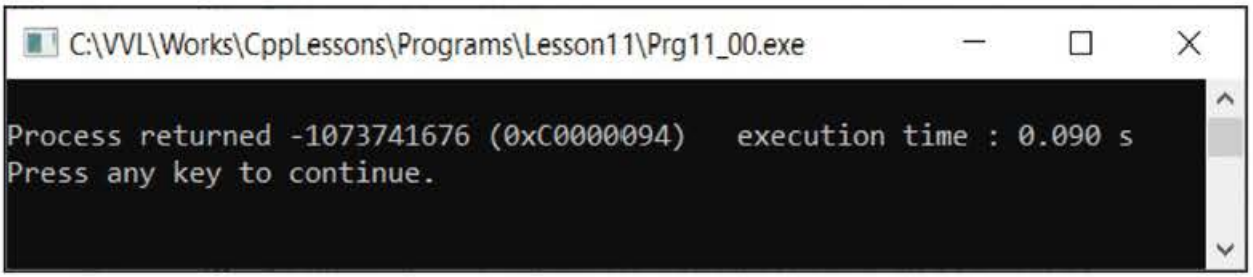


Рис. 15.1. Выполнение программы с делением на 0

Чтобы окончательно убедиться, что программа завершается аварийно, напишем оператор вывода после оператора с делением, например:

```
cout << "Ok\n";
```

Запустим программу заново. Но результат — тот же (см. рис. 15.1). Это означает, что произошло явно аварийное завершение программы, т. к. оператор вывода строкового литерала `"Ok\n"` не выполнился, и программа завершилась раньше, чем дошла до него. Таким образом, на ноль делить нельзя!

На этом пока завершим исследования разных интересных ситуаций с целыми числами и обратимся к вещественным.

Вещественные числа тоже не подарок

Вспомним программы с выводом диапазонов дробных чисел (см. листинг 4.2) и ее вывод в консоль (см. рис. 4.1). Нас интересуют числа: а) самые большие (по абсолютной величине); б) самые маленькие (близкие к нулю).

Допустим, в программе есть переменная `a` типа `float`, которая содержит величину, близкую к предельной, например 10^{37}. Что произойдет, если в программе написать такой оператор:

```
cout << a*1000 << endl;
```

Результат умножения $10^{37}*1000$ больше предельного значения 10^{38}, которое может поместиться в переменную типа `float`. Но при запуске программы выводится интересный результат:

```
inf
```

Никаких предупреждений компилятор не выдает, программа запускается и нормально отрабатывает. Это означает, что результат вполне «легальный».

Понятно, что аналогичная картина возникает и для других плавающих типов:

```
double b {1e307}; cout << b*1000 <<'\n';
long double d{1e4930L}; cout << d*1000 << '\n';
```

В обоих случаях получаем в консоли строку `inf`. Это означает infinity — бесконечность. Существует и отрицательная бесконечность `-inf` — можно вывести это значение:

```
cout << -a*1000 << endl;
```

Практически любые операции нормальных чисел с бесконечностью дают, естественно, `inf` или `-inf`. А вот результат деления числа на бесконечность равен нулю. Причем оказывается, что существует и положительный `+0`, и отрицательный `-0`.

А теперь поделим на ноль:

```
cout << 1.0/0.0 << '\n';
```

Получим, естественно, бесконечность. А вот если ноль поделить на ноль, то получим другой результат — `nan`. Это означает not a number — не число.

Можно говорить о том, что переполнение вещественных чисел значительно отличается от переполнения целых. Переполнение вещественных чисел — это *переполнение порядка*, а мантисса на это не влияет.

0.1 + 0.2 != 0.3

Вещественные числа полны сюрпризов и помимо переполнений. В школе мы привыкли, что всегда выполняется следующее равенство:

```
0.1 + 0.2 = 0.3
```

Однако на компьютере это равенство не выполняется! Напишем простейший код, показанный в листинге 15.1.

Листинг 15.1. 0.1 + 0.2 != 0.3

```
int  main(){
   double d = 0.1+0.2;
   if(d == 0.3)
     cout << "0.1+0.2 = 0.3" << endl;          // равны
   else
     cout << "0.1+0.2 != 0.3" << endl;         // не равны
   return 0;
}
```

Запускаем и с удивлением читаем результат, показанный на рис. 15.2.



Рис. 15.2. 0.1 + 0.2 != 0.3

При проверке этого равенства с помощью `static_assert` компилятор сообщит об ошибке в коде:

```
static_assert(0.1+0.2 == 0.3,"Числа не равны");
error: static assertion failed: Числа не равны
```

Почему получается такой неправильный результат и что с этим делать, мы разберемся позже. А сейчас продолжим изучение «граблей», которые часто «попадаются под ноги» программистам. Напишем функцию, накапливающую сумму вещественных чисел:

```
double f(int n){
    auto result = 0.0;
    for(int i = 0; i < n; ++i){ result += 1.001; }
    return result;
}
```

Вызовем функцию для n = 10, 1000 и 100000 и получим результаты: 10.01, 1001 и 100100. Никаких подводных камней вроде бы не наблюдается. Но это обманчивое впечатление. Давайте установим точность 15 знаков и выведем значения функции для тех же аргументов:

```
cout.precision(15);
cout << f(10)<< ' ' << f(1000) << ' ' << f(100'000) << '\n';
```

Поучаем более интересный результат:

```
10.01 1000.99999999998 100100.000000026
```

Первое значение — правильное, второе и третье станут правильными после округления до целых. Именно это и было сделано программой по умолчанию при отсутствии установки большой точности. Эти результаты показывают, что сравнения `f(1000) == 1001` и `f(100000) == 100100` не выполняются.

Примеры демонстрируют, что сравнивать вещественные числа на равенство нельзя! Однако вещественные числа доставляют гораздо более серьезные проблемы, чем невозможность сравнения. Давайте сделаем из функции `f`, показанной выше, две функции суммирования (листинг 15.2).

Листинг 15.2. Две функции суммирования

```
double f1(int n){
    auto result = 10'000;              // +10'000 до цикла
    for(int i = 0; i < n; ++i){        // накопление суммы
        result += 1.001;
    }
    return result;
}
double f2(int n){
    auto result = 0.0;
    for(int i = 0; i < n; ++i){        // накопление суммы
        result += 1.001;
    }
    return result + 10'000;                    // +10'000 после цикла
}
```

В первой функции число 10 000 добавляется в сумму *до* цикла, а во второй — *после* цикла. В остальном функции идентичны.

Запустим эти функции для тех же значений n = 10, 1000 и 100 000 (код показан в листинге 15.3).

Листинг 15.3. Запуск двух функций суммирования

```
int  main(){
   cout.precision(15);
   auto n = 10;
   cout << "f1(" << n << "): " << f1(n) << "; "
        << "f2(" << n << "): " << f2(n) << endl;
   n = 1000;
   cout << "f1(" << n << "): " << f1(n) << "; "
        << "f2(" << n << "): " << f2(n) << endl;
   n = 100000;
   cout << "f1(" << n << "): " << f1(n) << "; "
        << "f2(" << n << "): " << f2(n) << endl;
   return 0;
}
```

Вывод результатов даже несколько обескураживает (рис. 15.3).

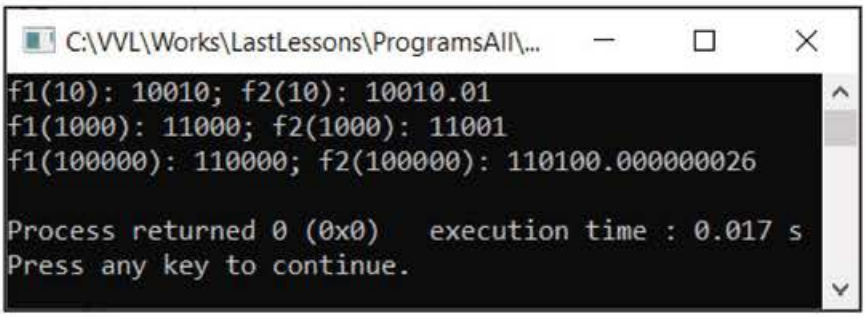


Рис. 15.3. Результаты выполнения двух функций суммирования

Функция `f1` выдает явно неправильные результаты. И если в первой строке ошибка составляет 0.01, то для n = 1000 результаты отличаются уже на целую единицу. Однако ошибка в третьей строке при n = 100 000 уже просто запредельная — 100.

Фактически эти две функции отличаются только порядком слагаемых: в функции `f1` слагаемое `10000` стоит первым, а в функции `f2` — последним. Получается, что результат зависит от перемены мест слагаемых.

«Страшный зверь» по имени UB

Как указано в стандарте языка C++, порядок вычисления подвыражений внутри выражений не определен. Не стоит полагаться на то, что вычисления выражений выполняются слева направо. Это не противоречит приоритету операций. Пусть у нас в программе написано выражение:

```
a = b + c + d;
```

где b, c и d сами являются выражениями. Операции сложения действительно будут выполняться слева направо, т. к. операция сложения левоассоциативна: сначала к b прибавится c, а потом к результату прибавится d. Однако порядок вычислений выражений b, c и d — не определен. Компилятор может сначала вычислить c, затем d и только потом b. Собственно, ну и пусть, скажете вы.

Однако не все так просто. Рассмотрим простой пример:

```
int i = 1, k = 2, j = 3;
i = (i=j-k) + (i=j+k);
cout << i << endl;
```

Ничего «криминального» не наблюдается. Выводится результат, равный 10. А если *поменять слагаемые местами*, тогда результат получается равным 2. Результат *изменился* от перемены мест слагаемых!

Однако дело не в компьютере, а в незнании нами стандарта C++. Подобные ситуации в стандарте называются *undefined behaviour* — неопределенное поведение. А программисты называют короче — UB (читается «убэ»).

Неоднократное присваивание одной и той же переменной в пределах одного выражения является основным признаком наличия UB в таких выражениях.

Неопределенное поведение может проявляться не только в присваиваниях, но и при использовании инкремента или декремента. Например, оператор:

```
i = ++i + 1;
```

является примером из стандарта, демонстрирующим UB.

С другой стороны, оператор:

```
i = 7, i++, i++;
```

в стандарте указан как пример *defined behaviour* — определенное поведение. Переменная i в данном случае будет равна 9. Операция запятая применяется для «придания определенности» арифметическим вычислениям. Напомню, что в C++ есть операции, которые гарантируют строгий порядок вычисления выражений — это логические операции И (||) и ИЛИ (&&).

К неопределенному поведению может привести самая простая конструкция. Например, использование локальной переменной до инициализации — это неопределенное поведение. Можно еще упомянуть следующие ситуации:

- разыменование нулевого указателя (см. *урок 11*);
- доступ за пределы массива («вылет за границу»);
- целочисленное переполнение (см. выше).

В стандарте C++ описано достаточно много ситуаций, в которых может возникать неопределенное поведение. Вообще говоря, программа, в которой есть UB, может давать разные результаты во время выполнения при компиляции разными компиляторами или в разных операционных системах.

Поэтому вычисления необходимо явно проверять в той среде, в которой вы работаете. И обязательно нужно почитать стандарт C++, чтобы выяснить, в каких ситуациях может возникнуть неопределенное поведение.

Обращаю внимание — ситуации, описанные выше для вещественных чисел, не являются неопределенным поведением. Эти проблемы существуют во всех языках программирования и возникают из-за ограниченности вещественных чисел в компьютере.

Как не наступать на «грабли»

Так как целые на ноль делить нельзя, то надо перед делением проверять, не равен ли делитель нулю. Пусть требуется получить частное a / b. Код должен выглядеть примерно так:

```
if(b != 0)
    result = a / b;
else                              // что-то надо делать
```

Важный вопрос, о котором нужно подумать *до* написания кода: что делать, если все-таки делитель равен нулю? Решение, как правило, зависит от задачи. Но в подобных ситуациях обычно требуется прекратить работу программы, сообщив о причинах завершения. Стандарт C++ включает для этих целей *механизм обработки исключений*. Даже начинающий программист обязан знать основы обработки исключений.

Исключения

Я уже упоминал об исключениях, когда писал об определении функции и, в частности, о спецификации исключений `noexcept` в заголовке (см. *урок 7*). Тогда речь шла о том, что функция с такой спецификацией в заголовке не может *генерировать* исключение. Компилятор это учитывает и создает более простой код. Сейчас же речь пойдет о функциях, которые не имеют такой спецификации в заголовке и должны заниматься «расчисткой подводных камней» по ходу выполнения программы.

В языке C++ исключение — это объект. При возникновении исключительной ситуации программа должна генерировать *объект-исключение*. Дальше мы будем говорить просто «исключение», понимая под этим объект-исключение.

Общая схема обработки исключений такова: в одной части программы, где обнаружена аварийная ситуация, исключение *порождается* (генерируется); другая часть программы *контролирует* возникновение исключения, *ловит* и обрабатывает его. Для осуществления этой работы в C++ включены три зарезервированных слова: `throw` (порождать, бросать), `try` (контролировать), `catch` (ловить). Кроме того, обработка исключений поддерживается STL, которая включает большую библиотеку стандартных исключений.

Генерация исключений

Генерируется исключение оператором `throw`. Его синтаксис:

```
throw выражение_генерации_исключения;
```

«Выражение_генерации_исключения» означает переменную или константу некоторого типа. Тип исключения может быть любым: как встроенным, так и реализованным, например:

```
throw 7;                                    // целая константа
throw "Ошибка: деление на ноль!";           // си-строка
throw Message[i];                           // элемент массива
throw 1.1;                                  // вещественная константа
```

Можно определить собственный тип исключения, объявив пустую структуру (см. *урок 10*):

```
struct NegativeArgument {};                 // новый тип данных
NegativeArgument exception;                 // объект нового типа
if ( x > 0 ) double t = x / sqrt( x ); // нормальная работа
else throw exception;                       // генерация исключения
```

Пустые классы неожиданно пригодились! Но объявлять переменную совсем не обязательно, можно обойтись временным объектом, например:

```
throw NegativeArgument();                   // вызов конструктора
```

Здесь создается временный объект с помощью конструктора по умолчанию.

Напишем шаблон функции с проверкой деления на ноль (листинг 15.4).

Листинг 15.4. Генерация исключения при делении на ноль

```
struct Divide_by_zero {};               // тип исключения
template<class T>
T divide (T dividend, T divisor){
    if(divisor == 0)
       throw Divide_by_zero();          // генерация исключения
    else
       return dividend/divisor;         // нормальный результат
}
```

В этом коде надо обратить внимание на определение типа исключения. Имя нового типа сделано «говорящим» и переводится как «деление на ноль». В остальном это типовая схема функции, которая сама не может обработать ошибочные параметры и должна сообщить об этом вызывающему коду.

С переполнениями целых можно разобраться точно так же: надо проверять возможность переполнения перед операцией. Для этого можно написать функции для каждой операции, в которой генерировать исключение при переполнении. Например, функция для сложения показана в листинге 15.5.

Листинг 15.5. Генерация исключения при переполнении целых

```
struct Owerflow {};                         // тип исключения
template<class T>
T add(T a, T b){
```

```
    auto maxi {numeric_limits<T>::max()};
    if(a > maxi - b){                          // правильная проверка
       throw Owerflow();                       // генерация исключения
    }
    return (a + b);                            // нормальный результат
}
```

Обратите внимание на проверку условия в условном операторе. Проверять возможность переполнения суммы нужно только так и никак иначе. Нельзя писать проверку в виде `a + b > maxi`, т. к. сумма `a + b` может вызвать переполнение, и функция в этом случае не выполнит свою работу.

Однако эта функция определяет переполнение только при сложении двух положительных чисел любого целого типа. При сложении отрицательных она не работает — в таких операциях происходит переход через нижнюю границу (сложение двух отрицательных дает положительный результат, см. выше). Поэтому для этого случая надо написать новую функцию.

> ***На заметку***
>
> Переполнение через верхнюю границу диапазона называется overflow. Переполнение через нижнюю границу — это underflow.

Аналогично требуется придумать способ проверки переполнения, когда из меньшего беззнакового целого вычитается большее (как в примере `(0-1)`).

В функции `add` оператор возврата написан как самостоятельный, без ветки `else`, в отличие от функции `divide` с делением на ноль.

Расставим в функции `add` операторы вывода и выясним последовательность работы операторов. А заодно узнаем, что происходит при вызове функции, которая генерирует исключение (листинг 15.6).

Листинг 15.6. Трассировка генерации исключения

```
#include <iostream>
#include <limits>
using namespace std;
struct Owerflow {};                            // тип исключения
template<class T>
T add(T a, T b){
    auto maxi {numeric_limits<T>::max()};
    if(a > maxi - b){
        cout << "overflow!" << endl;
        throw Owerflow();                      // генерация исключения
    }
    cout << "normal!" << endl;
    return (a + b);
}
```

```
int main(){
   cout << add(short(32767), short(1)) << endl; // переполнение
   cout << "normal finish!" << endl;
   return 0;
}
```

При запуске программы получим следующий вывод в консоль (рис. 15.4).



Рис. 15.4. Вывод в консоль при генерации исключения

Вывод показывает следующее:

- функция `add`, вызванная в операторе вывода в главной функции, выявила наличие переполнения (строка «`overflow!`»);
- выполнение оператора `throw` приводит к выводу строки, которая дословно переводится так: завершение вызова после выбрасывания экземпляра «Owerflow»;
- второй оператор вывода в главной программе после вызова функции не выполнился;
- программа завершилась с кодом возврата `3`, что не является нормальным завершением.

Таким образом, выполнение оператора `throw` (генерация исключения) приводит к аварийному завершению программы.

Перехват и обработка исключений

Сгенерировать исключение — это только половина работы. Исключение надо *перехватить* и *обработать*. Проверка возникновения исключения делается с помощью оператора `try`, с которым неразрывно связаны одна или несколько секций-*ловушек* `catch`. Оператор `try` объявляет в любом месте программы *контролируемый блок*, который имеет следующий вид:

```
try { /* контролируемый блок */ }
```

Контролируемый блок помимо контроля выполняет функции обычного блока: все переменные, объявленные внутри него, являются локальными в этом блоке и не видны вне его. После `try`-блока обязательно описывается один или несколько `catch`-блоков, которые называют секциями-ловушками (или *обработчиками* исключений). Формат секции-ловушки следующий:

```
catch ( спецификация_исключения ) { /* блок обработки */ }
```

Стандарт C++ определяет три формата спецификации исключения, но мы будем использовать самый простой и понятный:

```
( тип имя )
```

Тип, как и тип объекта в операторе `throw`, должен быть одним из допустимых типов. Исключение передается в блок обработки, чтобы там его как-то использовать. Например, для вывода информации в сообщении об ошибке. При этом исключение может передаваться в секцию-ловушку теми же способами, что и параметры в функцию, например:

```
catch ( TException e )                    // по значению
catch ( TException& e )                   // по ссылке
catch ( const TException& e )             // по константной ссылке
```

Вполне можно передавать исключение с помощью указателя:

```
catch ( TException *e )                   // по указателю
```

Однако нам пока такой способ передачи исключения не потребуется.

В отличие от параметров функций, для встроенных типов данных никаких преобразований по умолчанию *не выполняется*. Это означает, что если в программе написан заголовок обработчика:

```
catch ( double e )                        // по значению
```

то попадают в эту секцию только те исключения, для которых тип выражения исключения — `double`. Оператор:

```
throw 1;
```

генерирует исключение целого типа, поэтому будет обрабатываться секцией-ловушкой с заголовком, в которой описан целый тип исключения, например:

```
catch ( int e )
```

Работа конструкции `try-catch` напоминает работу оператора `switch`. Секции-ловушки похожи на альтернативы `case`, но работают не со значениями, а с типами. Если в `try`-блоке возникает исключение, начинается сравнение типа сгенерированного исключения и типов в секциях-ловушках. Выполняется тот `catch`-блок, тип параметра которого совпал с типом исключения. Если же такого блока не обнаружено, то программа завершается аварийно (см. рис. 15.4).

После выполнения всех операторов в секции-ловушке управление передается на первый оператор программы после всех секций, например:

```
try{   throw 1;          // исключение типа int
} catch(unsigned int t){
        // выполняемые операторы
} catch(int t){          // перехватывает эта секция
        // выполняемые операторы
}
// сюда переходит управление
```

Если обработка выполняется в функции, то выйти из секции-ловушки можно и с помощью оператора возврата `return`.

Стандартные исключения

Теперь напишем функцию проверки переполнения для умножения целых. Проверка переполнения при умножении делается несколько сложнее, чем для сложения. Прежде чем писать функцию, проанализируем варианты:

- если один из сомножителей равен нулю, то сразу возвращается ноль;
- если один из сомножителей равен 1, то это нормальное произведение и оно возвращается;
- если один из сомножителей равен минимуму диапазона, то при умножении обязательно происходит переполнение (подумайте, почему);
- и, наконец, в общем случае переполнение можно обнаружить, если деление результата на один из множителей не равно второму множителю.

Кроме того, давайте реализуем полную обработку с генерацией и перехватом исключений. Вместо «доморощенных» объектов-исключений используем стандартные исключения. Для этого подключается библиотека `<stdexcept>`. В этой библиотеке реализованы именно те исключения, которые нам сейчас нужны: `overflow_error`, `underflow_error` и т. п.

Стандартные исключения удобны тем, что вместе с исключением в `catch` можно передать диагностическое сообщение. Поэтому даже для одного типа исключений можно выдавать разные диагностические сообщения из разных мест программы. Полный код показан в листинге 15.7

Листинг 15.7. Обработка исключений переполнения при умножении

```
#include <iostream>
#include <limits>
#include <stdexcept>                              // библиотека исключений
using namespace std;
int multiply(int a, int b){
   if(a == 0 || b == 0) return 0;
   auto result = a * b;
   if(a == 1 || b == 1) return result;
   auto mini {numeric_limits<int>::min()};      // наименьшее
   if(a == mini || b == mini){
      throw overflow_error("Переполнение с минимумом!"s);
   } else if(result/b != a){
      throw overflow_error("Переполнение!"s);
   }
   return result;
}
int main(){
    auto mini {numeric_limits<int>::min()}    ;
    try {  cout << multiply(mini, 2) << endl;  }
```

```
    catch(const overflow_error &t){              // обработчик
        cout << t.what() << endl;                // вывод сообщения
    }
    try {  cout << multiply(abs(mini+1), 2) << endl;  }
    catch(const overflow_error &t){              // обработчик
        cout << t.what() << endl;                // вывод сообщения
    }
    return 0;
}
```

Код функции `multiply` реализован в точном соответствии с приведенными выше соображениями. В ней дважды генерируется стандартное исключение `overflow_error` с разными сообщениями об ошибке. В главной функции написано два контролируемых блока отдельно для каждого из исключений. Несмотря на то, что обработчики исключений одинаковые, сообщения в консоль выводятся разные (рис. 15.5).

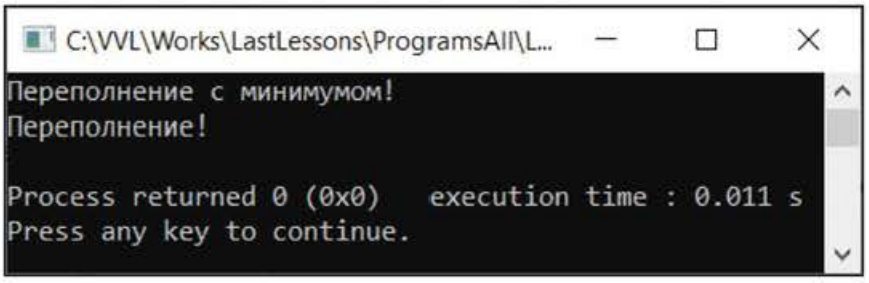


Рис. 15.5. Обработка исключения `overflow_error`

Конечно, хотелось бы еще иметь информацию о строке программы, где это произошло, но и так сообщение на русском языке лучше, чем на английском.

Обработка «плавающих граблей»

Проще всего решить проблему сравнений вещественных чисел: сравнивать нужно не сами числа, а абсолютное значение разности с маленьким числом, например 0.001 или 1e-6. Выглядит это так:

```
if(abs(a-b) < 0.0001) …
```

В таких случаях говорят: с точностью до некоторого эпсилон. Эпсилон — это буква греческого алфавита. В математике слово «эпсилон» обычно означает некоторое маленькое число. Какое конкретно — зависит от задачи.

Выше мы видели, что вычисления с вещественными числами никогда не бывают точными и всегда выполняются с некоторой погрешностью. Вот эта погрешность и должна быть меньше заданного эпсилон, иначе вычисления не могут считаться достоверными.

Поэтому пример `0.1 + 0.2 != 0.3` (см. листинг 15.1) нужно переписать так:

```
if(abs(d - 0.3) < 1e6)
   cout << "0.1+0.2 = 0.3 с точностью до 1e6" << endl;
```

В вычислительных задачах подобные проверки должны выполняться регулярно.

Функции *isnan*, *isinf*, *isfinite*

Так как вычисления с плавающими числами не генерируют исключения, то приходится проверять результаты вычислений с помощью специальных функций `isnan`, `isinf`, `isfinite`. Они находятся в библиотеке `<cmath>` и ее надо подключить, чтобы их можно было использовать. Посмотрите, как работать с этими функциями:

```
d = 0.3;
if(isfinite(d))
   cout << "Конечное число " << d << endl;
d = d/0.0;
if(isinf(d))
   cout << "Бесконечность  " << d << endl;
d = 0.0/0;
if(isnan(d))
   cout << "Не число       " << d << endl;
```

В библиотеке присутствует еще несколько функций, предназначенных для проверки результатов вычисления с плавающей точкой.

Задачи и упражнения

1. Напишите программу, в которой демонстрируются описанные выше проблемы с целыми числами.
2. Напишите программу, в которой используются функции *isnan*, *isinf*, *isfinite*.
3. Напишите программу, которая демонстрирует неопределенное поведение.
4. Реализуйте функцию проверки переполнения при сложении отрицательных целых чисел.
5. Реализуйте функцию проверки переполнения при вычитании беззнаковых целых чисел.
6. Реализуйте функцию проверки переполнения для операции инкремента.

- ЧАСТЬ III -

Вглубь C++

УРОК 16. КАК ВСЕ УСТРОЕНО
УРОК 17. ОПЯТЬ ФУНКЦИИ
УРОК 18. СНОВА УКАЗАТЕЛИ
УРОК 19. УЧИМСЯ ПИСАТЬ И ЧИТАТЬ
УРОК 20. БОЛЬШИЕ ПРОГРАММЫ

- Урок 16 -

Как все устроено

Предыдущий *урок 15* был посвящен разбору и анализу различных проблем и неприятностей, которые регулярно случаются в коде. Поэтому необходимо выяснить, почему так происходит. И в этом нам помогут двоичные числа.

Числа, числа и числа

До сих пор мы имели дело только с десятичными целыми числами. Но ранее (см. *урок 4*) было сказано, что помимо десятичных целых констант в C++ имеются шестнадцатеричные, восьмеричные и двоичные константы (литералы). Шестнадцатеричные и восьмеричные константы унаследованы еще от C, а двоичные были добавлены в C++14.

Все такие литералы должны быть написаны с ведущим нулем, но немного поразному. Десятичные литералы писать с ведущими нулями нельзя. Нечаянно написав ведущий ноль в константе, можно «наступить на грабли», которые в изобилии «рассыпаны под ногами» у программистов в языке C++.

Давайте просто выведем в консоль первые 17 чисел в трех видах: десятичном, восьмеричном и шестнадцатеричном (код в листинге 16.1).

Листинг 16.1. Первые 17 чисел в трех форматах

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(){
   cout.setf(ios::showbase);                          // индикатор
   for(int i{0}; i < 17; ++i){
    cout << setw(4) << dec << i << '\t'          // десятичная
         << setw(4) << oct << i << '\t'          // восьмеричная
         << setw(4) << hex << i << '\n';         // шестнадцатеричная
    }
   return 0;
}
```

Результат показан на рис. 16.1. Самый левый столбец — это десятичные числа, в среднем столбце — восьмеричные, в правом — шестнадцатеричные. Восьмеричные числа выводятся с начальным нулем, а шестнадцатеричные — с префиксом `0x`.

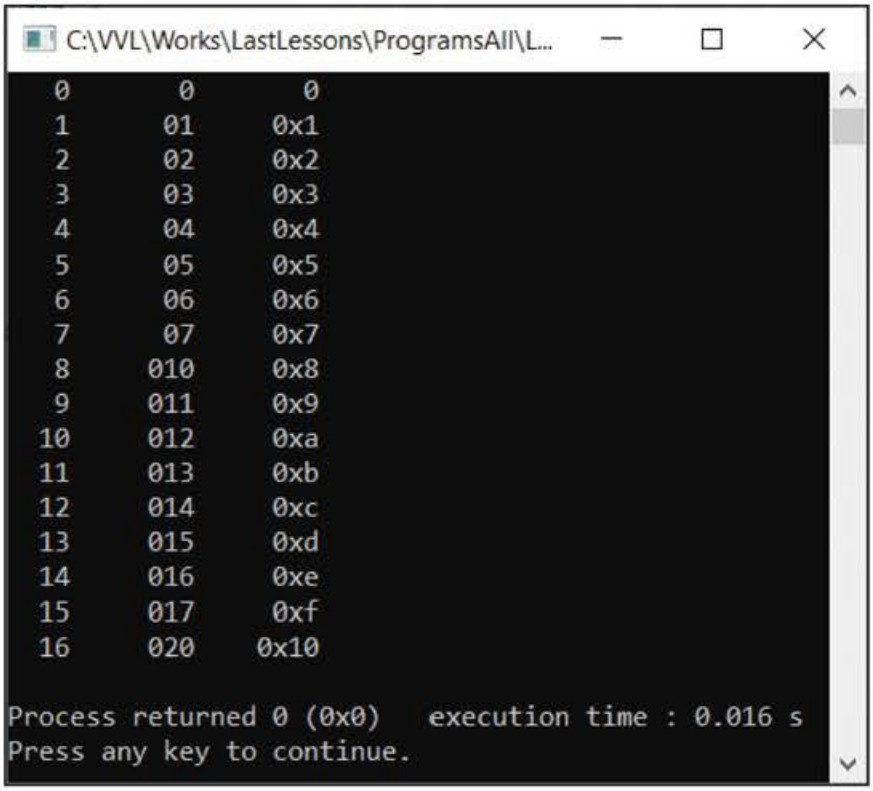


Рис. 16.1. Первые 17 чисел в трех форматах

Формат вывода определяется манипуляторами `dec` (decimal, десятичный), `oct` (octal, восьмеричный), `hex` (hexadecimal, шестнадцатеричный). Если не писать манипуляторы, то по умолчанию установлен вывод в десятичном виде. Но можно изменить умолчание, написав в операторе вывода манипулятор `setbase(b)`, где *b* может принимать значения 8, 10 и 16.

Перед циклом для выходного потока установлен флаг `showbase`, с помощью которого показывается, в какой *системе счисления* выведено число. На рис. 16.1 мы видим, что во втором столбце числа написаны с ведущим нулем (это признак восьмеричной системы), а в третьем — с префиксом `0x` (признак шестнадцатеричной системы).

В одной строке написано одно и то же число, представленное в разных системах счисления. В первой строке выведено три нуля, т. к. ноль во всех системах одинаков. От `0` до `7` все числа одинаковы, только префиксы разные. Десятка `010` во втором столбце появилась после восьми чисел от `0` до `7`. После следующих восьми чисел от `010` до `017` в последней строке видим `020`. Десятка `0x10` в третьем столбце появилась только после 16 чисел, причем после `9` следующие числа написаны буквами от `a` до `f`.

Мы видим, что десятки десятичных чисел появляются через каждые 10 чисел, восьмеричных — через 8 чисел и шестнадцатеричных — через каждые 16 чисел. Чтобы с этим разобраться, нужно выяснить, как устроены целые числа и как они хранятся в памяти компьютера. Начнем с систем счисления.

Числа как многочлены

Мы записываем десятичные целые числа в позиционной системе счисления. *Система счисления* — это способ записи чисел с помощью набора цифр. Система счисления называется *позиционной*, если значение цифры в числе определяется ее позицией. Мы называем числа десятичными, т. к. основание системы счисления — число 10. Количество цифр в системе тоже задается этим числом, причем все цифры от 0 до 9 меньше основания системы.

Десятичные числа мы пишем в короткой свернутой форме. Но число можно написать в развернутой форме:

$12345 = 10000 + 2000 + 300 + 40 + 5 = 1*10^4 + 2*10^3 + 3*10^2 + 4*10^1 + 5*10^0$

Эта запись представляет собой многочлен, с которыми мы уже имели дело (см. *урок 4*) в задаче «Математика должна быть экономной» (см. листинг 4.5). В математике многочлен называют *полиномом*. Полиномом $P(x)$ называется математическое выражение следующего вида:

$$P(x) = a_n x^n + a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \ldots + a_2x^2 + a_1x^1 + a_0x^0 \qquad (16.1)$$

Степень $x^1 = x$ и $x^0 = 1$ обычно не пишут, но в данном случае я написал, чтобы показать явное сходство с развернутой формой написания числа.

Если в этом выражении вместо x поставить основание системы счисления, а коэффициенты a_i считать цифрами, то получается развернутое представление числа в виде полинома. Обычно основание системы счисления обозначается как b (base, база). Тогда можно представить любое целое число в таком виде:

$$P_b = a_n b^n + a_{n-1}b^{n-1} + a_{n-2}b^{n-2} + \ldots + a_2b^2 + a_1b^1 + a_0b^0 \qquad (16.2)$$

Подставляя вместо b число 10, получаем десятичные числа с цифрами a_i от 0 до 9. Но можно вместо b подставить 8, тогда цифры a_i должны быть от 0 до 7, и число будет восьмеричным. Если вместо b подставить 2, и цифры a_i будут 0 или 1, то число будет двоичным.

Более интересная система — шестнадцатеричная. Понятно, что вместо b надо подставить 16, но где мы возьмем шестнадцать цифр — у нас их только 10. Поэтому *договорились*, что недостающие шесть цифр будут обозначаться первыми шестью буквами английского алфавита (что мы и видим на рис. 16.1).

Таким образом, набор шестнадцатеричных цифр представляется так:

```
0 1 2 3 4 5 6 7 8 9 a b c d e f.
```

Вместо строчных букв можно использовать прописные. Если в операторе вывода написать манипулятор `uppercase`, то вместо строчных букв будут выведены прописные.

Обычно система счисления указывается в числе в виде нижнего индекса, аналогично индексам в a_i в формулах 16.1 и 16.2. Например, число 10 в двоичной, восьмеричной и шестнадцатеричной системах записывают так:

10_2 10_8 10_{16}.

Это разные десятки, и ни одна из них не равна десятичному числу 10_{10}.

Перевод из десятичной в другую систему

Стандартные средства предоставляют возможности вывода (и ввода тоже) в трех системах счисления. А вот вывод чисел в двоичной системе отсутствует. Поэтому давайте напишем функцию, которая нам из десятичного целого беззнакового числа сформирует его двоичный вид в виде строки. Параметром является число, результат — строка, состоящая из двоичных цифр 0 и 1.

Переводить будем с помощью операции деления. Давайте рассмотрим для примера перевод числа 10_{10}. Если мы десятичное число 10 поделим нацело на 2 (переводим в двоичную систему), то получим частное 5 и остаток 0. Этот ноль является цифрой a_0 (формула 16.2) в двоичном представлении числа. Теперь частное 5 снова поделим нацело на 2, получим частное 2 и остаток 1. Эта единица — цифра a_1. Снова делим частное 2 на 2, получаем остаток 0 и частное 1. Остаток ноль — это цифра a_2, а частное — цифра a_3. Последующие деления дают частное 0, поэтому на этом деление можно прекращать.

Это можно записать на псевдокоде так:

```
число = 10, строка = ""                  // пустая
пока число > 0
  цифра = число % 2                      // двоичная цифра
  прицепить цифру к строке
  число = число / 2                      // «отрезаем» последнюю цифру
конец цикла
```

В результате перевода получаем $10_{10} = 1010_2$. Число 1010_2 — это *двоичный код* десятичного числа 10_{10}. Один разряд двоичного кода (одна двоичная цифра) — это один *бит*.

Так как мы собираемся переводить беззнаковое целое любого типа, то писать будем функцию-шаблон. Заголовок шаблона будет такой:

```
template<class T> string to_binary(T number)
```

Надо решить один вопрос: каков размер возвращаемой строки? Напомню, что размер чисел вычисляется в байтах, а нам нужно количество битов. Вспомним синонимы целых типов (см. *урок 4*) с количеством битов:

- `uint8_t` — 1 байт, 8 бит;
- `uint16_t` — 2 байта, 16 бит;
- `uint32_t` — 4 байта, 32 бита;
- `uint64_t` — 8 байт, 64 бита.

В C++ в библиотеке `<limits>` в классе пределов `numeric_limits` для типа `unsigned char` задана константа, равная количеству битов в этом типе: `numeric_limits<unsigned char>::digits`. Длину строки вычислим так:

```
auto length = sizeof(T)*numeric_limits<unsigned char>::digits;
```

Полный код функции приведен в листинге 16.2.

Листинг 16.2. Функция перевода в двоичную систему (версия 1)

```
template<class T>
string to_binary(T number){
    string binary{};                              // строка-результат
    auto length = sizeof(T)*numeric_limits<unsigned char>::digits;
    decltype(length) i = 0;                       // тот же тип
    while(i < length){
        auto digit = number % 2;                  // цифра
        binary = char(digit+'0') + binary; // добавляем спереди
        number = number / 2;                      // частное
        ++i;
    }
    return binary;
}
```

Теперь модифицируем цикл вывода чисел (см. листинг 16.1):

```
string bin;                                       // двоичная строка
for(int i{0}; i < 16; ++i){
   bin = to_binary(uint8_t(i));                   // достаточно 1 байта
   cout << setw(3) << dec << i << '\t'            // десятичное
        << setw(3) << oct << i << '\t'            // восьмеричное
        << setw(3) << hex << i << '\t'            // шестнадцатеричное
        << bin.substr(4) << '\n';                 // вывод 4 бит
```

Последняя строка оператора вывода содержит вызов метода `substr` для сформированной строки `bin`. Целое `i` размером 4 байта (32 бита) с помощью сужающего приведения `uint8_t(i)` в аргументе функции `to_binary` урезано до 1 байта (8 бит). Поэтому строка `bin` имеет размер 8 символов, из которых первые четыре равны нулю (для всех `i` нашего цикла). Метод `substr` «вырезает» из строки подстроку (substring), начиная с четвертого символа до конца строки.

Замечу, что строку `bin` можно было не объявлять, а написать в операторе вывода `to_binary(uint8_t(i)).substr(4)`. Результат будет тот же.

При запуске программы получим красивый вывод, показанный на рис. 16.2.

Понятно, что точно так же делается перевод и в восьмеричную систему (делим на 8 и получаем восьмеричные цифры — остатки от деления), и в шестнадцатеричную (делим на 16). Напишите эти функции сами.

И обратите внимание, что одна шестнадцатеричная цифра представляется четырьмя двоичными цифрами. Это неудивительно, т. к. $16^1 = 2^4$. Поэтому можно сначала перевести в шестнадцатеричную систему, а потом каждую шестнадцатеричную цифру «расписать» в четыре двоичных.

То же самое справедливо и для преобразований `8->2` и `2->8`.

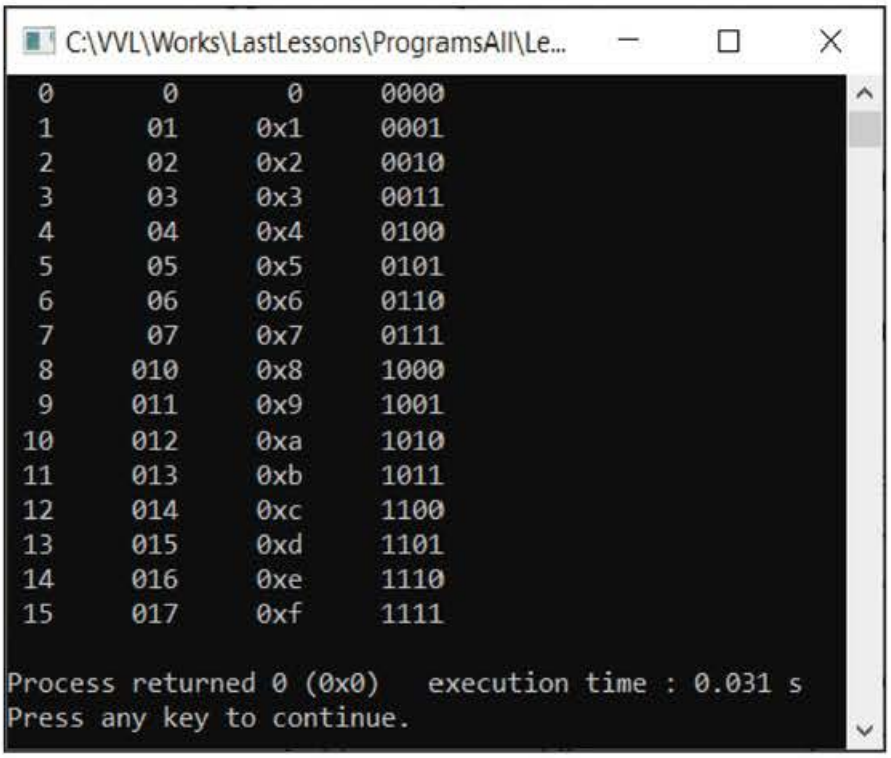


Рис. 16.2. Вывод 16 чисел в четырех системах счисления

Перевод в десятичную систему

Если перевод `10->2` выполняется делением, то обратный перевод `2->10` делается умножением. Просто запишем двоичный код в развернутой форме:

$1010_2 = 1*2^3 + 0*2^2 + 1*2^1 + 0*2^0 = 8 + 0 + 2 + 0 = 10_{10}$

Это вычисление является вычислением полинома (формула 16.1) для $x = 2$ с коэффициентами, которые являются двоичными цифрами. Для вычисления полинома применяется так называемая *схема Горнера*. Представим полином в следующем виде:

$$P(b) = (\ldots((a_n x + a_{n-1})x + \ldots + a_2)x + a_1)x + a_0 \qquad (16.3)$$

Тогда вычисления можно выполнять в такой последовательности:

$b_n = a_n$

$b_{n-1} = a_{n-1} + b_n x$

$b_{n-2} = a_{n-2} + b_{n-1} x$

...

$b_i = a_i + b_{i+1} x$

...

$b_0 = a_0 + b_1 x$

Значение b_0 — это и есть результат: значение $P(x)$. Реализуем такой алгоритм в виде функции, параметром которой будет двоичное число в виде строки. Очевидное название функции `to_decimal`, и прототип такой:

```
uint64_t to_decimal(string_view binary);
```

Код программы представлен в листинге 16.3.

Листинг 16.3. Перевод 2->10 по схеме Горнера

```
uint64_t to_decimal(string_view binary){
    uint64_t result{};
    auto i = begin(binary);                    // итератор по числу
    do{ result = result * 2;                   // x = 2
        result = result + (*i - '0');          // очередной бит
        ++i;                                   // передвинулись
    } while (i != end(binary));
    return result;
}
```

Теперь функции `to_binary` и `to_decimal` образовали пару, и это позволяет тестировать их совместно, например:

```
string binary = to_binary(uint8_t(255));              // 10 -> 2
uint64_t result = to_decimal(binary);                 // 2 -> 10
cout << binary << " "<< result << endl;               // 11111111 255
```

Вставив этот фрагмент в главную функцию, получим в консоли вывод:

```
11111111 255
```

Первые 8 единиц — это перевод 255_{10} -> 11111111_2. Второе число `255` — это обратный перевод 11111111_2 -> 255_{10}. Таким образом, убеждаемся, что все работает правильно.

Точно так же делается перевод в восьмеричную систему (надо умножать на 8) и шестнадцатеричную систему (надо умножать на 16).

Стандартные средства

Беззнаковые целые можно переводить стандартными функциями, которые появились в стандарте C++17:

- `to_chars` — перевод целого числа в символы в любой системе счисления;
- `from_chars` — перевод из символов в число в любой системе счисления.

Для работы с этими функциями надо подключить библиотеку `<charconv>`.

Работать с ними можно так:

```
char digits[65]{};
to_chars(begin(digits), end(digits), 128, 2);
```

Первые два параметра — указатели на массив символов, третий параметр — переводимое число, четвертый — система счисления. В массив `digits` запишется последовательность символов '1' '0' '0' '0' '0' '0' '0' '0' — двоичная запись числа `128`.

Вместо символьного массива можно задавать и array-массив, а переводимое число может быть задано и в восьмеричном, и в шестнадцатеричном виде.

Обратный перевод надо делать так:

```
auto [last, err] = to_chars(begin(digits), end(digits), 128, 2);
int number{};
from_chars(begin(digits), last, number, 2);     // обратный перевод
```

В переменной `number` получим исходное число `128`.

Здесь `last` — указатель на следующий элемент массива `digits` за последним символом результата.

Если мы хотим работать с шестнадцатеричными числами, то достаточно заменить четвертый аргумент в вызове стандартных функций с 2 на 16.

Перевод дробных чисел

Теперь займемся переводом дробных. Представим дробное число в виде многочлена. Многочлен (см. формулу 16.2) заканчивался нулевой степенью, и цифра имела такой же индекс — так представляются целые числа. После младшей цифры целого начинается дробная часть, и степени после запятой являются отрицательными, поэтому логично продолжить нумерацию цифр тоже отрицательную. Пусть число содержит k цифр после запятой:

$$P_b = a_n b^n + \ldots + a_0 b^0 + \boldsymbol{a_{-1} b^{-1} + a_{-2} b^{-2} \ldots + a_{-k} b^{-k}} \quad (16.3)$$

Не обращая внимания на целую часть, рассмотрим только дробную:

$$p_b = a_{-1} b^{-1} + a_{-2} b^{-2} \ldots + a_{-k} b^{-k} \quad (16.4)$$

Если перевод целых чисел `10->2` делался делением, то для дробных тот же перевод делается умножением. Если мы умножим p_b на b, то получим:

$$a_{-1} + a_{-2} b^{-1} \ldots + a_{-k} b^{-k+1}$$

Цифра a_{-1} — это первая цифра числа в новой системе счисления (целая часть умноженного числа). Умножая оставшуюся дробную часть снова на b, получим вторую цифру a_{-2} в целой части. Выполняем умножение до тех пор, пока оставшаяся дробная часть не станет равна нулю. Например, переведем 0.5_{10} в двоичную систему:

```
0.5₁₀           // исходное десятичное число
× 2
1.0             // первая цифра 1, дробная = 0 -> конец
```

Таким образом, $0.5_{10} = 0.1_2$. Результат выглядит немного неожиданно, но он совершенно правильный:

$$0.1_2 = 1 \times 2^{-1} = 0.5_{10}$$

Это у нас получилось удачно, перевод быстро завершился. Открою секрет: так удачно переводятся только степени 2 и их суммы. Но в общем случае почти всегда получается бесконечная двоичная дробь. Чтобы не проверять это вручную, напишем функцию перевода дробных десятичных чисел в двоичную систему.

Сразу возникает вопрос: в каком виде представлять исходное дробное число? Писать в программе десятичное число типа `float` или `double` с дробной частью беспо-

лезно — перевод выполнит компилятор. Поэтому исходное дробное число будем задавать в виде строки десятичных цифр. Будем писать число в виде «0.xx…x».

Результат — тоже строка, но цифры уже двоичные. Вместо умножения на 2 мы используем написанную ранее функцию сложения длинных чисел (см. *урок 13*, листинг 13.12): надо просто число сложить с самим собой. Так как мы не знаем, какого размера будут получаться двоичные числа, давайте ограничим размер результата 32 двоичными цифрами. Код показан в листинге 16.4.

Листинг 16.4. Перевод дробных чисел 10->2

```
string fraction_to_binary(string number){
  string result{"0."};                    // строка - результат
  const size_t n = 32;                    // количество битов
  uint64_t Zero{};                        // для проверки на ноль
  size_t i = 0;                           // счетчик битов
  string work = number.substr(2);         // получить целое без "0."
  auto length = size(work);               // размер числа для умножения
  do{ work = add(work, work);             // типа умножили на 2
      if(size(work) > length){            // если был перенос
         result = result + work[0];       // старшая цифра результата
         work = work.substr(1);           // изъяли старшую цифру
      } else result = result + '0';       // если не было переноса
                                         // переводим - не ноль ли остался в числе
      from_chars(work.data(),work.data()+size(work), Zero, 10);
      ++i;                                // текущее количество битов
      if (i == n) break;                  // достигли размера?
    } while(Zero != 0);
  return result;
}
```

Так как мы работаем со строками, то в функции выполняются манипуляции с результатом «умножения»: старшая цифра присоединяется к выходной строке и отрезается от «произведения». Затем оставшаяся дробная часть переводится уже в целое число и сравнивается с нулем. Отдельно проверяем количество битов.

В главной программе напишем операторы:

```
string binary;
binary = fraction_to_binary("0.500");
cout << 0.5 << "\t=" << binary << endl;
binary = fraction_to_binary("0.125");
cout << 0.125 << "\t=" << binary << endl;
binary = fraction_to_binary("0.625");
cout << 0.625 << "\t=" << binary << endl;
```

Получим в консоли вывод:

```
  0.5   =0.1          // степень 2^-1
0.125   =0.001        // степень 2^-3
0.625   =0.101        // степень 2^-3
```

А теперь напишем перевод и вывод чисел, не являющихся степенями двойки:

```
binary = fraction_to_binary("0.025");
cout << 0.025 << "\t=" << binary << endl;
binary = fraction_to_binary("0.275");
cout << 0.275 << "\t=" << binary << endl;
binary = fraction_to_binary("0.100");
cout << 0.1 << "\t=" << binary << endl;
binary = fraction_to_binary("0.345");
cout << 0.345<< "\t=" << binary << endl;
```

И получаем в консоли 32-битные последовательности:

```
0.025   =0.00000110011001100110011001100110    // периодическая дробь
0.275   =0.01000110011001100110011001100110    // периодическая дробь
0.1     =0.00011001100110011001100110011001    // периодическая дробь
0.345   =0.01011000010100011110101110000101    // периодическая дробь
```

Все числа в двоичной системе являются периодическими дробями.

> ***На заметку***
>
> Число 0.345, на первый взгляд, не является периодической двоичной дробью. Однако увеличив в функции количество битов до 64 (сделайте это самостоятельно), мы увидим, что и эта дробь имеет период (0000101000111101011 1).

Это означает, что точное вещественное число не может быть представлено типом `double`, и его приходится «усекать».

И это делается с любыми дробными числами, не кратными степени двойки. Таким образом, возникает *погрешность*, которая может накапливаться при вычислениях. Именно поэтому на компьютере $0.1_{10} + 0.2_{10} != 0.3_{10}$

Битовое устройство беззнаковых целых

Я уже писал (см. *урок 4*), что с некоторыми проблемами (см. *урок 15*) никак не разобраться без знания битовой структуры данных. Для беззнаковых целых функция `to_binary` позволяет получить битовое представление числа любого размера. Выше было показано (см. *урок 15*), что «фокусы» с целыми происходят независимо от их размеров. Поэтому будем работать с короткими целыми типа `uint8_t`.

Чтобы понять переполнение, нужно научиться складывать двоичные числа. Для двоичных чисел имеем следующие правила сложения:

```
0 + 0 =  0
0 + 1 =  1
1 + 0 =  1
1 + 1 = 10
```

При сложении двух единиц произошел перенос в следующий разряд (двоичные десятки). Рассмотрим сумму `255 + 1 = 256` в двоичной системе для типа `uint8_t`.

При переводе числа выглядят так:

255_{10} = 11111111_2

1_{10} = 00000001_2

256_{10} = 100000000_2

Сложение выполняется так:

```
cf    11111110                  // перенос
     |11111111|                 // 255
   + |00000001|                 //   1
    1|00000000|                 // 256 - не помещается в 8 бит
```

Складывая самые младшие цифры `1 + 1`, получаем `10`. При этом в результат идет ноль, а старшая единичка попадает в перенос. И все последующие цифры складываются с переносом, и каждый раз получается результат `10`, и возникает новый перенос.

И наконец, самая старшая цифра `1` первого числа складывается со старшей цифрой `0` второго числа и с переносом от сложения предыдущих цифр. И опять получается `10`. Цифра `0` попадает в старший разряд результата, а для переноса места нет, т. к. это будет девятый разряд, а в числах типа `uint8_t` битов всего восемь.

Все восемь разрядов равны нулю — именно этот результат получался у нас всегда при добавлении единицы к максимуму диапазона беззнаковых целых. Для 16-битных целых сложение выглядит так:

```
cf    1111111111111110          // перенос
     |1111111111111111|         // 65535
   + |0000000000000001|         //     1
    1|0000000000000000|         // 65536 - не помещается в 16 бит
```

И здесь получается тот же «фокус»: результат не помещается в 16 бит. Про такие ситуации говорят, что «результат не поместился в разрядную сетку». Для изучения знаковых целых нам потребуются битовые операции.

Битовые операции

Помимо арифметических операций с целыми числами (любых типов) в C++ допускается выполнять *битовые* операции. Напомню, что одна двоичная цифра называется *битом*, и целое число — это последовательность битов. С битовыми последовательностями разрешается выполнять битовые операции. Этих операций четыре:

- операция `&` — битовое И;
- операция `|` — битовое ИЛИ;
- операция `~` — битовое НЕ;
- операция `^` — битовое исключающее ИЛИ.

Мы уже рассматривали *логические* операции (см. *урок 5*) с целыми числами. Первые три битовые операции работают аналогично, только применяются они к от-

дельным битам, а не к числам. Если целое число имеет размер 8 бит (как `int8_t`), то операция применяется к каждому из 8 бит в отдельности. Результат отличается от результата аналогичной логической операции.

Операция битового `и` работает по следующим правилам:

```
0 & 0 = 0
0 & 1 = 0
1 & 0 = 0
1 & 1 = 1
```

Операция битового `или` работает так:

```
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1
```

Ну, и битовое `не` делает с битами (с каждым в отдельности) то же самое, что и логическое `не` с целыми:

```
~0 = 1
~1 = 0
```

> ***На заметку***
>
> Все битовые операции имеют форму с присваиванием.

Приведем пример, показывающий работу этих операций.

Пусть целое типа `uint8_t` имеет значение $85_{10} = 01010101_2$. Тогда битовое `не` даст результат: `~01010101 = 10101010`.

Такая операция часто называется инвертированием: нулевые биты становятся единицами, и наоборот. Отличие логического отрицания от битового в том, что выражение `!85` равно нулю. Операция логического отрицания выдает в результате целое число ноль или один. А операция битового отрицания получает ноль или один в каждом бите целого числа.

Операция битовое `и` с этими операндами даст такой результат:

```
  01010101
& 10101010
= 00000000
```

Операция битовое `или` с этими же операндами даст такой результат:

```
  01010101
| 10101010
= 11111111
```

Эти примеры можно записать прямо в коде, используя двоичные литералы.

Как писать двоичные литералы

Мы уже знаем, что шестнадцатеричные литералы пишутся с префиксом `0x`, а восьмеричные — с ведущим нулем (см. рис. 16.1).

Но C++17 позволяет писать и двоичные литералы с префиксом 0b, например:

```
int8_t x{0b01010101};                    // 0b01010101 = 85₁₀
```

Выполнение вывода:

```
cout << " x:    " << to_binary(x) << endl;
cout << "~x:    " << to_binary(uint8_t(~x)) << endl;
cout << " x&~x: " << to_binary(uint8_t(x & uint8_t(~x))) << endl;
cout << " x|~x: " << to_binary(uint8_t(x | uint8_t(~x))) << endl;
```

покажет в консоли следующие результаты:

```
 x:    01010101
~x:    10101010
 x&~x: 00000000
 x|~x: 11111111
```

Результаты именно те, которые показаны в примерах выше.

Исключающее *ИЛИ*

Более интересной является операция исключающего ИЛИ, которая кратко называется xor. Эта операция работает по следующим правилам:

```
0 ^ 0 = 0
0 ^ 1 = 1
1 ^ 0 = 1
1 ^ 1 = 0
```

Эту операцию иногда называют «сложение по модулю 2», т. к. выполняется сложение одного бита, а перенос в старший разряд (см. выше) теряется:

```
1 ^ 1 = (1+1)%2 = 2 % 2 = 0
```

Вообще-то эта операция реализуется с помощью трех первых так:

```
p ^ q = (p | q) & ~(p & q)
```

Но разработчики языка C включили ее в состав битовых операций, т. к. обычно она входит в состав операций компьютера. Логической операции такой нет, но можно ее реализовать по аналогичной формуле, которая отличается только знаками операций. Напишем функцию XOR (листинг 16.5).

Листинг 16.5. Логическая операция исключающее *ИЛИ* (*xor*)

```
bool XOR(bool p, bool q){
    return (p || q) && !(p && q);              // исключающее ИЛИ
}
```

Вывод в консоль:

```
cout << XOR(true, true) << endl;               // 0 - false
cout << XOR(true, false) << endl;              // 1 - true
cout << XOR(false, true) << endl;              // 1 - true
cout << XOR(false, false) << endl;             // 0 - false
```

Результаты соответствуют правилам работы операции исключающего ИЛИ.

Рассмотрим пример работы этой операции:

```
  01111111                          // x
^ 10111001                          // y
  11000110                          // r - результат
```

А теперь выполним операцию `r ^ y`:

```
  11000110                          // r - результат
^ 10111001                          // y
  01111111                          // получаем x!
```

Получили исходное значение `x`! Таким образом, выполняются соотношения:

```
r = x ^ y
x = r ^ y
y = r ^ x
```

Это позволяет написать простой шифратор текста. Если x — это символ, а y — ключ шифрования, то r — результат шифрования символа x ключом y:

```
r = x ^ y                           // шифрование
```

Тем же ключом y текст расшифровывается:

```
x = r ^ y                           // расшифровка
```

Но не забывайте, что это работает с однобайтными кодировками.

Сдвиги

Нам осталось разобраться еще с парой очень важных операций, которые называются сдвигами. Сдвигов, естественно, два:

- операция >> сдвига вправо;
- операция << сдвига влево.

Операции сдвига — бинарные: слева от операции задается операнд целого типа (переменная), справа — целочисленное выражение (целый литерал). Левый операнд сдвигается на количество битов, указанное в правом операнде.

Пусть x — целая беззнаковая переменная типа `uint8_t`, равная `0b11111111`. Сдвиг на 1 бит вправо и влево показан рис. 16.3.

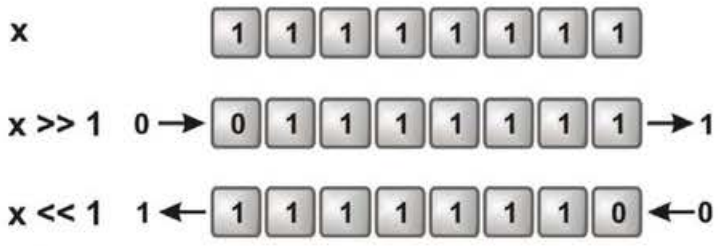


Рис. 16.3. Сдвиги беззнаковых целых на 1 разряд вправо и влево

Если значение левого операнда не более половины диапазона, то сдвиг влево на 1 разряд является умножением на 2. Например, пусть `x` = 3_{10} = 00000011_2. Тогда `(x << 1) = (00000011 << 1) = 00000110`.

Значение 00000110_2 равно 6_{10}.

Сдвиг вправо — это целочисленное деление на 2 (проверьте самостоятельно).

Для знаковых целых сдвиг влево выглядит точно так же, а сдвиг вправо имеет некоторые особенности, с которыми мы познакомимся дальше.

Битовое устройство целых со знаком

Теперь с помощью битовых операций перепишем функцию `to_binary`, чтобы она смогла работать и со знаковыми, и с беззнаковыми целыми. Вместо операции получения остатка от деления на 2 (`number % 2`) применим битовую операцию `И` с младшим битом числа (`number & 0b1`). А вместо операции деления на 2 (`number / 2`) используем операцию сдвига вправо на один бит (`number >> 0b1`). Сделаем шаблон и назовем функцию `to_bits` (листинг 16.6).

Листинг 16.6. Функция перевода в двоичную систему (версия 2)

```
template<class T>
string to_bits(T number){
    string binary{};
    auto length = sizeof(T)*numeric_limits<unsigned char>::digits;
    decltype(length) i = 0;
    while(i < length){
        auto digit = number & 0b1;              // младший бит
        binary = char(digit+'0') + binary;      // добавляем спереди
        number = (number >> 1);                 // сдвигаем
        ++i;
    }
    return binary;
}
```

Выведем с помощью этой функции двоичное представление чисел `int8_t`:

```
int8_t yy;
yy = +1;   cout << setw(3) << int(yy) << ' ' << to_bits(yy) << endl;
yy = +127; cout << setw(3) << int(yy) << ' ' << to_bits(yy) << endl;
yy = -1;   cout << setw(3) << int(yy) << ' ' << to_bits(yy) << endl;
yy = -128; cout << setw(3) << int(yy) << ' ' << to_bits(yy) << endl;
```

Получаем в консоли результат:

```
  1 00000001                  // +1   старший бит 0
127 01111111                  // +127 старший бит 0
 -1 11111111                  // -1   старший бит 1
-128 10000000                 // -128 старший бит 1
```

Вывод показывает, что положительные числа имеют в старшем бите ноль, а отрицательные — единицу.

От количества битов этот факт не зависит. Все целые знаковые числа устроены одинаково. Поэтому 32-битное число -1 (минус единица) состоит из 32 единиц, а 64-битное — из 64 единиц.

Интересно, что ноль знака не имеет. Мы можем писать в коде и +0, и -0, но в памяти оба нуля представлены нулевым битами.

Как мы видим, битовая структура знаковых целых и беззнаковых целых одна и та же. А теперь вернемся к проблемам с целыми числами (см. урок 15):

```
int8_t   sbyte;                 // знаковое: от -128 до +127, размер 1
uint8_t  ubyte;                 // беззнаковое: от 0 до 255,  размер 1
sbyte = -1; ubyte = -1;
cout << int(sbyte) << ' ' << int(ubyte) <<'\n' ;
sbyte = -128; ubyte = -128;
cout << int(sbyte) << ' ' << int(ubyte) <<'\n' ;
```

В консоль выводится:

```
-1 255          // двоичные 11111111 11111111
-128 128        // двоичные 10000000 10000000
```

Двоичное представление знаковых и беззнаковых чисел сразу объясняет, почему получаются такие странные, на первый взгляд, результаты:

беззнаковое $255_{10} = 11111111_2$

знаковое $-1_{10} = 11111111_2$

беззнаковое $128_{10} = 10000000_2$

знаковое $-128_{10} = 10000000_2$

Таким образом, получается, что фактического преобразования не происходит.

Сложим +127 и -128 в двоичном коде:

```
  01111111                      +127
+ 10000000                      -128
  11111111                      -1
```

Результат получился именно тот, который и должен быть, — минус единица. Но складывали-то мы числа как беззнаковые целые! Запишем теперь для сложения +1 и -1 (удобнее первым слагаемым взять -1):

```
  11111111                      -1
+ 00000001                      +1
```

Даже складывать не требуется, т. к. сразу видно, что это совершенно то же самое, что и сложение 255 + 1 для беззнаковых (см. выше). И результат тоже равен нулю с «вылетом» старшего разряда из разрядной сетки (8 бит).

Теперь разберемся, как происходит переполнение при сложении знаковых целых чисел. Сложим +127 и +1 в двоичном виде:

```
  01111111                      +127
+ 00000001                      +1
  10000000                      -128
```

Перенос попал в знаковый разряд, и получилось отрицательное число. Это и есть переполнение знаковых целых.

Сдвиг вправо знакового целого числа

Чтобы понять, как происходит сдвиг вправо целых чисел знаком, напишем несколько операторов:

```
int8_t mm(0b10000000);                                  // число -128
cout << setw(4) << int(mm) << ' ' << to_bits(mm) << endl;
cout << setw(4) << int(mm >> 1) << ' '
     << to_bits(int8_t(mm>>1)) << endl;
mm = 0b01111111;                                        // число +127
cout << setw(4) << int(mm) << ' ' << to_bits(mm) << endl;
cout << setw(4) << int(mm >> 1) << ' '
     << to_bits(int8_t(mm>>1)) << endl;
```

Получим в консоли вывод:

```
-128 10000000
 -64 11000000
 127 01111111
  63 00111111
```

Этот вывод демонстрирует, что при сдвиге вправо целых чисел со знаком происходит копирование знакового бита: если он был 0, то слева добавляется ноль. Если число отрицательное, то дублируется единица (рис. 16.4).

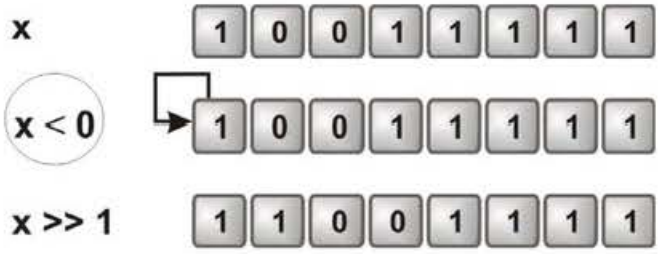


Рис. 16.4. Сдвиг вправо отрицательного целого числа

И такой сдвиг тоже делит нацело на 2 знаковые числа.

Дополнительный код

Складывать двоичные числа мы умеем, теперь нужно научиться вычитать. Однако в компьютерах операция вычитания целых заменена сложением:

```
n - m = n + (-m)
```

Здесь требуется уметь вычислять противоположное число. Рассмотрим эту операцию на примерах с целыми типа `int8_t`. Пусть имеется целое число `+1`: 00000001_2. Чтобы получить противоположное число `-1`, надо выполнить всего две операции:

- инвертировать число с помощью битовой операции `~` (`not`);
- прибавить 1.

Вот как это выглядит:

```
 00000001                    // +1
~00000001                    // операция инвертирования
 11111110                    // обратный код
+00000001                    // прибавить 1
 11111111                    // -1 дополнительный код
```

Окончательный результат, как мы видим, равен `-1`. Говорят, что знаковые целые числа представлены в памяти в *дополнительном* коде. Инвертирование числа дает *обратный* код.

Тем же способом можно из отрицательного числа получить противоположное ему положительное.

Задачи и упражнения

1. Напишите функцию перевода беззнакового целого `toOctal` в восьмеричную систему и обратную функцию `fromOctal` для перевода в десятичную систему.
2. Напишите функцию `toHex` для перевода целого в шестнадцатеричную систему и обратную функцию `fromHex` для перевода в десятичную систему.
3. Перепишите функции `to_binary` и `to_decimal`, используя стандартные функции `to_chars` и `from_chars`.
4. Реализуйте функцию сложения длинных двоичных чисел. Модифицируйте ее для сложения восьмеричных и шестнадцатеричных чисел.
5. Напишите функцию вычитания длинных двоичных чисел (используйте дополнительный код).

- Урок 17 -

Опять функции

И снова мы возвращаемся к функциям. Мы уже познакомились с рекурсией, перегрузкой, шаблонами, constexpr-функциями. Однако функции в C++ богаче, чем в других языках программирования. Одной из обязательных возможностей в любом языке программирования является использование функций как параметров в других функциях. Это позволяет писать более универсальный код. Как обычно, эта часть языка имеет средства, унаследованные от языка C, и средства, добавленные в C++. Особенно полезными оказались так называемые лямбда-выражения.

Параметры по умолчанию

Выше (см. *урок 7*) я уже писал, что список аргументов функции при вызове должен точно соответствовать списку формальных параметров, но возможны отступления от этого общего правила. В C++ разрешается задавать значения параметров по умолчанию. Для простых типов при передаче параметров по значению это выглядит так:

```
тип имя=выражение
```

Как мы видим, объявление параметра совпадает по форме с инициализацией переменных. Но C++ допускает другие виды инициализации переменной:

```
тип имя (выражение)
тип имя {выражение}
```

Такая форма присвоения значения для параметров встроенных типов не допускается.

Параметры со значениями по умолчанию должны объявляться последними (крайними правыми) параметрами в списке. При вызове функции такие параметры можно не указывать — используется заданное по умолчанию. Если же значение фактического параметра явно задано при вызове, то оно и используется.

В качестве примера напишем функцию, которая выводит ряд натуральных чисел, пропуская числа, кратные заданному. На месте пропускаемых чисел выводится строка минусов. Назовем функцию `skipnumber`.

Параметров в функции определим два:

- количество проверяемых чисел;
- проверяемая кратность числа.

Таким образом, заголовок функции будет такой:

```
void skipnumber(int n, int d)
```

Код представлен в листинге 17.1.

Листинг 17.1. Функция с параметрами по умолчанию

```
#include <iostream>
using namespace std;
void skipnumber(int n = 10, int d = 3)  // значения по умолчанию
{  int count {0};
   while(count < n){
      ++count;
      if (count % d)
           cout << count << endl;
      else cout << "---" << endl;
   }
}
int main(){
   skipnumber();            // 10 чисел, не кратных 3
   skipnumber(16);          // 16 чисел, не кратных 3
   skipnumber(14,6);        // 14 чисел, не кратных 6
   return 0;
}
```

В функции `main` написано три вызова. В первом случае в консоль выводится 10 чисел с пропуском чисел, кратных 3. Второй вызов показывает 16 чисел, среди которых опять же пропущены кратные 3. И в третий раз выводится 14 чисел с пропусками тех, которые кратны 6.

Вызов функции с параметрами по умолчанию считается корректным, если не указываются самые правые параметры. Язык C++ запрещает писать вызовы с пропущенными первыми и заданными последними параметрами.

Поэтому вызов:

```
skipnumber(,5);
```

писать нельзя.

Аналогично при определении не допускается такой заголовок:

```
void skipnumber(int n = 10, int d)
```

Значения параметров по умолчанию можно задавать и для реализованных типов, но в классе должны быть реализованы конструкторы инициализации.

Функции как параметры

Ранее, при изучении способов передачи массивов в функции (см. *урок 12*) мы уже рассматривали некоторые стандартные функции из библиотеки STL, которые при-

нято называть *стандартными алгоритмами*. И, в частности, упоминали такие алгоритмы, как `count`, `replace`, `find` и т. п. Практически у всех таких алгоритмов существует версия с дополнительным параметром. Например, для алгоритма `find` имеется родственный ему алгоритм `find_if`, который имеет прототип:

```
template< class InputIt, class UnaryPred >
InputIt find_if( InputIt first, InputIt last, UnaryPred p );
```

Обратите внимание на третий параметр — это функция-предикат, которая определяет условие поиска. Причем параметр у нее единственный, о чем говорит тип параметра `UnaryPred`.

Напишем функцию, определяющую четность целого числа:

```
bool isEven(int k){
  return !(k & 0b1);                    // преобразование int -> bool
}
```

Тогда мы можем использовать эту функцию в таком контексте:

```
vector<int> v {1,3,5, 4,6,9};
auto it = find_if(begin(v), end(v), isEven);          // поиск слева
cout << distance(begin(v),it) << ' ' << *it << endl;
auto rt = find_if(rbegin(v), rend(v), isEven);        // поиск справа
cout << distance(rbegin(v),rt) << ' ' << *rt << endl;
```

Этот код выдаст в консоль следующие числа:

```
3 4
1 6
```

Первые два числа — это индекс и значение первого четного числа в векторе `v`, поиск которого начинался с начала вектора. Итератор `it` — прямой итератор. Следующие два числа — это индекс и значение четного числа, поиск которого осуществлялся с конца вектора. Итератор `rt` — обратный итератор, и он отсчитывается с конца вектора, начиная тоже с нуля.

Функция `find_if`, «пробегая» по вектору, применяет функцию `isEven` к каждому элементу. Она возвращает итератор первого элемента, для которого `isEven` выдала `true`.

В набор стандартных алгоритмов входит алгоритм `for_each`, с помощью которого можно обработать все элементы любого из видов массивов (встроенный, `array`, `vector`). Например, можно удвоить все элементы вектора `v`, показанного выше, написав функцию:

```
void two(int &k){ k = (k*2); }
```

и задав ее третьим аргументом в `for_each`:

```
for_each(begin(v), end(v), two);
```

Функция `for_each`, передвигаясь по элементам вектора `v`, запускает функцию `two`, передавая ей текущий элемент. Функция `two` удваивает свой параметр. А так как он передается по ссылке, то происходит удвоение элемента вектора.

Функциональные объекты

Мы уже не один раз обращались к алгоритму `accumulate`. Ранее я упоминал, что функция `accumulate` «умеет» не только складывать, но и умножать (см. *урок 12*). Это можно сделать разными способами. Например, напишем функцию умножения, как мы уже делали:

```
int multiply(int a, int b){ return a*b; }
```

Укажем ее вызов в вызове функции `accumulate`:

```
int64_t S = accumulate(begin(v), end(v),1, multiply);
```

Третий параметр равен, конечно, не нулю, а единице. Если вектор `v` содержит элементы `{1,2,3,4,5}`, то произведение `S` получится равным `120`.

Однако нам нет необходимости писать функцию умножения. В стандартной библиотеке реализованы так называемые *функциональные объекты*, которые и предназначены для применения в подобных случаях. С функциональным объектом вместо функции вызов `accumulate` выглядит так:

```
int64_t P = accumulate(begin(v), end(v),1, multiplies<int>());
```

Результат получается тот же самый: произведение равно `120`.

Такие функциональные объекты реализованы в STL для всех логических, арифметических и битовых операций. Операции сравнения тоже реализованы в таком виде.

Указатель на функцию

А теперь давайте разберемся, как написать функцию в качестве параметра другой функции. Например, попробуем написать более простую версию стандартного алгоритма `for_each`, которая могла бы обрабатывать любую последовательность элементов массива. Операция обработки — это и есть функция, передаваемая в качестве параметра и выполняющая модификацию одного элемента массива заданным образом. Наша «главная» функция должна применить эту операцию (вызвать функцию) для каждого элемента массива.

Язык C++ разрешает передавать функции в качестве параметра. Для этого среди параметров (лучше первым или последним) «главной» функции надо указать прототип функции-параметра. Очевидно, что параметром функции-параметра в данном случае должен быть элемент массива, а результат — того же типа, что и элементы массива.

Напишем шаблон. Его прототип должен выглядеть так:

```
template <class T> void ForEach(T *begin, T *end, T f(T elem));
```

Проверять параметры не будем, чтобы не отвлекаться от главного. Код шаблона «главной» функции приведен в листинге 17.2.

Листинг 17.2. Шаблон *ForEach*

```
template <class T>
void ForEach(T *begin, T *end, T f(T elem))
{ for (; begin < end; ++begin) *begin = f(*begin); }
```

Выглядит шаблон, на удивление, очень простым. Это обычная картина при программировании: если исходные принципы хороши, то программирование существенно упрощается.

Теперь определим реальные функции и проверим работу нашего шаблона:

```
int f2 (int a)                          // удвоение параметра
{ return (a *= 2); }
double f3 (double a)                    // +2 к параметру
{ return (a += 2); }
```

В функции `main` напишем код:

```
int a [] = { 1, 2, 3, 4, 5, 16, 7, 8, 9, 10 };
ForEach (a, a + 10, f2);
copy(begin(a),end(a),ostream_iterator<int>(cout," ")); cout << '\n';
array<double,size(a)> b { 10, 12, 13, 14, 15, 16, 17, 18, 19, 20 };
ForEach (begin(b), end(b), f3);
copy(begin(b),end(b),ostream_iterator<int>(cout," ")); cout << '\n';
```

На экран выводится совершенно правильный результат в обоих случаях:

```
2 4 6 8 10 32 14 16 18 20
12 14 15 16 17 18 19 20 21 22
```

Таким образом, мы написали универсальную функцию для обработки любой последовательности однотипных элементов в массивах.

С параметрами-функциями возникает множество вопросов, например: каким способом передается параметр-функция? Очевидный отрицательный ответ — не по значению. Тело функции не попадает в стек — это совершенно ясно. Тогда вариантов два: по ссылке и по указателю. И опять надо вспомнить, что еще в языке C были реализованы функции-параметры, когда ссылок не было. Тогда становится совершенно очевидно, что, как и для массивов, функция передается как параметр-указатель (на функцию). Имя функции-аргумента при вызове принимающей функции преобразуется в указатель на функцию.

Лямбды

Самое полезное нововведение C++11, которое в C++14 и C++17 улучшалось и совершенствовалось — это *лямбда-выражения* (которые часто называют анонимными функциями). Мы будем называть их просто *лямбды*.

Основная идея нововведения состоит в том, чтобы позволить не определять функции, передаваемые в качестве параметра, а писать их непосредственно на месте аргумента.

Давайте перепишем приведенные выше примеры с использованием лямбд. Поиск четных элементов вектора выглядит так:

```
vector<int> v {1,3,5, 4,6,9};
auto it = find_if(begin(v),end(v),[](int k){ return !(k & 0b1);});
cout << distance(begin(v), it) << ' ' << *it << endl;
auto rt = find_if(rbegin(v),rend(v),[](int k){ return !(k & 0b1);});
cout << distance(rbegin(v), rt) << ' ' << *rt << endl;
```

Посмотрите, в третьем аргументе вместо имени функции написана сама функция в несколько необычном виде:

```
[](int k){ return !(k & 0b1);}
```

Функция анонимная (нет имени) и начинается с квадратных скобок `[]`. После них задан тот же список параметров и потом то же самое тело, которое было написано в определении функции выше.

Формальный синтаксис лямбда-выражения такой:

```
[](параметры){ действия }
```

Лямбда-выражение начинается с квадратных скобок `[]` — это отличительный признак лямбды.

> ***На заметку***
>
> Квадратные скобки называются «страшным» словом «интродуктор», но мы в дальнейшем будем называть их просто «захват» в соответствии с исполняемой ими функцией (см. ниже).

Затем, как и в обычной функции, в круглых скобках задаются параметры.

Начиная со стандарта C++14 для параметров можно указывать значения по умолчанию. После списка параметров, как и в обычной функции, в фигурных скобках прописываются действия, выполняемые лямбда-выражением.

Приведем еще несколько примеров. Напишем вызов `for_each` с лямбдой:

```
for_each(begin(v), end(v), [](int &k){ k = k * 2;});
```

И снова мы видим, что после захвата фактически повторяется написанная ранее функция `two`.

И вызов `accumulate` тоже легко модифицируется:

```
int64_t S = accumulate(begin(v),end(v),1,                    // произведение
                       [](int a, int b){ return a*b; } );    // лямбда
```

Результат для `v = {1,2,3,4,5}`, как и полагается, тоже равен `120`.

Лямбду в стандартных алгоритмах можно писать даже с параметрами `auto`:

```
double S = accumulate(begin(v),end(v),1.0,                   // 1.0 - double
                      [](auto a, auto b){ return a*b; } );   // лямбда
```

Компилятор правильно разбирается с типами таких параметров. Но обратите внимание на константу `1.0` — она имеет тип `double`, как и результат `S`.

Несомненно, для программиста гораздо удобнее писать операцию обработки непосредственно на месте параметра, а не определять отдельную функцию.

Ну, а как же должен выглядеть параметр в определении функции, чтобы на его месте можно было написать лямбду? Такой параметр выглядит как обычный указатель на функцию:

```
void Do(int a, int b, int work(int, int)){
    cout << work(a, b) << endl;          // выполнение операции
}
```

Вызов с лямбдой выглядит так:

```
Do(7, 4, [](int x, int y){ return x + y; });
```

Результат, который выводится, естественно, равен `11`.

Лямбда-функции

Довольно часто в практике программирования возникают ситуации, когда нужно написать функцию внутри функции. Эта внутренняя функция нужна именно внутри вызывающей функции, она должна выполнить какую-то внутреннюю работу, и больше она нигде не нужна. Раньше программистам приходилось изыскивать обходные пути для преодоления этого запрета.

Теперь такого запрета в C++ нет. Лямбды можно объявлять как переменные. И это сразу снимает вопрос о вложенных функциях. Неопытный программист может подумать: какая разница, функция вложена или нет — главное, чтобы функция была определена. Но я вам как опытный программист скажу, что в больших проектах очень важно придерживаться принципа *инкапсуляции*, о котором уже было несколько раз упомянуто ранее.

> ***Совет***
>
> Всегда старайтесь придерживаться принципа инкапсуляции в своем коде.

И в C++ есть средства, предназначенные именно для поддержки этого принципа: например, классы и пространства имен. Внутренние локальные функции помогают поддерживать инкапсуляцию на уровне имен функций.

В наших программах это тоже можно использовать. Напомню, что мы написали программу «Числа — прописью» (см. *урок 13*), и там реализована функция `Word`, которая добавляла нужное окончание к словам «миллион», «миллиард», «триллион» (см. листинг 13.10). Так как эта функция больше нигде не нужна, ее лучше сделать внутренней локальной функцией в функции `toTread` (см. листинг 13.11). Код новой версии представлен в листинге 17.3.

Листинг 17.3. Лямбда как вложенная функция

```
string toTread(uint16_t number, string_view word)
{ // -- массивы числительных --
  auto Word = [](uint16_t tread, string word) {        // лямбда-функция
    if (tread / 10 % 10 == 1) word += "ов";
    else switch (tread % 10)
    {  case 0: case 5: case 6:
       case 7: case 8: case 9: word += "ов"; break;
       case 2: case 3: case 4: word += "а";  break;
    }
    return (word + ' ');
  };
    string result;                                     // строка-результат
```

```
    // ----- прежний код ----------------------
    if(size(word) > 0){
        string cword{word};
        if(cword != "тысяч"s)
            result += Word(number, cword);              // вызов лямбды
        else{
            // -- прежний код --
        }
    }
    return result;
}
```

Как видите, функция-переменная объявляется так:

```
auto имя = лямбда-заголовок {
        тело
};              // конец объявления переменной!
```

Тип лямбды всегда указывается как `auto`. Обращаю внимание на точку с запятой в конце — это переменная, причем с инициализацией! Как для любой переменной, инициализацию можно написать в универсальном стиле:

```
auto имя { лямбда-заголовок { тело } };
```

Еще нужно обратить внимание на то, что мы ни разу в примерах не указали, какой тип возвращает лямбда. Просто писали оператор возврата, и этого было достаточно — компилятор ошибок не выдавал. Это означает, что компилятор самостоятельно выводит тип возвращаемого значения на основании кода лямбды. Однако можно это задать явным образом:

```
auto имя { лямбда-заголовок -> тип { тело } };
```

Для лямбды `Word` это выглядит так:

```
auto Word = [](uint16_t tread, string word) -> string {...};
```

Такой вариант объявления типа возврата теперь можно задавать не только в лямбдах, но и в заголовках обычных функций. И даже заголовок функции `main` можно написать в таком же виде:

```
auto main() -> int
```

И это тоже указано в стандарте C++, но, конечно, практически всегда пишут заголовок `main` в старом привычном стиле.

Как вы видели в листинге 17.3, вызов лямбда-функции ничем не отличается от вызова обычной функции:

```
имя(аргументы)
```

Как и обычные функции, лямбды можно объявлять `constexpr`, например:

```
constexpr auto square = [](int n) constexpr { return n * n; };
```

Переменная `square` является `constexpr` лямбда-выражением и вычисляется на этапе компиляции. Это легко проверить, задав в главной функции операторы:

```
constexpr int result = square(5);
static_assert(result == 25, "Результат должен быть равен 25");
```

Напомню: достаточно написать логическое выражение, значение которого равно `false`, и сообщение будет выдано компилятором как ошибка.

Напоследок скажу, что использование лямбд повышает читабельность кода и его компактность. Поэтому рекомендую писать лямбды как можно чаще.

Теперь осталось разобраться только с захватом.

Захват

Захват позволяет лямбдам использовать переменные из контекста, где лямбда определена. Например, внутренняя лямбда может захватить локальные переменные функции, внутри которой она определена. Но, конечно, эти переменные должны быть видны в точке захвата. Захват можно задавать в разных вариантах:

- `[=]` — захват всех видимых переменных по значению;
- `[&]` — захват всех видимых переменных по ссылке;
- `[var]` или `[&var]` — захват конкретной переменной по значению или по ссылке.

Ранее мы писали (см. *урок 14*, листинг 14.11) рекурсивную функцию поиска минимума массива с трассировкой вызовов. Теперь можно всю трассировку локализовать внутри функции поиска минимума массива (листинг 17.4).

Листинг 17.4. Внутренние анонимные лямбды с захватом

```
int mini(int a[], int left, int right)
{  static int tt = 0;                    // счетчик уровня вложенности
   auto pp = [](int L, int R){           // лямбда вывода точек
       int i{0}; while(i < tt){ cout << "."; ++i; }
       cout << "mini(" << L << ',' << R << ')' << endl;
   };
   if (left == right) return a[left];
   //  анонимная лямбда и ее вызов - вложенность увеличивается
   [&](int L, int R){ ++tt; pp(L,R); }(left, right);
   int x = mini(a, left+1, right);
   //  анонимная лямбда и ее вызов - вложенность уменьшается
   [&](int L, int R){ --tt; pp(L,R); }(left, right);
   if (x < a[left]) return x; else return a[left];
}
```

Прежняя функция `pp` разделена на две части. Лямбда `pp` занимается теперь только выводом в консоль. Для увеличения и уменьшения счетчика вложенных выводов написаны две безымянные лямбды, которые вызваны прямо в точке определения:

```
[&](int L, int R){ ++tt; pp(L,R); }(left, right);      // увеличиваем
[&](int L, int R){ --tt; pp(L,R); }(left, right);      // увеличиваем
```

Можно было бы написать одну поименованную, но я специально написал две отдельно, чтобы акцентировать ваше внимание. Обратите внимание на захват всех

переменных по ссылке. Это значит, что внутри лямбды можно изменять захваченные переменные (что и делается с переменной-счетчиком). Но самое интересно, что здесь же захватывается и лямбда pp! И только поэтому эту лямбду можно вызвать внутри безымянной. Если мы напишем захват только переменной tt, то компиляция не пройдет: компилятор напишет, что pp не захвачена (pp is not captured).

Вызов функции mini с теми же данными, что и ранее:

```
int numbers[]{3, 2, 4, 5, 7, -2, 6, 7, 9, -1};
auto x = mini(numbers, 0, size(numbers)-1);
cout << x << endl;
```

приводит к тем же результатам (см. рис. 14.4).

Перегрузка операций

Язык C++, помимо перегрузки функций, разрешает и перегрузку операций, о которой мы уже не раз упоминали (см. *урок 3*) и которую неоднократно наблюдали. Например, операция «+» перегружена для строк как операция сцепления, а операции сдвига (см. *урок 16*) перегружены для ввода/вывода.

Перегрузка операций разрешена почти для всех операций (кроме некоторых). Причем разрешается перегружать почти все операции (кроме четырех) и как функцию, и как метод класса. Механизм перегрузки имеет ограничения:

1. Нельзя никаким образом изменить приоритет операции.
2. Нельзя никаким образом изменить количество аргументов — унарные операции должны иметь один аргумент, бинарные — два.
3. Операндами перегруженной операции не могут быть встроенные типы.

Давайте перепишем функцию сложения длинных чисел (см. листинг 13.12) как операцию сложения. Определим синоним типа (см. *урок 13*, задание 3):

```
using biginteger = std::string_view;
```

Тогда прототип функции можно сделать таким:

```
biginteger operator+(biginteger a, biginteger b);
```

Фактически изменено только имя функции: вместо add задано operator+. Подобные имена являются стандартными в C++ и имеют общий формат:

```
operator@
```

Вместо символа @ задается символ реализуемой операции.

Код операции сложения показан в листинге 17.5.

Листинг 17.5. Перегрузка операции сложения для длинных чисел

```
biginteger operator+(biginteger a, biginteger b){
    uint8_t  cf = 0;                                  // перенос
    uint8_t digit;                                    // цифра короткого
    biginteger lesser = (size(a) < size(b))?a :b;     // короткое
    biginteger larger = (size(a) < size(b))?b :a;     // длинное
```

```
    string result {larger};                                  // результат
    auto next_lesser = rbegin(lesser);                       // справа налево
    auto next_result = rbegin(result);                       // справа налево
    while(next_result != rend(result)){                      // по длинному числу
        if(next_lesser != rend(lesser)){                     // короткое еще есть
            digit = (*next_lesser -'0');                     // цифра короткого
            ++next_lesser;                                   // к следующей цифре
        }else digit = 0;                                     // цифр короткого нет
        uint8_t ds = (*next_result - '0') + digit + cf;
        cf = ds / 10;                                        // текущий перенос
        *next_result = (ds % 10) + '0';                      // цифра результата
        ++next_result;                                       // к следующей цифре
    }
    if(cf > 0) {                                  // перенос из старшего разряда
        digit = (cf + '0');                       // дополнительная цифра
        result = char(digit) + result;
    }
    return result;
}
```

Я специально привел полный код, чтобы вы воочию убедились: код тот же самый (кроме инициализации `result`), что и в листинге 13.12. Таким образом, перегрузка операций — это такое же написание функции, как и прежде.

Задачи и упражнения

1. Реализовать функцию из задания 1 *урока 7*. Функцию сделать параметром в функциях заданий 1.1 и 1.2.
2. Реализовать функцию из задания 1 *урока 7* как внешнюю лямбду, которую передавать как параметр в функции заданий 1.1 и 1.2.
3. Реализовать функцию-предикат задания 2 *урока 7* как внутреннюю лямбду в функции поиска простых чисел от A до B.
4. Реализовать функцию преобразования массива. Параметрами являются итераторы массива. Третий параметр — функция-предикат, которая задает критерий отбора элементов (см. выше `find_if`). Четвертый параметр — функция-модификатор элемента массива, если он удовлетворяет критерию отбора. Критерий отбора и модификатор реализовать в виде лямбд при вызове.
5. Реализовать сортировку выбором (см. задание 6 *урока 9*) как функцию с тремя параметрами. Первые два — итераторы по массиву, третий — функция-предикат сравнения элементов. При вызове функцию-предикат задавать в виде лямбды.

- Урок 18 -

Снова указатели

Мы познакомились с указателями (см. *урок 11*), однако один важный вопрос так и не понятен до конца: а зачем все-таки они нужны? Как выяснилось (см. *урок 11* и *12*), указатели являются итераторами по массивам и позволяют писать более универсальные функции для работы с массивами. Второй довод в пользу указателей «из той же оперы»: уменьшить расходы памяти при вызове функции, если параметром является массив, — в функцию передается указатель, размеры которого всегда намного меньше массива.

Однако этого явно недостаточно, чтобы заводить в языке дополнительную довольно сложную сущность. Конечно, есть значительно более весомая причина: указатели используются для работы с *динамическими переменными*. Такие объекты создаются и уничтожаются явным образом во время выполнения программы, и доступ к таким переменным осуществляется только с помощью указателей. Поэтому рассмотрим более детально, как в C++ осуществляется работа с памятью.

Память и объекты

Каждый объект C++ является *областью памяти*. Объект обладает рядом свойств, например имеет тип и может иметь имя. Объекту приписывается *период хранения* в памяти (storage duration). Это свойство определяет время жизни объекта. Различается четыре периода хранения и, соответственно, — четыре класса памяти для размещения объектов. Нас интересует три из них:

- статическая память (static storage duration);
- автоматическая память (automatic storage duration);
- динамическая память (dynamic storage duration).

В *статической памяти* размещаются все глобальные объекты, объявленные вне каких-либо функций и классов, а также любые статические переменные, в том числе объявленные в функциях (и в классах). Память для этих объектов выделяется в момент запуска программы, а уничтожаются они только при завершении программы. Все глобальные объекты инициализируются по умолчанию нулевыми значениями.

В *автоматической памяти* размещаются локальные нестатические объекты, которые объявляются в любом блоке, в том числе в теле функции. Память таким объектам выделяется в момент объявления, а уничтожаются они в конце их области видимости. Мы уже не раз упоминали, что для размещения локальных переменных в функциях используется *стек* (см. *урок 7* и *14*), который является практической реализацией автоматической памяти. Локальные переменные по умолчанию *не инициализируются*.

Динамическая память (которую часто называют *свободной* памятью или *кучей*) используется для создания динамических объектов. Такие объекты создаются и уничтожаются по мере необходимости во время выполнения программы. Для создания динамических переменных используется операция `new`, для уничтожения — операция `delete`. Динамические массивы создаются операцией `new[]`, а уничтожаются операцией `delete[]`.

Для работы с динамической памятью нужно подключить библиотеку `<new>`.

Динамическая память для встроенных типов

Сначала создадим динамические объекты встроенных типов, например:

```
int *pi = new int{1024};                // универсальная инициализация
double *pd = new double(1.2345);
```

Эти объекты сразу инициализированы. Объект целого типа инициализирован универсальным способом, а второй — устаревшим, который был определен еще до стандарта C++. При создании объект можно обнулить:

```
int *pi0 = new int();
int *pi1 = new int{};
```

Без явной инициализации в переменной может оказаться «мусор», и поведение программы будет непредсказуемо.

Теперь с этими динамическими объектами мы можем работать. Как обычно, для доступа к значению используется операция разыменования:

```
*pi0 = 2;
```

Объявление указателя и создание переменной не обязательно делать в одном операторе. Объявим в программе указатель:

```
int *p;
```

В нужном месте программы мы можем создать динамическую переменную:

```
p = new int;
```

Переменная создана без инициализации только для того, чтобы показать такую возможность. Постарайтесь так не делать, т. к. значение `*p` в данном случае может быть каким угодно. Если мы используем это значение до инициализации, то получим неопределенное поведение (UB).

Когда объект перестанет быть нужен, мы можем удалить его:

```
delete pi;
pi = nullptr;           // обнуление указателя
```

Операция `delete` сама не обнуляет указатель, и я написал это в явном виде. Обнулять указатель необязательно, но если этого не сделать, то там останется адрес объекта, который уже уничтожен. Если попытаться как-то использовать этот указатель в программе, то получим неопределенное поведение (UB).

Таким образом, для встроенных типов операция `new` выделяет память, записывает адрес в указатель и, если необходимо, инициализирует объект. Динамический объект не имеет своего явного имени и доступ к нему только косвенный — по указателю. Видимость динамического объекта определяется видимостью указателя.

Операция `delete` возвращает память системе (и, тем самым, уничтожает объект). Поэтому время жизни динамических объектов встроенных типов — от операции `new` до операции `delete`.

Динамическая память для реализованных типов

Динамически создавать можно объект любого типа, в т. ч. и реализованный в виде класса (структуры). Мы уже писали класс `Date` (см. *урок 10*). Можно объявить указатель типа `Date` и создать динамическую переменную:

```
Date *pdd = new Date;
Date *pdd = new Date();
Date *pdd = new Date{};
```

Все три способа работают одинаково. При создании объекта в этом случае вызывается конструктор без аргументов (конструктор по умолчанию), если он в классе есть. Иначе возникнет ошибка компиляции. Убедиться в этом можно, написав конструктор с выводом сообщения:

```
Date() { cout << "default" << endl; };
```

При запуске программы любой из трех способов создания объекта приводит к выводу сообщения `default`. Вызов конструктора — это важное отличие создания динамических объектов встроенных типов и реализованных в виде класса. Если реализован конструктор инициализации, то динамический объект можно инициализировать:

```
Date *today = new Date{2024,6,17};
```

Методы динамического объекта можно вызывать двумя способами. Первый вариант — использовать селектор (точка) и операцию разыменования (*):

```
(*today).print();
```

Скобки ставить обязательно, т. к. приоритет операции «точка» (селектор) выше приоритета операции разыменования.

Второй способ обеспечивается операцией доступа по указателю ->:

```
today->print();
```

Если у вас структура или в классе открытые поля, то «добраться» до полей можно теми же способами:

```
(*today).year = 2024;
today -> month = 12;
```

Для уничтожения динамических объектов, реализованных в виде класса, тоже применяется операция `delete`. Но в этом случае ее работа отличается от работы `delete` для встроенных типов.

Деструктор

При создании объекта работает конструктор, который в любом классе всегда присутствует: или создается по умолчанию, или определяется явно. Однако объекты не только «рождаются», но и «умирают». Поэтому «для симметрии» в классе нужно иметь средства, которые работают при уничтожении объекта. В C++ есть стандартный механизм для этого — *деструктор*. Конструктор создает объект, деструктор его уничтожает.

Деструктор — это метод, имеющий имя класса, но первым символом должен быть символ «~» (тильда). Деструктор не имеет параметров и не возвращает результата. Деструктор может быть только один — перегрузка деструкторов не разрешается. Если деструктор не определен явно, он создается системой автоматически. Деструктор по умолчанию имеет вид:

```
~имя_класса(){}
```

В классе `Date` явный деструктор нам не требовался, поэтому он создавался в классе автоматически.

Деструктор вызывается автоматически *при уничтожении* объекта класса. Для локальных объектов это делается при выходе из блока, а для динамических, которые создаются операцией `new`, — при каждом вызове операции `delete`.

Для того чтобы отследить вызов деструктора, напишем в классе `Date` деструктор с выводом сообщения. Код класса и функции `main`, где создаются и уничтожаются объекты типа `Date`, представлен в листинге 18.1.

Листинг 18.1. Вызов конструктора и деструктора класса

```
#include <iostream>
using namespace std;
class Date {
    public:
    Date() { cout << "default" << endl; }              // конструктор
    Date(unsigned y, unsigned m, unsigned d){          // конструктор
        cout << "initial" << endl;
        year = y; month = m; day = d;
    }
    ~Date(){ cout << "destructor" << endl; }           // деструктор
private:
    unsigned year{2024}, month{5}, day{1};             // по умолчанию
};
int main(){
  Date *today = new Date{2024,6,17};    // 1 - конструктор инициализации
  delete today;                         // 2 - деструктор вызывается
```

```
  Date *def = new Date{};                    // 3 - конструктор без аргументов
  delete def;                                // 4 - деструктор вызывается
  Date t;                                    // 5 - конструктор без аргументов
  return 0;                                  // 6 - деструктор вызывается
}
```

Как видно по коду, первым вызывается конструктор инициализации при создании динамического объекта с указателем `today`. Операция `delete today;` вызывает деструктор (строка `// 2`). Затем создается динамический объект с указателем `def`. Операция `delete def;` вызывает деструктор (строка `// 4`).

Последним создается локальный объект `t`, который уничтожается при завершении функции `main` (строка `// 6`). При этом тоже вызывается деструктор.

Запустив программу, увидим в консоли порядок работы конструкторов и деструктора (рис. 18.1).

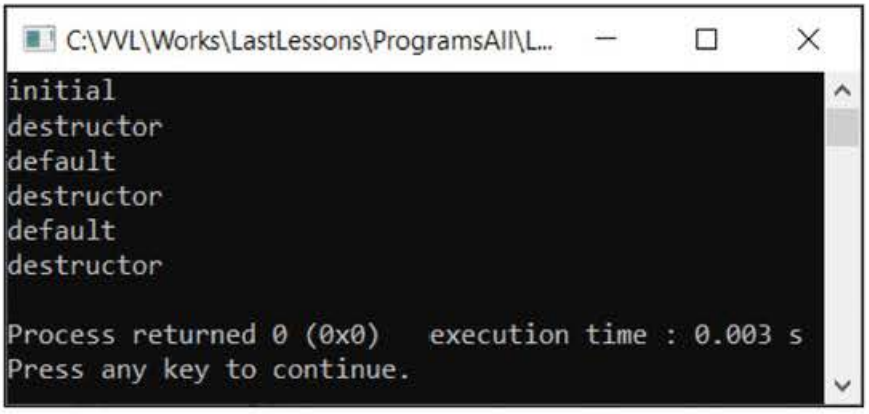


Рис. 18.1. Последовательность вызовов конструкторов и деструктора

Обычно деструктор явно реализуется только тогда, когда при уничтожении требуется выполнить некоторые специальные действия.

Динамические массивы

Практически основное использование динамической памяти — это создание динамических массивов. Динамические массивы создаются точно так же, как и динамические объекты (операция `new[]`):

```
double *dynamic = new double[1000];
```

Массив, естественно, можно обнулить при создании, написав после квадратных скобок круглые или фигурные:

```
double *dyn0 = new double[1000]();
double *dyn1 = new double[1000]{};
```

Доступ к элементам осуществляется так же, как и для уже изученных нами массивов: по индексу или с помощью указателя. Естественно, функция `size` не работает с динамическими массивами, т. к. имеем указатель, а не массив (как при передаче массива в функцию).

Динамический массив уничтожается операцией `delete[]`:

```
delete[] dynamic;
```

Квадратные скобки `[]` писать обязательно. Если вы создаете динамический массив, то работает операция `new[]` со скобками. Поэтому и удалять массив нужно операцией `delete[]` со скобками (иначе получаем неопределенное поведение).

Для динамического массива можно задавать размер выражением. Если `N` — переменная целочисленного типа, значение которой присвоено любым допустимым способом (например, введено из файла или вычислено), то объявить динамический массив любого встроенного типа можно так:

```
double *dynamic = new double[N]{};
```

Массив точно так же обнуляется, как и массив фиксированного размера. С таким массивом дальше можно работать как с массивом фиксированного размера. Но функцию `size` и в этом случае использовать не получится.

Динамические массивы реализованных типов можно создавать аналогично:

```
Date *p = new Date[3];                  // вызываются конструкторы по умолчанию
for(auto i{0}; i < 3; ++i) p[i].print();        // смотрим
delete[] p;                             // вызываются деструкторы
```

Создание и уничтожение массивов реализованных типов имеет очень важные особенности:

1. При создании массива для каждого элемента массива вызывается конструктор по умолчанию.
2. При уничтожении массива вызывается для каждого элемента деструктор.

Это можно наблюдать при запуске программы (рис. 18.2).

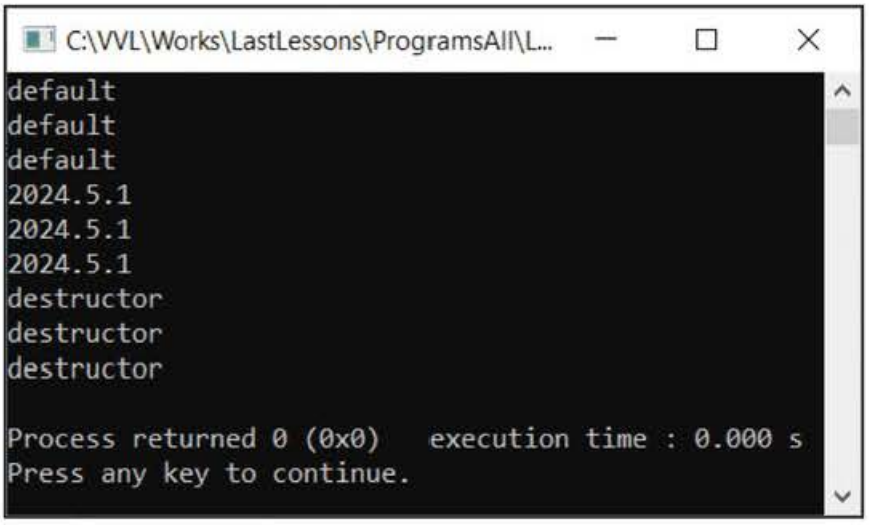


Рис. 18.2. Вывод при создании динамического массива

Мы видим в консоли:

- сначала три раза вызывается конструктор по умолчанию — для каждого элемента массива;

- конструктор в каждом элементе массива проинициализировал поля, значения которых в цикле выводятся в консоль;
- операция `delete[]` вызывает деструктор для каждого элемента массива.

Если программист забудет написать оператор `delete[]`, то деструктор не будет вызван и произойдет утечка памяти. Никогда не забывайте об этом!

Размеры массивов

Теперь выясним максимальные размеры массивов, размещаемых в разной памяти. Размеры зависят и от реализации компилятора, и от операционной системы. Статическую память распределяет компилятор при компиляции. Автоматическую память (стек) для локальных переменных программе выделяет операционная система при запуске. Операции для работы с динамической памятью, хотя и реализованы в языке C++, в конечном итоге обращаются к подсистеме динамической памяти операционной системы.

Напишем простую программу, с помощью которой выясним размеры разных видов массивов в ОС Windows и «Альт».

Основная идея состоит в том, что программа либо не будет компилироваться, либо аварийно завершится при превышении размера массива (листинг 18.2).

Листинг 18.2. Максимальные размеры массивов

```
#include <iostream>
using namespace std;
constexpr size_t sizeL =  0x001f'8fff;
constexpr size_t sizeG =  0x74ff'ffff;
constexpr size_t sizeD =  0x13'ffff'ffff;
    char global[sizeG];                    // глобальный массив
int main()
{   char local[sizeL];                     // локальный массив
    char *dynamic = new char[sizeD];       // динамический массив
    cout << sizeL << endl;
    cout << sizeG << endl;
    cout << sizeD << endl;
    return 0;
}
```

Тип массивов задан `char`, чтобы не заботиться о размере одного элемента. Размеры массивов представлены тремя шестнадцатеричными константами:

- `sizeL` — локальный;
- `sizeG` — глобальный;
- `sizeD` — динамический.

Выводим константы в десятичном формате. Для сравнения можно вывести конкретное число мегабайт или гигабайт, например:

```
cout << sizeL << endl;                    // размер локального массива
cout << 2ull * 32*32*32*32 << endl;       // 2 мегабайта
```

В системе Windows эти два оператора вывода покажут в консоли два числа:

```
2068479                                   // размер массива
2097152                                   // 2 мегабайта
```

Для локальных переменных выделяется не более 2 мегабайт (2^{20}) памяти.

Для глобальных массивов (и переменных) в совокупности выделяется не более 2 гигабайт (2^{30}) памяти, что показывает вывод двух значений:

```
1979711487                                // размер массива
2147483648                                // 2 гигабайта
```

И наконец, система динамической памяти в Windows позволяет выделять гигантский объем памяти для динамического массива:

```
85899345919                               // размер массива
85899345920                               // 80 гигабайт
```

Эксперименты с той же программой в операционной системе «Альт» выявили совершенно другие размеры:

```
constexpr size_t sizeL =  0x007f'dfff;
constexpr size_t sizeG =  0x0003'bfff'ffff;
constexpr size_t sizeD =  0x0003'bfff'ffff;
```

Для локальных переменных выделяется 8 мегабайт автоматической памяти:

```
8380415                                   // размер массива
8388608                                   // 8 мегабайт
```

Размеры глобального и динамического массива в операционной системе «Альт» одинаковы и составляют до 15 гигабайт:

```
16106127359                               // размер массива
16106127360                               // 15 гигабайт
```

Если превысить размеры динамического массива, программа завершается аварийно, т. к. генерируется исключение `bad_alloc`. Если исключение не обрабатывать, то выводится аварийное сообщение (рис. 18.3).

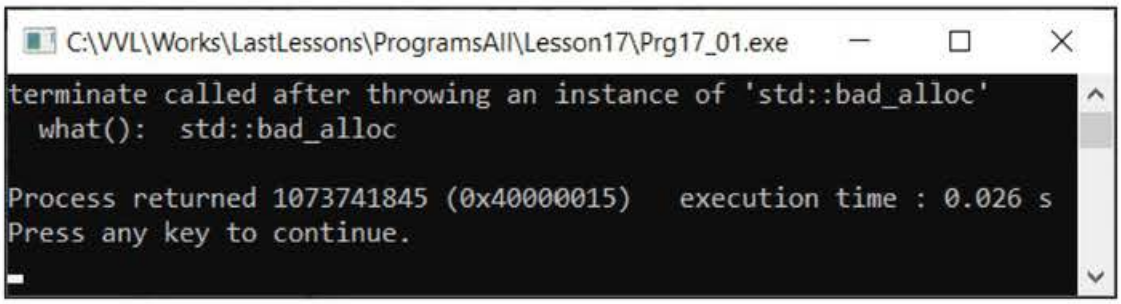


Рис. 18.3. Исключение при нехватке динамической памяти (Windows)

Единственное отличие такой ситуации в ОС «Альт» — другой код возврата при аварийном завершении. Сообщение о завершении будет тем же.

В 64-битной операционной системе такие ситуации бывают очень редко; в наших учебных программах — никогда, но знать об этом необходимо.

Стандартный динамический массив *vector*

Динамические массивы, при всей гибкости, имеют одно важное ограничение: созданный массив не может изменять свой размер. Нельзя просто добавить элемент в динамический массив. Допустим, у нас есть динамический массив размера N, и нужно добавить еще один элемент. Это большая работа:

- создать новый динамический массив размера N+1;
- скопировать N элементов исходного массива в новый;
- присвоить новое значение последнему элементу нового массива.

Вставка элемента в начало или в середину массива тоже требует выполнения аналогичной работы.

Подобные действия регулярно писались в реальных проектах. Поэтому в стандартной библиотеке был реализован динамический массив `vector`, в котором вся эта работа выполняется «под капотом», т. е. невидимо для программиста. Естественно, в библиотеке он называется контейнером, а мы в дальнейшем будем называть этот контейнер *вектором*. Для работы с этим контейнером нужно подключить библиотеку `<vector>`.

Объявление и инициализация вектора

Контейнер `vector` — это динамический массив, размер которого может быть задан переменной. И конечно, он обладает всеми возможностями, которыми обладает array-массив.

Сначала рассмотрим объявление векторов и их инициализацию:

```
vector <int> v(10);                    //1 - 10 нулей
vector <int> x(6, -1);                 //2 - шесть -1
```

Вектор `v` имеет размер 10 элементов типа `int`. Размер, как обычно, можно получить с помощью функции `size(v)` или в объектно-ориентированном стиле — `v.size()`. По умолчанию все элементы инициализируются нулями.

Вектор `x` имеет размер 6 элементов, и все элементы равны `-1`.

Можно инициализировать вектор универсальным способом, например:

```
vector <int> xx {6, -1};               //3 - два элемента 6, -1
```

Это объявление существенно отличается от объявления вектора `x`. В этом случае размер вектора `xx` равен количеству выражений в фигурных скобках, т. е. 2, и эти два элемента имеют значения `6` и `-1`.

> ***На заметку***
>
> При объявлении вектора нельзя одновременно задать его размер в круглых скобках и инициализацию в фигурных скобках.

Векторы можно инициализировать элементами любых массивов с помощью любого вида итераторов:

```
vector <int> t(x.begin(), x.end());      //4 - итераторы контейнера
int a[10] = { 1, 2, 3, 4, 5 };
vector <int> w (a, a + 10);              //5 - указатели
vector <int> ww(begin(a), end(a));       //6 - итераторы массива
vector <int> www(rbegin(a), rend(a));    //7 - обратные итераторы
array<int, 5> d {1,2,3,4,5};
vector <int> vd (cbegin(d), cbegin(d)); //8 - константные итераторы
```

Вектор `t` повторяет вектор `x`: размер равен 6 и все элементы равны `-1`. Векторы `w` и `ww` имеют размер 10 и все элементы равны элементам массива `a`:

```
1 2 3 4 5 0 0 0 0 0
```

Вектор `www` имеет тот же размер 10, но элементы в векторе расположены в обратном порядке:

```
0 0 0 0 0 5 4 3 2 1
```

Вектор `vd` по размеру и содержанию совпадает с array-массивом `d`.

И наконец, выясним, как объявляются векторы переменного размера и как можно менять размеры такого вектора.

```
vector <int> u;                          //9 - пустой вектор
```

Вектор `u` не содержит элементов. Это подтверждает вызов функции `size(u)`, который показывает ноль. Можно задать размер переменной, например:

```
int n;
cout << "Input n: "; cin >> n;
vector <int> vv(n,-1);                   //10 - n элементов -1
```

Такое объявление обычно используют при реализации олимпиадных задач.

Можно заполнять вектор всеми способами, с которыми мы познакомились ранее для массивов (см. *урок 12*).

Размеры векторов

Теперь выясним, сколько памяти занимают векторы, и зависит ли это от количества элементов. Выведем в консоль размеры всех объявленных выше векторов в том же порядке:

```
cout << sizeof(v) << endl;               //1 - 10 нулей
cout << sizeof(x) << endl;               //2 - шесть -1
cout << sizeof(xx) << endl;              //3 - два элемента 6,-1
cout << sizeof(t) << endl;
cout << sizeof(w) << endl;
cout << sizeof(ww) << endl;
cout << sizeof(www) << endl;
cout << sizeof(vd) << endl;
cout << sizeof(u) << endl;               //9 - пустой вектор
cout << sizeof(vv) << endl;              //10 - n элементов -1
```

Получим в консоли десять чисел 24. Следовательно, объем статической или автоматической памяти объекта типа vector не зависит от размера вектора — элементы размещаются в динамическом массиве, который выделяется при объявлении вектора.

Программисту предоставлены некоторые средства работы с этой памятью:

- метод capacity позволяет выяснить, сколько элементов можно разместить в памяти вектора на текущий момент;
- метод reserve резервирует память для большего количество элементов;
- метод shrink_to_fit возвращает всю память вектора системе.

Последний метод особенно полезен тогда, когда суммарный размер динамических переменных в программе приближается к предельному.

Операции с вектором

Векторы разрешается присваивать один другому. Принимающий вектор становится копией присваиваемого, т. е. получает и размер, и элементы присваиваемого вектора. Массивы вектору присваивать нельзя, но можно скопировать с помощью функции copy, используя итераторы.

Самое интересное — изменение размера вектора во время работы программы. Это можно делать разными способами. Можно воспользоваться методом assign:

```
u.assign(12,-2);
```

Вектор u (см. выше) получит размер 12 элементов, и все равны -2. При этом совершенно неважно, какого размера и содержания был вектор u до этого оператора (новый размер может быть как больше, так и меньше прежнего). То же самое делает метод resize.

Можно удалить все элементы вектора с помощью метода clear:

```
u.clear();
```

После этого функция size будет выдавать ноль (но выделенная вектору память не возвращается).

Но самое интересное, что можно добавлять в вектор элементы по одному:

```
for(auto i = 0u; i < 10; ++i) u.push_back(i);
```

Если размер вектора был 12, то после этого количество элементов увеличится на 10, и функция size выдаст размер 22.

Для удаления последних элементов используется метод pop_back. Вставка новых элементов в любое место делается методом insert, а удаление любой последовательности элементов или одного элемента (не последнего) можно выполнить методом erase.

Вектор можно присоединить к другому вектору:

```
vector<int> v1 = {1, 2, 3};
vector<int> v2 = {4, 5, 6};
```

```
copy(v1.begin(), v1.end(), back_inserter(v2));          // добавляет к v2
cout << size(v2) << endl;                               // выведет 6
copy(v2.begin(), v2.end(), ostream_iterator<int>(cout, ", "));
```

Последний оператор выведет:

```
4, 5, 6, 1, 2, 3,
```

Мы видим, что первый вектор добавился ко второму, изменив его размер.

Доступ к элементам вектора осуществляется по индексу или по итератору. Как обычно, для контейнеров STL к первому элементу можно обратиться с помощью метода `front`, а к последнему — методом `back`.

Так как вектор меняет свой размер, то можно ненароком «вылететь за границу». Для предотвращения таких аварий реализован специальный метод доступа по индексу `at`. Например, где-то в цикле написано выражение `v.at(k)`. Если значение индекса `k` равно или больше размера вектора `size(v)`, то возникает исключение `out_of_range`, которое можно обработать.

Снова длинные числа

Мы уже реализовали операцию сложения длинных чисел, используя строки (см. листинг 17.5). Реализуем умножение вектора на однозначное целое число.

Длинные числа `biginteger`, представленные в виде строки, начинаются со старшей цифры, поэтому в функции `add` использовались обратные итераторы. Для вектора мы можем реализовать обычный подход: младшая цифра — по младшему индексу. Нужно только преобразовать строку в вектор:

```
using number = vector<uint8_t>;                         // чтобы меньше писать
number v(size(s));                                      // вектор
transform(rbegin(s),rend(s), begin(v),                  // строка в вектор
          [](char ch){return (ch - '0');}               // символ в цифру
          );
```

Как видите, с помощью стандартного алгоритма и лямбды это делается буквально в одну строку кода.

Можно реализовать операцию умножения многократным сложением.

```
number operator*(const number &V, int k){
    number result{0};                   // для возврата нуля
    if(k == 0) return result;   // вернули нуль
    // если k не ноль
    result = V;                         // один раз умножили
    int i = --k;
    while(i-->0)                        // «домножаем» остальное
        result = result + V;
    return result;
}
```

Однако такое умножение очень неэффективно (подумайте, почему), поэтому реализуем умножение непосредственно (листинг 18.3).

Листинг 18.3. Умножение длинного числа на целый множитель

```
number operator*(const number &V, int k){
    number result{0};                          // для возврата нуля
    if(k == 0) return result;                  // вернули нуль
    uint8_t cf = 0;                            // перенос
    result = V;
    auto next_result = begin(result);
    while(next_result != end(result)){
        uint8_t ds = *next_result * k + cf;    // умножили цифру
        cf = ds / 10;                          // текущий перенос
        *next_result = (ds % 10);              // цифра результата
        ++next_result;                         // к следующей цифре
    }
    if(cf > 0) {                               // перенос из старшего разряда
        result.push_back(cf);
    }
    return result;
}
number operator*(int k, const number &V){ return (V * k); }
```

Функция получилась проще сложения, т. к. нужен только один итератор. Вторая функция требуется, чтобы можно было менять сомножители местами. В ней мы применили *делегирование*: функция делает свою работу, вызывая другую функцию. Эти две функции позволяют нам умножать длинное число на целое число и слева, и справа:

```
number L, R; Result;
Result = L * 9;                // первая функция
Result = 2 * R;                // вторая функция
```

Вывод длинного числа, представленного вектором, надо делать в обратном порядке, т. к. старшая цифра находится в конце.

```
for(auto t = rbegin(Result); t != rend(Result); ++t) cout << int(*t);
```

Но можно написать перевод вектора в строку, «перевернув» число.

Многомерные массивы

Существует множество задач, в которых нужно обрабатывать таблицы данных. Таблица — это *двумерный* массив данных. Каждый элемент таблицы размещается в какой-то строке на некотором месте в этой строке. Говорят, что элемент находится

в такой-то строке и в таком-то столбце таблицы. Поэтому при объявлении двумерного массива требуется два индекса: индекс строки и индекс столбца. Такие массивы называются либо таблицей, либо матрицей в зависимости от задачи.

Массивы, элементы которых имеют более одного индекса, называются *многомерными* массивами. *Многомерный* массив в C++ рассматривается как одномерный, элементами которого являются массивы. Понятно, что элемент многомерного массива сам может быть многомерным массивом.

«Многомерность» массива называется его *размерностью* и определяет количество индексов. Таким образом, если массив двумерный, то индексов — два, а если трехмерный, то три и т. д. Один индекс называется *измерением*. Поэтому размерность — это количество измерений.

Обращаю ваше внимание на то, что размер и размерность — это совершенно разные понятия. Размер — это количество элементов, а размерность — это количество измерений (индексов). По каждому измерению может быть разный размер — разное количество элементов.

Объявим двумерные массивы:

```
int A[3][4];                                    // двумерный массив
array<array<int,4>,3> B;                        // двумерный array-массив
vector<vector<int>>D(3, vector<int>(4));        // двумерный вектор
```

Только встроенный массив выглядит как двумерный массив (индексов в объявлении два). Массивы `array` и вектор демонстрируют одномерность: `array` — это массив массивов, а вектор — это вектор векторов.

Запись двумерного вектора выглядит не очень красиво. Можно улучшить читабельность, применив using-объявление:

```
using line = vector<int>;
vector<line>D(3, line(4));
```

Можно объявить и вектор массивов, и массив векторов:

```
vector<array<int,4>> M(3);      // 3 массива по 4 элемента int
array<vector<int>,3> P;         // 3 вектора, размер может меняться
```

Все эти объявления представляют собой массив из трех элементов, в чем можно убедиться с помощью функции `size`:

```
cout << size(A) << endl;
…
cout << size(P) << endl;
```

Все операторы выдают одно и то же число `3`.

Элементами каждого массива (кроме массива `P`) являются массивы из четырех элементов типа `int`. Массив `P` имеет три элемента переменного размера: можно добавлять и удалять элементы векторов, например:

```
P[0].push_back(42);
```

Как и другие переменные, многомерный массив можно инициализировать:

```
int A[3][4]{{1,2,3,4},{2,3,4,5},{3,4,5,6}};
```

Элемент A[0] получил значение {1,2,3,4}, A[1] равен {2,3,4,5}, A[2] равен {3,4,5,6}. Задать количество значений во внутренних скобках больше или меньше четырех нельзя — компилятор «ругается». Но без вложенных скобок можно инициализировать, например:

```
int A[3][4]{14,2,3,5,8,6};
```

Тогда A[0] равен {14,2,3,5}, A[1] равен {8,6,0,0}, A[2] — {0,0,0,0}.

Массив array тоже можно инициализировать, но только вторым способом:

```
array<array<int,4>,3> B{1,2,3,4,2,3,4};
```

Здесь элемент B[0] получит значение {1,2,3,4}, элемент B[1] станет равным {2,3,4,0}, а элемент B[2] получит нули во всех элементах.

Вектор, у которого заданы размеры, инициализировать нельзя. А вот вектор без указания размеров разрешается инициализировать так же, как и массив:

```
vector<vector<int>>DD { {1,2,3,4}, {2,3,4,5}, {3,4,5,6} };
```

Естественно, размеры этого вектора будут те же, что и у массива A.

Мы воспринимаем двумерный массив как таблицу из строк и столбцов. Элементы массива проиндексированы следующим образом:

```
Столбцы    0       1       2       3
         [0][0]  [0][1]  [0][2]  [0][3]      строка 0
         [1][0]  [1][1]  [1][2]  [1][3]      строка 1
         [2][0]  [2][1]  [2][2]  [2][3]      строка 2
```

Доступ к элементу двумерного массива записывается как A[i][j], где первый индекс — это номер строки, а второй — номер столбца.

Однако память представляет собой одномерный массив, поэтому элементы двумерного массива располагаются в памяти последовательно: A[0], A[1], A[2]. Это значит, что двумерный массив расположен в памяти по строкам. На рис. 18.4 показано, как размещается массив A.



Рис. 18.4. Размещение в памяти массива A[3][4]

В таком же порядке располагаются в памяти элементы массива array. Элементы вектора векторов таким же образом размещаются в динамической памяти.

При инициализации одномерных встроенных массивов (см. *урок 9*) можно было не указывать количество элементов. Для встроенных двумерных массивов тоже можно не задавать размер, но только в самых левых скобках:

```
int A[][3] = { { 1, 2, 3 }, { 5, 7 } };
```

Массив A состоит из двух элементов, каждый из которых — массив из трех целых. Кстати, массив си-строк представляет собой двумерный массив символов. Вспом-

ним задачу вычисления дня недели (см. листинг 9.2). В коде был объявлен массив строк `Days`. Объявление этого массива для си-строк выглядит так:

```
char Days[][12] = { "Воскресенье", "Понедельник", "Вторник",
                    "Среда", "Четверг", "Пятница", "Суббота" };
```

Массив стал двумерным, причем размер элемента (это си-строка) должен задаваться явно — таковы правила в языке C. Это не очень удобно, и это одна из причин включения реализации строк в стандартную библиотеку.

Для array-массивов размеры должны указываться всегда.

Для вектора можно вообще не указывать количество элементов. А вектор векторов может быть объявлен с одним размером вместо двух:

```
vector<vector<int>> D(3);                          // двумерный вектор
```

В этом случае задается количество элементов, тип которых `vector<int>`, а размер одного элемента не указывается. При using-объявлении, как это показано выше, объявление вектора выглядит просто как одномерный массив:

```
vector<line> D(3);                                 // двумерный вектор
```

Понятно, что здесь объявлен вектор из трех элементов типа `line`.

Динамические многомерные массивы

Массив динамических объектов — это массив указателей:

```
int *pi[] = { new int(5), new int{2} };
```

Здесь объявлен массив из двух указателей на объекты типа `int`, причем оба объекта инициализированы. Обращаться к элементу такого массива нужно с операцией разыменования:

```
cout << *pi[0] << endl;
```

Таким же образом можно создать и массив указателей на динамические массивы. Причем размер этих массивов можно задавать переменной, их можно даже проинициализировать:

```
int n = 5;                                         // размер - переменная
int *pm[] = {new int[n]{10,11,12,13,14}, new int[2]() };
```

Если количество инициализаторов будет больше размера массива, то никаких сообщений компилятора мы не получим. Но при выполнении возникнет исключение `std::bad_array_new_length`, и программа закончится аварийно.

Обращение к элементам таких массивов осуществляется без разыменования:

```
for(auto i{0}; i < n; ++i) cout << pm[0][i] << ' ';
```

И, наконец, можно создать динамический массив динамических массивов. В этом случае мы сможем задавать любые размеры с помощью переменных.

Один динамический массив создается с помощью указателя. Для создания динамического массива динамических массивов требуется указатель на динамический массив указателей, т. е. *указатель на указатель*.

Например, во многих математических задачах часто фигурируют матрицы — прямоугольные или квадратные таблицы вещественных чисел. Обычно размер матрицы пишут так: n × m — n строк по m элементов в строке.

Создадим такую матрицу динамически. Пусть размеры матрицы заданы в переменных `n` и `m`. Тогда создание матрицы `M` делается так:

```
float **M = new float *[n];            // динамический массив указателей
for(int i = 0; i < m; ++i)
    M[i] = new float[m];               // динамические массивы элементов
```

Обращение к элементам матрицы `M` делается обычным образом: `M[i][j]`.

Удаление массивов тоже делается обычным образом, только надо соблюдать порядок: сначала в цикле удаляются динамические массивы элементов, а потом удаляется динамический массив указателей:

```
for(i = 0; i < n; i++)
    delete[] M[i];              // удаление массивов элементов
delete []M;                     // удаление массива указателей
```

Если требуется написать функцию с параметром — двумерным динамическим массивом, то нужно передавать указатель на указатель:

```
void PrintMatrix (int **m, int n)              // квадратная матрица n*n
```

Можно в качестве параметра написать массив указателей:

```
void PrintMatrix (int *m[], int n)             // квадратная матрица n*n
```

Конечно, если матрица не квадратная, то требуется передавать два размера. Можно еще поисследовать варианты передачи такого параметра по ссылке, но мы не будем этого делать, т. к. вместо динамических массивов сейчас практически повсеместно используются векторы.

Задачи и упражнения

1. Написать функцию преобразования длинного числа типа `number` в число типа `biginteger`.
2. Реализовать все задания *урока 12* с вектором.
3. Напишите функцию вывода двумерного массива в виде таблицы.
4. Напишите функцию заполнения двумерного массива змейкой и выведите в консоль. Пример:

```
 1  2  3  4  5
10  9  8  7  6
11 12 13 14 15
20 19 18 17 16
```

5. Напишите функцию заполнения массива по спирали. Пример:

```
1 2 3
8 9 4
7 6 5
```

- Урок 19 -

Учимся писать и читать

До сих пор мы использовали в программах самые простые средства ввода и вывода данных — стандартные потоки (см. *урок 2* и *3*), и познакомились с простыми способами форматирования вывода с помощью манипуляторов. Но в стандартной библиотеке реализовано гораздо больше средств ввода/вывода.

Основные концепции ввода/вывода C++ унаследовал от языка C. Во-первых, средства ввода/вывода отделили от языка и вынесли в отдельную библиотеку. Во-вторых, удалось реализовать независимость ввода/вывода от конкретных устройств. Программа не имеет дела ни с устройствами, ни с файлами — она работает с потоками.

Поток — это последовательность символов. Ввод данных осуществляется из *входного* потока, вывод производится в *выходной* поток. Ну, а уже поток можно связать и с устройством, и с файлом. Таким образом, можно работать с устройствами и файлами единообразно. Чтобы писать ввод/вывод на C++, программисту требуется минимум знаний о файловой системе. Практически все, что нужно знать (по крайней мере, на первых порах), — как пишутся имена файлов в программе.

Потоковый ввод/вывод стал популярным, и сейчас реализован в стандартных библиотеках всех языков программирования.

Потоки

В библиотеке STL реализованы три вида потоков: стандартные, *файловые* и *строковые* потоки. Стандартные потоки бывают только *однонаправленными*: либо входные (данные только читаются из потока), либо выходные (данные только пишутся в поток).

Как мы знаем, стандартные потоки имеют имена, определенные в стандарте, которые нельзя использовать в другом смысле. Эти имена по умолчанию «привязаны» к стандартным устройствам: клавиатуре и экрану. Однако есть возможности перенаправить стандартные потоки на другие устройства.

Файловые потоки могут быть как *одно-*, так и *двунаправленными*: из потока можно вводить данные и в тот же поток — выводить. И это естественно, т. к. файловые потоки обычно *связываются* с файлами на диске.

Строковые потоки позволяют работать и с си-строками, и со стандартными строками типа `string` в памяти программы. Эти потоки также могут быть двунаправленными.

Стандартные и строковые потоки являются *форматируемыми*. Если поток форматируемый, то это означает, что в операциях ввода/вывода выполняется преобразование данных. Так как поток — это последовательность символов, то при вводе выполняется преобразование данных из символьного вида в двоичное представление, а при выводе — наоборот.

Файловые потоки могут быть как форматируемыми, так и нет — это можно задать в коде при объявлении потока. Мы будем работать с форматируемыми файловыми потоками.

> ***НА ЗАМЕТКУ***
>
> Потоки еще делятся по виду символов на широкие (wide) и узкие (narrow). Мы работаем только с узкими потоками символов типа char.

Стандартные потоки

Вывод в стандартный поток `cout` нам уже хорошо знаком. Единственное, о чем не надо забывать, — о приоритетах операций, если нужно выводить выражение. Операция вывода `operator<<` изначально является операцией сдвига влево и имеет свой приоритет. При перегрузке ее приоритет остался таким же. Поэтому при выводе выражений порядок выполнения операций может оказаться неожиданным, например:

```
cout << (d < f) ?d :f << '\n';
```

Работает этот оператор совершенно не так, как ожидается: вместо значения `d` или `f` выводится `0` или `1` в зависимости от выполнения неравенства `(d < f)`. Поэтому возьмите себе за правило при выводе выражений всегда писать скобки:

```
cout << ((d < f) ?d :f) << '\n';
```

В этом случае оператор работает именно так, как задумывалось.

Ввод встроенных типов

Ввод из стандартного потока в простых вариантах мы тоже уже неоднократно использовали. Рассмотрим некоторые подробности. Ввод осуществляется из входного потока `cin` в переменные программы:

```
int i; cin >> i;
```

Ввод завершается нажатием клавиши <Enter>. Оператор ввода осуществляет разбор и преобразование последовательности вводимых символов в двоичное представление в соответствии с типом приемника. На клавиатуре мы можем набирать любые символы, однако ввод в переменную прекращается при наборе недопустимого для данного типа символа.

Например, наберем на клавиатуре последовательность символов:

```
123.567
```

Тогда в переменную `i` попадет значение `123`, т. к. точка — недопустимый символ для целого числа. Однако в связи с тем, что ввод буферизуется, символы `.567` не пропадают и могут попасть при следующем вводе в другую переменную. Например, мы собираемся ввести значения двух переменных:

```
int ii; double dd;
cin >> ii; cin >> dd;
```

Набираем на клавиатуре число `123.567`. В переменную `ii` попадет значение `123`, а в переменную `dd` — значение `0.567`.

Набирать на клавиатуре несколько значений можно любым из двух способов:

- каждое число заканчивать нажатием клавиши <Enter>;
- после каждого числа набирать пробел, а последнее число закончить нажатием клавиши <Enter>.

Пробелов между числами может быть несколько. Пробел и табуляция — это символы-разделители. При вводе чисел ведущие символы-разделители пропускаются, а первый же пробел после допустимых символов для данного типа переменной является ограничителем значения.

Булевы значения тоже можно вводить с клавиатуры. Если в операторе ввода не написан манипулятор `boolalpha`, то нужно вводить `1` (`true`) или `0` (`false`). Манипулятор можно написать:

```
bool t;
cin >> boolalpha >> t;
```

Тогда можно вводить с клавиатуры слова `true` или `false`. Любые другие символы воспринимаются как недопустимые для данного типа.

Символы вводятся так же, как и значения других элементарных типов:

```
char c1, c2, c3;
cin >> c1 >> c2 >> c3;
```

Ввод символа (даже одного) должен заканчиваться клавишей <Enter>. При вводе символов операцией `operator>>` пробелы также пропускаются, поэтому можно задавать последовательность символов разными способам. Так:

```
йцу<enter>
```

или так:

```
й<пробел>ц<пробел>у<enter>
```

Пробелов может быть сколько угодно. Чтобы пробелы не пропускались, можно написать манипулятор `noskipws`:

```
cin >> noskipws >> c1 >> c2 >> c3;
```

Если мы наберем на клавиатуре символы:

```
й<пробел>ц<пробел>у<enter>
```

в `c1` попадет «й», в `c2` — пробел, а в `c3` — «ц».

Ввести в символьную переменную любой символ можно с помощью метода `get`. Ввод символа делается так:

```
ch = cin.get();          // вариант 1 (завершить ввод нажатием <Enter>)
cin.get( ch );           // вариант 2 (завершить ввод нажатием <Enter>)
```

Символьные переменные являются интегральным типом и могут участвовать в арифметических операциях как целые. И мы можем без проблем вывести переменную типа `char` в виде целого числа. Однако ввести целое значение в символьную переменную нельзя. Даже если набирать на клавиатуре цифры, в символьную переменную всегда попадает код цифровой клавиши.

Ввод строк с пробелами

Как мы знаем (см. *урок 3*), ввод строк с помощью операции `operator>>` выполняется до пробела. Для ввода си-строк с пробелами в символьный массив нужно использовать либо метод `get()`, либо метод `getline()`.

Метод `get()` позволяет ввести в символьный массив из n элементов не более $n-1$ символов, следит за символом завершения ввода и проставляет после введенных символов нуль-байт. Метод перегружен и имеет прототипы:

```
istream& get(char *str, streamsize count);
istream& get(char *str, streamsize count, char delim);
```

Для ввода строки с пробелами в массив `str` нужно написать оператор:

```
char str[50];
cin.get(str, 50);
```

Завершается ввод, как обычно, нажатием клавиши <Enter>.

После этого во входной поток попадает символ `'\n'`, а в массив символов на последнее место записывается нуль-байт. Чтобы можно было корректно продолжить ввод (следующей строки), символ `'\n'` надо удалить из входного потока. Это делается методом `ignore()`, который извлекает символы из потока, ничего не занося в переменные. Таким образом, ввод одной строки в массив `str` выполняется так:

```
cin.get(str,50); cin.ignore();
```

Символ `'\n'` является символом завершения ввода строки по умолчанию, однако мы можем задать любой символ завершения, например точку (`'.'`):

```
cin.get(s,50,'.'); cin.ignore();
```

Правда, для окончания ввода все равно придется нажать клавишу <Enter>.

> ***На заметку***
>
> Символ завершения остается в буфере и после обычного ввода операцией >>.

Метод `getline()` работает при вводе строк аналогично методу `get()`, но еще и удаляет из входного потока тот самый символ-завершитель строки (символ `'\n'` по умолчанию), так что `ignore()` вызывать нет необходимости.

Прототипы метода следующие:

```
istream& getline(char* str,  streamsize count);
istream& getline(char* str,  streamsize count, char delim);
```

Наши примеры при использовании метода `getline()` выглядят так:

```
cin.getline(str,50);
cin.getline(str,50,'.');
```

Кстати, посчитать количество реально введенных символов позволяет такой хороший метод, как `gcount()`:

```
const int lineSize = 250;
char Buf[lineSize];
cin.getline(Buf, lineSize);     // ввод до \n или не более 250 символов
int readSymbol = cin.gcount();  // сколько символов прочитано
```

Методы входного потока `get()` и `getline()` не предназначены для ввода стандартных строк типа `string`, т. к. их первый параметр — указатель на символ. В библиотеке `<string>` имеется собственная функция `getline()`, которая позволяет вводить строки с пробелами. Вызывать функцию надо так:

```
string s;
getline(cin, s);        // ввод из стандартного потока
```

В качестве первого параметра эта функция принимает поток. Очевидно, это сделано для большей общности, т. к. на месте первого параметра может стоять не только стандартный поток `cin`, а, например, файловый поток.

Эта функция (вернее, шаблон функции) перегружена и может иметь третий параметр, представляющий собой ограничитель строки — аналогично методу `getline()` из библиотеки `<iostream>`. Поэтому вводить строку можно и так:

```
string s;
getline(cin, s,'.');    // -- ввод из стандартного потока
```

Точка является завершающим символом строки (однако нажимать клавишу <Enter> все равно нужно), она не включается в строку, и в переменную `s` попадут символы до точки. Например, если мы наберем на клавиатуре строку «привет от строки. продолжение строки.», в переменную `s` попадут символы до первой точки «привет от строки».

Как видите, поведение этой функции совершенно аналогично поведению одноименного метода из библиотеки `<iostream>`.

Пишем новый калькулятор

Вспомним программу-калькулятор *урока 8* (см. листинг 8.8). Выражения для вычисления нужно было задавать без пробелов в таком виде:

```
1.1+2.67
2*3.4
78/3
4-1.5
```

Выражения с отрицательными числами тоже задавались без пробелов:

```
-1.1+2.67
2*-3.4
-78/-3
```

Эти выражения вычислялись калькулятором правильно, однако выглядят они не очень хорошо. Лучше такие выражения задавать с пробелами:

```
-1.1 + 2.67
2 * -3.4
-78 / -3
```

Чтобы программа смогла обрабатывать выражения с пробелами, нам нужно реализовать три шага:

1. Ввести строку с пробелами.
2. Удалить пробелы в строке.
3. Получить из строки значения переменных `a`, `op` и `b`.

Эти три действия заменяют оператор ввода переменных `a`, `op` и `b`.

Первое действие мы уже делать умеем:

```
string expr;                          // строка для ввода
getline(cin, expr);                   // ввод строки с пробелами
```

Удаление пробелов из строки легко выполнить в цикле, но мы используем стандартные средства. Для строк реализован метод `erase` (стирать), которому надо указать, с какого места строки начинать и где заканчивать удаление. Эти аргументы являются итераторами. В C++17 удалять надо так:

```
expr.erase(remove(expr.begin(),expr.end(),' '), expr.end());
```

На месте первого аргумента задан вызов стандартного алгоритма `remove`, который выполняет перемещение всех пробелов в конец строки и возвращает итератор на первый пробел. Таким образом, метод `erase` удаляет все пробелы, перемещенные в конец строки.

Если мы работаем в стандарте C++20, то все делается гораздо проще:

```
erase(expr,' ');
```

И наконец, нужно получить значения переменных `a`, `op` и `b` из строки `expr`. В этом нам помогут строковые потоки.

Строковые потоки

Строковые потоки — это универсальные средства для перевода внутренних двоичных значений в строки (выходной строковый поток) и обратно (входной строковый поток). Таким образом, эти потоки по определению форматируемые. Чтобы можно было использовать строковые потоки, нужно подключить библиотеку `<sstream>`. После этого в программе можно объявлять строковые потоки трех видов:

- входной `istringstream`;
- выходной `ostringstream`;
- двунаправленный `stringstream`.

Нам для получения значения переменных `a`, `op` и `b` из строки `expr` требуется входной поток (мы вводим данные). Это делается так:

```
istringstream ssin{expr};
ssin >> a >> op >> b;
```

Обработанная строка `expr` без пробелов помещается в строковый поток `ssin`, а потом из строкового потока вводятся значения переменных. Как видите, такой ввод отличается от ввода из стандартного входного потока только именем потока.

Теперь соберем все вместе и отрефакторим калькулятор (листинг 19.1).

Листинг 19.1. Новый калькулятор

```
#include <iostream>
#include <sstream>                      // строковые потоки
#include <string>
#include <algorithm>                    // remove
using namespace std;
int main(){
   double a, b; char op {'?'};
   cout << "Задайте выражение: ";
   string expr;                         // входная строка-выражение
   getline(cin, expr);                  // ввод выражения с пробелами
   expr.erase(remove(expr.begin(),expr.end(),' '), expr.end());
   istringstream ssin{expr};            // входной строковый поток
   ssin >> a >> op >> b;                // получение значений переменных
   switch (op)
   { case '+':  cout << a+b; break;
     case '-':  cout << a-b; break;
     case '*':  cout << a*b; break;
     case '/':  if (b!=0) cout << a/b;
                else cout << "Делитель равен нулю!\n";
                break;
     default: cout << "Неверный знак операции!" << endl;
   }
   return 0;
}
```

Запустим программу и зададим выражение с пробелами (рис. 19.1).

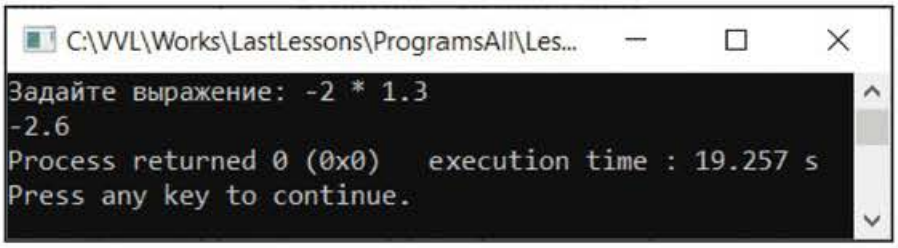


Рис. 19.1. Работа нового калькулятора (выражения с пробелами)

Как видим, все работает правильно.

Но хотелось бы сделать вывод более понятным, например:

```
-2 * 1.3 = -2.6
```

Это можно сделать с помощью выходного строкового потока. Сначала введенная строка с пробелами сохраняется в строковой переменной:

```
string work = expr;
```

После вычисления результата в программе надо добавить три строки кода:

```
ostringstream ssout;                    // выходной строковый поток
ssout << work << " = " << res;  // форматирование выходной строки
cout << ssout.str() << '\n';            // вывод строки в консоль
```

Результат работы этого кода показан на рис. 19.2.

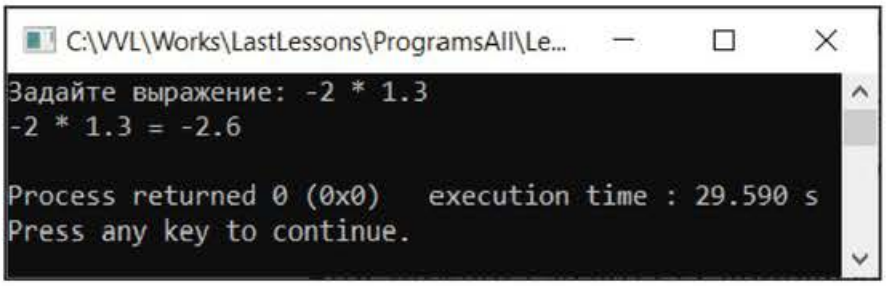


Рис. 19.2. Красивый вывод вычисления выражения

Файлы

Файл — это поименованная порция информации, размещенная на внешнем носителе. В настоящее время файлы размещаются на магнитных дисках. Для управления множеством файлов в состав операционной системы входит *файловая система*. Именно файловая система определяет, как именуются файлы и где они размещаются. В разных операционных системах — разные файловые системы и разные правила именования файлов.

> ***На заметку***
>
> В Windows полное имя файла должно содержать имя диска, а в Linux имя диска не указывается.

В программе на C++ имя файла обычно задается либо строковым литералом, либо строковой переменной. В старом стандарте разрешалось задавать имя файла только си-строкой. В C++11 устранили этот недостаток, и теперь допускается имя файла задавать стандартной строкой типа `string`.

Имя файла состоит из собственно имени и расширения, которое может и отсутствовать. Количество символов в расширении не регламентируется, но по традиции обычно задают от 1 до 3 символов. Если расширение есть, оно отделяется от имени точкой.

В настоящее время в именах файлов разрешается использовать кириллицу. Но возьмите себе за правило писать имена только в английском алфавите, чтобы не возникало неожиданных проблем.

Правила работы с файлами были разработаны в «доисторические» времена, однако сохранились до настоящего времени практически в неизменном виде:

1. Файл открывается.
2. Выполняются операции обмена между программой и файлом.
3. Файл закрывается.

Файл может быть открыт для чтения и (или) записи. Если файл открыт для чтения, то данные читаются из файла в переменные программы, и файл является *входным*. При чтении данных из входного файла программа должна определить, когда данные закончились. Если файл открыт для записи, то данные пишутся в файл из переменных, и файл называется *выходным*.

Файлы и потоки

Программа на C++ имеет дело не с файлом, а с потоком. Поток — это обычная переменная, которая имеет область видимости и время жизни в соответствии с объявлением, и не имеет никакого отношения к файлам на диске. Поэтому, чтобы работать с файлом в программе, нужно как-то связать потоковую переменную с этим файлом. Обычно эта связь задается либо при открытии файла, либо при создании потока.

При закрытии файла связь разрывается. Подчеркиваю, что разрывается только связь потоковой переменной с файлом, сама переменная продолжает «жить» в соответствии с объявлением, и ее снова можно связать с тем же или с другим файлом.

Обычно различают *текстовые* и *двоичные* файлы. Мы рассмотрим только текстовые файлы, т. к. они используются в олимпиадных задачах и ЕГЭ.

Текстовые файлы состоят из строк, которые завершаются символом конца строки. В программе на C++ этот символ обозначается как `'\n'`.

> ***НА ЗАМЕТКУ***
>
> В Windows и Linux конец строки в текстовых файлах устроен по-разному. Поэтому при переносе текстового файла из одной системы в другую требуется перекодировка.

В операциях обмена с текстовым файлом выполняется преобразование данных аналогично тому, как это происходит для стандартных и строковых потоков. По нашей классификации потоки, связанные с текстовыми файлами, являются форматируемыми.

Чтобы можно было использовать потоки для работы с файлами, нужно подключить библиотеку `<fstream>`. Это позволит нам работать и с входными, и с выходными файлами.

Читаем и пишем

Напишем программу с выводом случайных чисел в файл (листинг 19.2). Датчик инициализируем нашей функцией `time` (см. листинг 8.4). Выведенный файл потом прочитаем и выведем в консоль.

Листинг 19.2. Вывод чисел в файл

```
int main(){ mt19937_64 gen { time() };  // инициализация датчика
    {  ofstream strm("oonumber.txt");   // си-строковый литерал
       for(int i = 0; i < 10; i++)      // выводим 10 чисел
          strm << (gen() % 20) << '\n'; // выводим в столбик
    }                                          // деструктор закроет файл
    //  читаем записанный файл как входной
    {  ifstream strm("oonumber.txt"s);         // string литерал
       string ss{};                            // читаемая строка файла
       getline(strm, ss);
       while(!strm.eof())               // проверяем признак конца файла
       { cout << ss << ' ';             // выводим на экран
         getline(strm, ss);             // читаем числа как строки
       }
    }                                   // деструктор закрывает файл
 return 0;
}
```

Обратите внимание на внутренние блоки в теле функции `main()`. В блоках объявлены локальные объекты-потоки `strm`: в первом — выходной (output file stream, **`of`**`stream`), во втором — входной (input file stream, **`if`**`stream`). Больше ничего не требуется, т. к. объявление объекта и открывает файл, и связывает поток с файлом. При выходе из блока локальная переменная уничтожается, деструктор закрывает файл и разрывает связь.

В первом блоке файл с именем `"oonumber.txt"` создается (имя указано символьным си-литералом). В файл пишется 10 случайных чисел в «столбик» (после каждого числа выводится символ `'\n'`):

```
strm << (gen() % 20) << '\n';   // выводим в столбик
```

Как видим, вывод в файловый поток фактически не отличается от вывода в стандартный поток. При выводе выполняется преобразование двоичного числа в строку символов, и эта строка записывается в файл.

При выходе из блока переменная `strm` уничтожается (вызывается деструктор) и файл закрывается. В следующем блоке объявляется другая локальная переменная с тем же именем, но тип — входной поток (имя файла задается строковым литералом). Строки файла читаются в цикле без преобразования и выводятся в консоль. В условии цикла вызывается специальный метод потока `eof` (end of file), который проверяет, все ли данные из файла прочитаны.

Напишем программу записи-чтения файлов в более традиционном виде — явно открывая и закрывая файл. Выходной файл запишем в виде, который принят для олимпиадных задач: в первой строке файла задается количество данных `n`, в следующей строке файла записываются числа через пробел.

Читать данные будем в вектор, создавая префиксные суммы (см. *урок 12*). Напомню, что префиксная сумма k-го элемента массива — это сумма всех элементов от нулевого до k-го. Код программы показан в листинге 19.3.

Листинг 19.3. Получение префиксных сумм из файла

```
int main(){
  mt19937_64 gen { time()};              // инициализация датчика
  int N {};                              // количество чисел
  ofstream strm;                         // выходной поток
  strm.open("soonumber.txt");            // открываем выходной
  if (strm.is_open())                    // если создался
  { N = 10;                              // размер вектора
    strm << N << '\n';                   // выводим размер
    for(int i = 0; i < N; i++)           // выводим N чисел
      strm << (gen() % 20) << ' ';       // выводим в строку
    strm.close();                        // закрываем выходной поток
                                         // суммирование чисел, записанных в файле
    ifstream strm("soonumber.txt");      // открываем тот же файл
    if (strm)                            // если открылся
    { strm >> N;                         // читаем количество
      vector<int> prefix(N);             // создаем вектор префиксных сумм
      int number, summa = 0, i = 0;
      while(strm >> number)              // ввод числа
      { summa += number;                 // суммирование
        prefix[i] = summa;               // положили сумму в вектор
        ++i;
      }
      strm.close();                      // закрываем поток-файл
      copy(begin(prefix),end(prefix),                  // выводим вектор
           ostream_iterator<int>(cout, "\n"));         // в столбик
    }
  }
  return 0;
}
```

В этой программе выходной поток `strm` объявлен в теле (блоке) главной функции, а входной поток с тем же именем — во вложенном блоке оператора `if`. Конфликта не возникает, т. к. второе имя скрывает первое (см. *урок 7*).

Еще нужно обратить внимание на условие цикла, в котором вводится очередное число. Здесь, как обычно, происходит преобразование результата «вычисления» выражения (`strm >> number`) в булевский тип. Пока поток в нормальном рабочем состоянии, результат преобразования дает `true`, и цикл продолжается. Если при очередном чтении возникнет какая-нибудь ошибка или числа закончатся (поток окажется в состоянии end-of-file), то результатом преобразования станет `false`, и цикл завершится.

Кстати, если требуется вывести заполненный массив (вектор) в файл, то это можно сделать в две строки кода. Вывод делается с помощью итераторов вывода.

Например, выведем вектор prefix:

```
ofstream ostrm("prefix.txt");              // объявили файл (поток)
copy(begin(prefix), end(prefix), ostream_iterator<int>(ostrm, "\n"));
```

Такой вывод отличается от вывода в консоль только именем потока. В данном случае числа выводятся «в столбик», т. к. разделитель — "\n".

Рассмотрим еще пример ввода из файла в двумерный массив. Предположим, в некоторой олимпиадной задаче требуется ввести двумерный массив нулей и единиц. В первой строке файла записаны числа n и m, которые задают количество строк и количество элементов в каждой строке. Следующие n строк содержат по m нулей и единиц. Например, в файле могут быть данные:

```
3 4
0 1 1 1
1 0 0 1
1 0 1 0
```

Это массив из 3 строк по 4 элемента.

Пусть такой массив хранится в файле с именем 2D.txt. Ввод делается так:

```
using row = vector<int>;
int n{}, m{};                              // размеры массива
ifstream strm("2D.txt");                   // открываем для чтения
strm >> n >> m;                            // читаем размеры
row line(m);                               // вектор для строки массива
vector<row> matrix(n);                     // двумерный массив
int i = 0;                                 // индекс по строкам массива
while (i < n){
   int j = 0;                              // индекс по строке массива
   while(j < m) { strm >> line[j]; ++j; }      // ввод строки массива
   matrix[i] = line;
   ++i;
}
strm.close();                               // закрываем поток-файл
```

Для проверки правильности ввода можно написать следующий код:

```
i = 0;
while(i < n){
   row t{matrix[i]};
   copy(begin(t), end(t), ostream_iterator<int>(cout," "));
   cout << "\n";                           // завершение вывода строки массива
   ++i;
}
```

Как видите, ничего сложного в работе с файлами нет.

ЕГЭ и файлы

В ЕГЭ по информатике файлы используются во многих задачах. Например, в демонстрационном варианте КИМ 2025 года в девяти задачах предоставляются дополнительные наборы данных в файлах.

Рассмотрим задачу 17, которая сформулирована так:

> В файле содержится последовательность натуральных чисел. Ее элементы могут принимать целые значения от 1 до 100 000 включительно. Определите количество пар последовательности, в которых остаток от деления хотя бы одного из элементов на 16 равен минимальному элементу последовательности. В ответе запишите количество найденных пар, затем максимальную из сумм элементов таких пар. В данной задаче под парой подразумевается два идущих подряд элемента последовательности.

Файл данных для этой задачи имеет имя demo_2025_17.txt, содержит 10 000 чисел «в столбик», причем в файле количество чисел не указано.

Давайте решим эту задачу. Она четко разбивается на две части. Сначала нужно прочитать файл в вектор и в процессе чтения определить минимум последовательности. Во второй части надо найти уже в векторе количество и максимум сумм пар чисел, удовлетворяющих условию задачи (листинг 19.4).

Листинг 19.4. Задача 17. Демонстрационный вариант КИМ ЕГЭ 2025

```
int main(){
  int number {0};                              // вводимое число
  vector<int> V;                               // количество чисел неизвестно
  ifstream strm("demo_2025_17.txt");           // входной файл
  int mini {100'000};                          // минимум последовательности
  while(strm >> number)                        // читаем число из файла
  { mini = min(number, mini);                  // выбираем минимум
    V.push_back(number);                       // добавляем в вектор
  }
  strm.close();                                // закрыли поток (файл)
  int count{0}, int maxi{0};                   // количество и максимум
  for(auto i{0u}; i < size(V)-1; ++i){
    if((V[i]%16 == mini)||(V[i+1]%16 == mini)){
        ++count;
        maxi = (maxi, V[i]+V[i+1]);
    }
  }
  cout << count << ' ' << maxi << endl;        // результат решения
  return 0;
}
```

Первая часть заканчивается закрытием входного файла. Нужно сказать, что необязательно читать файл в вектор. Тогда вторая часть выполняется как повторное чтение файла (напишите этот вариант самостоятельно).

Для проверки работы вы должны создать небольшой файл из нескольких чисел, удовлетворяющих условию задачи 17. Имя файла должно быть demo_2025_17.txt, и поместить его надо в тот же каталог, где находится исходный код программы.

Отмечу одну особенность олимпиадных и ЕГЭ-задач — обрабатывать ошибки во входных данных не требуется, т. к. в файлах предоставляются только корректные данные. Поэтому, если в файле первой строкой задается количество последующих чисел, то ввод можно сделать очень простым.

Создается вектор `N` элементов:

```
int N;                                  // количество записей файла
ifstream strm("input.txt");             // входной файл
strm >> N;                              // ввод размера
vector<int> V(N);                 // количество записей известно
```

И тогда ввод данных из файла выполняется с помощью входных итераторов:

```
copy(istream_iterator<int>(strm), istream_iterator<int>(), begin(V));
```

Не нужно писать цикл, не требуется проверять конец файла — это делается внутри стандартной функции `copy`.

Задачи и упражнения

1. Написать программу, которая генерирует вещественные числа в диапазоне от `-3` до `+3` и выводит их в файл по два на строке через пробел.
2. Файл из предыдущей задачи — это координаты x, y точек на плоскости. Считайте данные из файла и найдите минимальное расстояние между точками.
3. В задаче 26 КИМ ЕГЭ 2025 требуется обрабатывать следующий входной файл:

   В первой строке входного файла находится число *N*, обозначающее количество студентов (целое положительное число, не превышающее 10 000). Каждая из следующих *N* строк содержит 5 чисел через пробел: ID студента (целое положительное число, не превышающее 100 000) и четыре оценки, полученные им за сессию. Гарантируется, что общее число студентов *N* кратно 4 и хотя бы один студент имеет более двух «двоек». Во входном файле все ID различны.

   Пример файла:

   ```
   8
   4 4 4 4 4
   7 5 5 5 2
   10 3 4 4 5
   1 4 4 4 3
   6 3 5 5 3
   2 2 2 2 2
   13 2 2 2 3
   3 3 3
   ```

   Напишите программу, которая формирует такой файл.

- Урок 20 -

Большие программы

Все наши программы, которые мы писали ранее, помещались в один файл с расширением cpp. Но это только потому, что они были не очень большие. Реальные программы состоят из сотен тысяч строк кода. Содержать код такого размера в одном файле — это значит создавать большие проблемы при компиляции и сборке. Если в программе из сотни тысяч строк кода изменить всего одну константу, то компилятору придется заново компилировать всю сотню тысяч строк, хотя они не изменились. А потом линкеру придется все снова собирать. С таким размером кода это будет занимать много времени.

Поэтому программы делят на части. Обычно отдельная часть большой программы называется *модулем*[1]. Разбиение программы на части выгодно по многим причинам: каждый модуль разрабатывается одним программистом; с небольшим модулем удобнее работать; разбиение улучшает инкапсуляцию, что облегчает обнаружение и исправление ошибок.

При разбиении программ на модули каждый из них компилируется отдельно, и потом линкер собирает все вместе в исполняемую программу — такой подход называется *раздельной* трансляцией. Если в большой программе изменился только один модуль, то только он и транслируется. Раздельная трансляция поддерживается и интегрированной средой, и языком C++.

Разделение программы на модули — это довольно сложная работа, которая требует вдумчивого анализа. Ведь требуется так разделить программу, чтобы модули как можно меньше зависели один от другого. А еще лучше, если они вообще не зависят друг от друга. Но так не бывает, т. к. из одного модуля обычно требуется вызвать функции, объявленные в других модулях. Тем не менее нужно стремиться к минимизации таких зависимостей.

Этот урок посвятим разработке системы обучения арифметике для начальных классов, когда школьники учатся работать с целыми числами. А мы поучимся разделять программу на модули.

---

[1] В данном случае под модулями понимаются файлы исходного кода. Конструкция модуля включена в стандарт C++20, однако пока компиляторы реализуют эту конструкцию с некоторыми ограничениями. Поэтому мы сделаем разделение проекта на части традиционным способом, как это всегда делалось в C/C++.

Проект «Арифметика»

Мы уже писали пару довольно больших программ-тренажеров (см. *урок 8*). Теперь объединим эти тренажеры в один проект, разделим его на отдельные модули и добавим к проекту новые тренажеры. Эта работа — самая настоящая разработка реального программного продукта.

Сначала надо вернуться к нашим программам и проанализировать, что у них есть общего, а что значительно отличается.

Во-первых, очевидно, что функция для вывода приветствия у нашей большой программы должна быть одна, а не у каждого тренажера в отдельности.

Во-вторых, функции, выводящие инструкции в консоль, очень похожи и отличаются только содержанием инструкции. Поэтому желательно иметь единственную функцию вывода инструкции, а сам текст как-то передавать ей в качестве параметра. Причем, очевидно, что текст инструкции неразрывно связан с тренажером и является его частью.

В-третьих, в проект включим только тренирующие программы, и исключим демонстрацию таблицы умножения. Сначала реализуем два тренажера: по умножению и по сложению двузначных чисел. Когда система в таком виде заработает, добавим тренажеры вычитания и деления.

И последнее: в функции `menu` выводится меню для выбора тренажера и вводится ответ пользователя (см. листинг 8.7). Поэтому все тренажеры должны быть отдельными функциями, вызываемыми в функции меню. Тексты инструкций связаны с тренажерами, и поэтому вывод инструкции нужно вызвать перед запуском тренажера.

Этих соображений достаточно для начала разработки.

Разделение на модули

Сразу «возьмем быка за рога» и создадим проект `Trainers`. Как обычно, в проекте присутствует функция `main` в файле main.cpp. Это будет главный модуль нашей обучающей системы. В этом модуле реализуется интерфейс с пользователем, т. е. должна быть определена функция вывода приветствия и функция вывода меню, а также главная функция, которая их вызывает.

Добавим в проект четыре пустых файла:

- mul_trainer.cpp — тренажер умножения (таблица умножения);
- add_trainer.cpp — тренажер сложения (двузначных чисел);
- sub_trainer.cpp — тренажер вычитания (двузначных чисел);
- div_trainer.cpp — тренажер деления.

Я задал «говорящие» имена файлов, и такие имена показывают назначение каждого модуля. Таким образом, в проекте сейчас пять модулей (вместе с главным модулем).

В главном модуле main.cpp определим следующие функции:

- `hello` — функция приветствия;
- `menu` — функция вывода меню;
- `main` — главная функция, вызывающая эти две.

После рефакторинга функции `menu` из тренажера по таблице умножения (см. листинг 8.7) главный модуль станет таким (листинг 20.1).

Листинг 20.1. Главный модуль системы «Арифметика»

```
#include <iostream>
using namespace std;
void hello(){
    cout << "Вас приветствует программа-тренер Арифметика.\n";
    cout << "Как вас зовут ?\n> ";
    string name;
    cin >> name; cin.ignore();      // удаление из буфера ввода '\n'
    cout << "Приветствую вас, " + name + "!\n";
}
int menu(){
    int number = -1;
    cout << "Вы можете выбрать:\n"
    "\t1. Умножение (таблица умножения)\n"
    "\t2. Сложение\n"
    "\t3. Вычитание\n"
    "\t4. Деление\n"
    "\t0. Закончить программу\n";
    cout << "Ваш выбор: ";
    cin >> number;  cin.ignore();   // удаление из буфера ввода '\n'
    switch(number){
            case 1: multrainer();   break;
            case 2: addtrainer();   break;
            case 3: subtrainer();   break;
            case 4: divtrainer();   break;
            case 0: break;
            default: cout << "Такого пункта меню нет!\n";
    }
    return number;
}
int main(){
    hello();                            // приветствие
    int choice = -1;
    while(choice){
        choice = menu();                // выбор и запуск тренажера
    // здесь можно собирать статистику выбора тренажера
    }
```

```
    cout << "Game Over!" << endl;
    return 0;
}
```

Главная функция не изменилась. В функциях `hello` и `menu` после оператора ввода добавили удаление символа `'\n'` из входного потока.

Программа пока не транслируется, т. к. не определены функции тренажеров. Определим пока пустые функции в каждом модуле, как мы это уже делали ранее (см. *урок 8*), но собирать пока не будем. Надо заняться разработкой первого модуля. Это будет модуль тренажера умножения mul_trainer.cpp.

Вывод инструкции

Любой тренажер должен выводить инструкцию, поэтому начнем разработку с определения функции вывода инструкции. Эта функция должна быть общей для всех тренажеров, поэтому она должна находиться в главном модуле.

Сама инструкция должна находиться в модуле тренажера. Пока определим ее как обычную строковую константу. Инструкцию по умножению возьмем из тренажера по таблице умножения (см. *урок 8*, рис. 8.2). Выглядит она так:

```
const string instruction {
    "\nВам будут предложены задания в форме\n"
    "a * b = \n"
    "Например:\n"
    "9 * 8 = \n"
    "Вы должны ввести ответ с клавиатуры (72)\n"
    "\nКоличество ошибок подсчитывается\n"
    "Для окончания тренировки нужно задать ноль (0)\n"s
};
```

Тогда прототип функции вывода может выглядеть так:

```
void print_help(string_view instruction);       // прототип
```

Параметр единственный, и его тип `string_view`, т. к. надо просто вывести строку в консоль. Сама функция определена в главном модуле:

```
void print_help(string_view instruction){
    cout << instruction;                        // вывод инструкции
    cout << "\nЕсли все прочитали, нажмите <Enter>\n";
    cin.get();                                  // ожидание <Enter>
}
```

После вывода инструкции функция приостанавливает свою работу, ожидая, когда пользователь прочтет выведенный текст и нажмет клавишу <Enter>.

В модуле тренажера определена функция-тренажер, которая на данном этапе содержит только вызов функции вывода инструкции:

```
void multrainer (){
    print_help(instruction);
}
```

Для того чтобы в модуле тренажера можно было вызвать функцию вывода инструкции, нужно прописать ее прототип в этом модуле.

Первая сборка

По такой же схеме оформляем все остальные модули. На данном этапе каждый модуль с тренажером выглядит так, как показано в листинге 20.2 (на примере тренажера вычитания).

Листинг 20.2. Общий вид кода в модуле с тренажером

```
#include <string>
using namespace std;
const std::string instruction {
    "\n Вам будут предложены задания в форме\n"
    " ... ... ... ... ... ... ... ... ... ... "
    "\nКоличество ошибок подсчитывается\n"
    "Для окончания тренировки нужно задать ноль (0)\n"s
};
void print_help(string_view instruction);       // прототип
void subtrainer (){                             // имя функции тренажера
    print_help(instruction);
}
```

Чтобы собрать все воедино, нужно в главном модуле прописать прототипы функций тренажеров:

```
void multrainer ();
void addtrainer ();
void subtrainer ();
void divtrainer ();
```

Все собирается и запускается. Можно вывести каждую инструкцию в консоль и прочесть. Программа будет ждать нажатия клавиши <Enter>.

Обратите внимание на то, что в Code::Blocks внизу **Build log** выводятся сообщения о трансляции каждого модуля (всего пять сообщений). Это и есть раздельная трансляция, о которой было сказано в начале урока. И последнее шестое сообщение выводит линкер, собирая все модули в один исполняемый файл.

Но немедленно возникает вопрос: почему компилятор не обращает внимания на то, что имена всех инструкций одинаковые?

Межмодульное взаимодействие

Для того чтобы ответить на этот вопрос, нужно вспомнить о видимости имен и познакомиться с понятиями «единица трансляции» и «связывание».

Из-за отсутствия настоящих модулей в языке C++ было определено понятие «единица трансляции». *Единица трансляции* — это файл с исходным кодом на C++, который получается после обработки препроцессором.

Вспомним *урок 2*: мы там выяснили, что директиву `#include` нельзя разбить на две строки, т. к. возникает ошибка трансляции:

```
error: #include expects "FILENAME" or <FILENAME>
```

Это сообщение говорит о том, что операндом директивы является имя файла, задаваемое в угловых скобках или в кавычках. Естественно, код в этом файле написан на C++. Таким образом, препроцессор[2] на место директивы `#include` подставляет код из указанного файла. После этого «чистый» файл отдается компилятору «на растерзание».

Раздельная трансляция требует согласования объявлений и определений в разных единицах трансляции. По стандарту единица трансляции не должна содержать более одного определения любой переменной или константы, функции, класса и шаблона (в каждой области видимости, естественно). Это — так называемое правило одного определения (One-Definition Rule, ODR). Во всех единицах трансляции должны быть согласованы объявления и определения переменных, констант, функций, классов и шаблонов. Если какие-то объявления и определения имен не согласованы, то мы получим ошибку либо при компиляции, либо при сборке (компоновке).

Получается, что в нашей системе все определения и объявления оказались согласованы. Это определяется связыванием и видимостью имен.

Связывание

Каждое имя, обозначающее объект, имеет некоторую область видимости, в которой это имя объявлено и может использоваться. Перечислим области видимости (по возрастанию «объема»), с которыми мы уже имели дело:

- *Оператор*. Идентификатор, объявленный в условии операторов `if`, `switch`, `while` или в операторе `for`, действителен только до конца тела оператора;
- *Прототип*. Идентификаторы, указанные в списке параметров прототипа функции, имеют областью действия только прототип функции;
- *Блок*. Имена, объявленные в блоке, являются локальными в этом блоке и не видны вне его. Параметры, объявленные в заголовке функции, видимы до конца блока определения и локальны в этом блоке;
- *Функция*. Только метки имеют такую область действия. В одной функции все метки должны быть разными;
- *Класс*. Все имена, объявленные в классе, видимы внутри него независимо от модификатора доступа. Вне класса доступны имена с модификатором доступа `public`;

---

[2] Более детально с работой препроцессора мы познакомимся ниже.

- *Файл*. Имена объектов, определенных вне перечисленных выше областей видимости, видимы в пределах единицы трансляции. За счет объявлений такие имена можно использовать и в других единицах трансляции;
- *Пространство имен*. C++ позволяет явно задать область видимости имен, присвоив области видимости некоторое имя.

> ***На заметку***
> В стандарте определены и еще несколько других областей видимости.

Нас в первую очередь интересуют две последние области видимости. Имена в одной области видимости не должны быть одинаковыми, но в разных областях они могут совпадать.

Кроме области видимости каждое имя обладает еще одним свойством — связыванием. *Связывание* (linkage) идентификатора определяет, относятся ли другие объявления с этим именем к тому же объекту или нет.

В стандарте C++17 определено три вида связывания:

- без связывания (no linkage);
- внутреннее связывание (internal linkage);
- внешнее связывание (external linkage).

Первое нас не интересует, т. к. фактически устанавливает, что имя объекта связывания не имеет. Такими объектами являются локальные переменные.

У нас в каждом модуле (файле, единице трансляции) определено два имени:

- имя инструкции, которое является именем константы;
- имя функции тренажера.

Оба имени имеют одинаковую область видимости (единица трансляции), но обладают разным связыванием. Первое (имя константы) имеет внутреннее связывание, а второе (имя функции) — внешнее.

Внутреннее связывание делает объект локальным в единице трансляции. Такими объектами по умолчанию является константы. Поэтому разрешается в разных файлах объявлять константы с одинаковыми именами, и это будут определения разных объектов.

Внешнее связывание запрещает определять объекты с одинаковыми именами в разных единицах трансляции. Поэтому имена всех наших функций разные. Нельзя определить две функции с одинаковыми именами в разных файлах — это будет ошибка при сборке. Компоновщик сообщит, что обнаружено повторное определение функции. Зато прототип (объявление) функции можно писать в любых файлах. Что мы и сделали, написав прототипы функций тренажеров в главном модуле, а объявление функции вывода инструкции указали в модулях.

Внешним связыванием обладают имена глобальных переменных. Нельзя объявить две глобальные переменные в разных файлах с одним и тем же именем — компоновщик сообщит о повторном определении.

Пространства имен

Возникают вопросы: можно ли поменять связывание (внешнее изменить на внутреннее и наоборот)? И при чем здесь пространства имен?

Дело в том, что все глобальные имена с внешним связыванием попадают в неявно определенное глобальное пространство имен. Это глобальная область видимости. А в одной области видимости нельзя определять два одинаковых имени. Поэтому нельзя определить две функции с одинаковыми именами — это нарушает ODR. И с глобальными переменными такая же картина.

Я уже упоминал о том, что глобальные переменные — это не очень хорошо. В разработке программ нужно стремиться к инкапсуляции — это, как показала мировая практика, снижает количество ошибок при разработке. Инкапсуляция состоит в том, что надо объект помещать в как можно меньшую область видимости. Поэтому в языке C++ появилась совсем маленькая область видимости — оператор.

С этой «колокольни» нам в тренажерах лучше иметь объекты с внутренним связыванием, а не внешним. Поэтому нам интересны способы изменения внешнего связывания на внутренне. Самый современный из них состоит в том, что в каждой единице трансляции явно объявляется пространство имен.

Конструкция очень простая:

```
namespace {
// элементы пространства имен
}
```

Такое пространство имен называется анонимным или безымянным. И все элементы внутри него становятся локальными в единице трансляции. И тогда проблем с одинаковыми именами в разных единицах трансляции не будет.

Добавим тренажеры

Теперь надо сделать новый рефакторинг нашего проекта и подключить к проекту тренажер умножения (см. *урок 8*, листинг 8.6). Но лучше рассмотрим и проанализируем код сразу двух тренажеров, чтобы выявить общие черты и облегчить реализацию и подключение следующих тренажеров. Итак, снова заглянем в код листингов 8.6 и 8.7. В обоих тренажерах видим, что основная работа выполняется в цикле до ввода ответа-нуля. Тело цикла включает следующие действия:

- генерируются два случайных числа — операнды тренируемой операции;
- увеличивается счетчик заданий;
- выводится задание по заданному формату — для каждого тренажера свой формат вывода;
- вводится ответ;
- если ответ не ноль, то выполняется проверка — для каждого тренажера проверка индивидуальна;
- если ответ неправильный, то увеличивается счетчик ошибок.

Перед циклом инициализируется датчик случайных чисел, а после выводится количество заданий и количество совершенных ошибок. Для инициализации датчика реализована функция `time` (см. листинг 8.4).

Таким образом, эти два тренажера имеют много общего.

Поэтому нужно написать единственную функцию, в которой выполняется цикл вывода и проверки задания, а специфические для каждого тренажера элементы передавать ей как параметры.

Мы видим, что реализация тренажеров разделяется на две части:

- общие для всех тренажеров функции;
- функции, специфические для конкретного тренажера.

При разработке тренажера сложения (см. листинг 8.7) мы уже выделили один конкретный элемент в отдельную функцию вывода задания `task`. Очевидно, что и проверку ответа надо тоже сделать отдельной функцией. Кроме того, диапазон операндов у наших двух тренажеров — тоже разный.

Напишем псевдокод общей функции для всех тренажеров, исходя из приведенных выше соображений (листинг 20.3).

Листинг 20.3. Псевдокод общей функции тренажеров

```
void common_trainer(trainer_t trn){
    random_t gen { unsigned(time()) };
    auto [a,b] = trn.uid;        // получили от тренажера
    distribution_t uid(a, b);    // установили интервал операндов
    // объявление переменных
    task_t counters {0,0};       // счетчики заданий
    int answer {-1};             // ответ
    while(answer){
        trn.task(gen,uid);       // вывод задания
        std::cin >> answer;      // ввод ответа
        std::cin.ignore();
        if(answer){              // проверка ответа
            counters.errors += trn.checkout(answer);    // ошибки
            ++counters.tasks;
        }
    }
    statistics(counters);        // вывод счетчиков
}
```

Функция еще не реализована, поэтому пока мы имеем псевдокод, очень похожий на код C++. В этой функции псевдокодом является следующее:

1. Предполагается, что при вызове функция получает от тренажера параметр `trn`, в котором находится все необходимое для работы; тип параметра предполагается `trainer_t`.

2. В функции инициализируется датчик `gen` и выставляется диапазон случайных чисел, который функция получает в `trn` от тренажера.
3. В цикле сначала вызывается функция тренажера `task`, и ей передаются `gen` и распределение `uid`, чтобы тренажер сгенерировал очередные два операнда и вывел задание в консоль.
4. После ввода ответа, если он не ноль, вызывается функция проверки ответа `checkout` (полученная от тренажера), которая при ошибке возвращает 1, и эта единица увеличивает счетчик ошибок.

После цикла вызывается функция `statistics` для вывода счетчиков заданий и ошибок. Причем сами счетчики собраны в некую структуру типа `task_t`. При модификации системы эту функцию можно будет заменить на другую, не меняя код функции `common_trainer`.

Параметр `trn`, передаваемый тренажером в эту функцию, должен содержать:

- диапазон случайных чисел, генерируемых для заданий тренажера;
- имя функции-задания (в листинге 20.3 написано `task`);
- имя функции проверки ответа (в листинге 20.3 написано `checkout`).

Реализуем типы, необходимые для работы тренажера (код в листинге 20.4).

Листинг 20.4. Типы для работы тренажера

```
// типы для тренажеров
using random_t = std::default_random_engine;
using distribution_t = std::uniform_int_distribution<>;
using operands_t = std::tuple<int,int>;          // текущие операнды
using interval_t = std::tuple<int, int>;         // интервал операндов
using trainer_t  =                               // данные для функции common_trainer
struct data_trainer{
    interval_t uid;                              // интервал генерируемых операндов
    std::function<void(random_t&,distribution_t)>task;
    std::function<int(int)>checkout;
};
```

Как видите, названия типов «говорящие».

Тип для представления интервала случайных чисел `interval_t` определен как синоним `std::tuple<int, int>`. Английское слово tuple переводится как «кортеж». Этот класс-шаблон реализован в STL и предназначен для создания набора разнотипных элементов, чтобы не использовать для этого структуру. Текущие операнды тренажера представлены синонимом кортежа `operands_t`. Для работы с кортежами в библиотеке реализовано много удобных средств. Конечно, нужно подключить библиотеку `<tuple>`.

Еще более интересным является тип `function`. Обратите внимание на его шаблонный аргумент: в нем задается прототип функций (без имени), которые имеют ука-

занный список параметров и тип возвращаемого результата. Имена таких функций могут быть присвоены объекту этого типа. Например, во втором объекте `function` задан тип `int(int)`. Это означает, что функция, имя которой может быть присвоено объекту `checkout`, должна иметь прототип:

```
int имя (int);
```

Теперь посмотрим, как реализовать тренажер (код показан в листинге 20.5).

Листинг 20.5. Тренажер умножения

```
operands_t current;                              // текущие операнды
// генерация операндов
auto next_operands = [](random_t &gen, distribution_t uid){
    auto [left, right] = current;
    left = uid(gen); right = uid(gen);
    return (current = make_tuple(left, right));
};
// вывод задания
auto task = [](random_t &gen, distribution_t uid){
    auto[L,R] = current = next_operands(gen, uid);     // операнды
    cout <<' ' << setw(2) << L << " *" << setw(2) << R << " = ";
};
// функция проверки ответа
auto checkout = [](int answer){
    auto op = [](operands_t t){          // операция тренажера
        auto [left, right] = t;
        return left * right;             // операция
    };
    return int(op(current)!= answer);    // проверка
};
```

Инструкцию я не стал приводить, чтобы не занимать место. Напомню, что все это должно быть написано внутри пространства имен.

После пространства имен надо написать главную функцию тренажера (листинг 20.6).

Листинг 20.6. Главная функция тренажера умножения

```
void multrainer (){
    print_help(instruction);                     // вывод инструкции
    trainer_t t {{2,9}, task, checkout };        // объект для общей функции
    common_trainer(t);                           // вызов общей функции
}
```

Этот код тренажера содержит именно те функции, о которых мы говорили ранее: генерация операндов `next_operands`, вывод задания `task`, проверка ответа `checkout` и главная функция тренажера `trainer_table`. Все функции, кроме главной, написаны

как лямбда-функции. Главная функция тренажера содержит всего три строки! Вторая — это создание объекта для передачи его в общую функцию тренажеров, которая вызывается в третьей строке.

Сравним реализацию тренажера сложения (листинг 20.7).

Листинг 20.7. Тренажер сложения

```
operands_t current;              // текущие операнды
auto next_operands = [](random_t &gen, distribution_t uid){
    auto [left, right] = current;
    left = uid(gen); right = uid(gen);
    return (current = make_tuple(left, right));
};
auto task = [](random_t &gen, distribution_t uid){
    auto[L,R] = current = next_operands(gen, uid);
    cout << setw(3) << L << "\n+ \n"
         << setw(3) << R << "\n----\n";
};
auto checkout = [](int answer){
    auto op = [](operands_t t){
        auto [left, right] = t;
        return left + right;
    };
    return int(op(current)!= answer);    // проверка
};
// главная функция тренажера
void addtrainer (){
   print_help(instruction);
   trainer_t t {{10,99},task,checkout };         // объект для общей функции
   common_trainer(t);                            // вызов общей функции
}
```

Изменилась функция вывода задания `task` и вложенная лямбда `op` в функции проверки. Этот новый код показывает, что добавлять тренажеры в систему стало очень просто: надо скопировать тренажер для таблицы умножения и модифицировать некоторые строки в коде.

Мы видим, что функция генерации операндов не изменилась. Но она не является универсальной для всех тренажеров. Подумаем о вычитании. В заданиях на вычитание, как обычно, генерируются два операнда, но первый операнд должен быть больше второго. Нужно сгенерировать два числа и присвоить уменьшаемому максимум, а вычитаемому — минимум:

```
auto next_operands = [](random_t &gen, distribution_t uid){
    auto [left, right] = current;
    left = uid(gen); right = uid(gen);
```

```
    current = make_tuple(max(left, right),min(left,right));
    return current;
};
```

Теперь подумаем о тренажере для деления. В нем тоже надо генерировать два операнда, но делимое должно быть кратно делителю. Поэтому сначала генерируется делитель, а делимое получается умножением делителя на небольшой (например, от 2 до 9) случайный множитель.

Таким образом, генерацию операндов надо писать для каждого тренажера индивидуально.

Вывод из описанной работы совершенно однозначный: предварительный анализ задачи (или уже имеющейся программы) и разделение ее на части очень облегчают последующий рефакторинг, модификацию и расширение.

Препроцессор и модули

Теперь нам надо пересмотреть деление нашего проекта на модули.

Каждый тренажер должен, как и прежде, задаваться в отдельном модуле. Но появляется общий для всех тренажеров модуль, содержащий общие функции: функцию инициализации датчика случайных чисел, функцию вывода инструкции, общую функцию тренажеров и функцию вывода счетчиков `statistics`. Пусть этот модуль называется all_trainers.cpp.

Таким образом, в системе должны быть следующие модули:

- all_trainers.cpp — общие функции для всех тренажеров;
- mul_trainer.cpp — тренажер умножения (таблица умножения);
- add_trainer.cpp — тренажер сложения (двузначных чисел);
- sub_trainer.cpp — тренажер вычитания (двузначных чисел);
- div_trainer.cpp — тренажер деления;
- main.cpp — главная функция и приветствие.

Но возникает одна большая проблема: типы для тренажеров (см. листинг 20.4) должны быть известны и всем тренажерам, и функции `common_trainer`. Можно написать типы во всех модулях, где они требуются, но это не очень хорошая практика. Лучше определить их один раз, но сделать так, чтобы их видели все нужные модули. Решить эту проблему помогает препроцессор.

Создадим новый файл с расширением h и включим в него типы для тренажеров и прототипы функций (код показан в листинге 20.8).

Листинг 20.8. Включаемый файл trainers.h

```
#ifndef TRAINERS_H_INCLUDED              // страж включения
#define TRAINERS_H_INCLUDED
// общие для всех тренажеров стандартные библиотеки
```

```
#include <iostream>
#include <iomanip>
#include <string_view>
#include <string>
#include <tuple>
#include <random>
#include <chrono>
#include <functional>
// типы для тренажеров
... см. листинг 20.04
// прототипы функций
void print_help(std::string_view instruction); // вывод инструкции
void common_trainer(trainer_t trn);            // общая функция
#endif // TRAINERS_H_INCLUDED
```

В каждом модуле (кроме главного) надо прописать директиву препроцессора:

```
#include "trainers.h"
```

Первая строка этого файла является условной директивой препроцессора, в которой проверяется, было ли ранее определено имя `TRAINERS_H_INCLUDED`. Окончание условной директивы — последняя строка с директивой `#endif`. Внутри условной директивы помещается определение проверяемого имени `TRAINERS_H_INCLUDED` с помощью директивы `#define`. Это имя — «страж» включения.

Эти директивы созданы автоматически при создании h-файла. Буква h — это первая буква слова header (заголовок). Такие файлы еще в языке C назывались заголовочными и всегда были устроены так, как сгенерировала нам среда. Директивы обеспечивают то, что заголовочный файл будет включен в любой исходный файл не более одного раза. Это важно, если в таком файле присутствуют, например, определения классов. В нашей системе все прекрасно собирается и работает даже без этих директив, но лучше оставить все как есть.

Задачи и упражнения

1. Напишите полный тренажер вычитания и подключите его к системе «Арифметика».
2. По аналогии с системой «Арифметика» реализуйте систему «Площади» для тренировок вычислений площадей различных геометрических фигур: прямоугольника, треугольника, трапеции, круга.
3. Реализуйте тренажер деления, создавая операнды следующим образом: генерируются два случайных числа, как в тренажере умножения, в диапазоне от 2 до 9. Произведение случайных чисел становится делимым, а любой из сомножителей — делителем. Текст задания выводить в следующем виде: `a / b = `.

Подведение итогов

Я поздравляю вас, читатель — вы добрались до последней страницы этой книги. Это было нелегко, но вы справились! Было изучено множество новых понятий, незнакомых терминов, конструкций языка C++. Вместе с автором были написаны более сотни учебных программ. И даже реализован самый настоящий реальный проект для обучения основам арифметики в начальной школе.

Надеюсь, вы решали задачи после каждого урока. Если вы этого еще не делали, то самое время начать делать это прямо сейчас — это тоже порядка сотни новых программ, которые вы напишите самостоятельно. И не бойтесь экспериментировать с кодом.

Главное — вы научились программировать! Однако это только самое начало пути. В книге мы только чуть-чуть коснулись объектно-ориентированного программирования, слегка окунулись в стандартную библиотеку и немного познакомились с обобщенным программированием.

Программирование — гораздо более широкая сфера деятельности по сравнению с затронутыми в этой небольшой книжке основами C++. Надеюсь, эти уроки вдохновят вас продолжить изучение языка и подумать о профессиональной карьере программиста. Дальнейшее освоение программирования на C++ включает:

- объектно-ориентированное программирование;
- обобщенное программирование;
- параллельное программирование.

И конечно, вместе с языком C++ абсолютно необходимо изучать структуры данных и алгоритмы, с основами которых вы познакомились в этой книге. В этом вам поможет перечень книг и интернет-ссылок, которые приведены в *приложениях 1* и *2*.

Главное — не переставать работать. Упорство и настойчивость в достижении цели — одно из важнейших качеств характера программиста. Успехов вам, читатель, в дальнейшем освоении программирования и языка C++!

Приложения

ПРИЛОЖЕНИЕ 1. СПИСОК ЛИТЕРАТУРЫ

ПРИЛОЖЕНИЕ 2. ПЕРЕЧЕНЬ РЕСУРСОВ ИНТЕРНЕТА

ПРИЛОЖЕНИЕ 3. УСТАНОВКА CODE::BLOCKS

ПРИЛОЖЕНИЕ 4. ИНТЕГРИРОВАННАЯ СРЕДА CODE::BLOCKS

- Приложение 1 -

Список литературы

Учебники по современному C++:

1. Прата С. Язык программирования C++. Лекции и упражнения, 6-е изд. — М.: ООО «И.Д. Вильямс», 2012. — 1248 с.
2. Липпман С.Б., Лажойе Ж., Му Б.Э. Язык программирования C++. Базовый курс, 5-е изд. — М.: ООО «И.Д. Вильямс», 2014. — 1120 с.
3. Страуструп Б. Программирование: принципы и практика с использованием C++, 2-е изд. — М.: ООО «И.Д. Вильямс», 2016. — 1328 с.

Книги для программистов:

4. Лоспинозо Д. C++ для профи. — СПб.: Питер, 2021. — 816 с.
5. Уильямс Э. C++. Практика многопоточного программирования. — СПб.: Питер, 2020. — 640 с.
6. Вандевурд Д., Джосаттис Н.М., Грегор Д. Шаблоны C++. Справочник разработчика, 2-е изд. — СПб.: ООО «Альфа-книга», 2018. — 848 с.
7. Джосаттис Н.М. Стандартная библиотека C++: справочное руководство, 2-е изд. — М.: ООО «И.Д. Вильямс», 2014. — 1136 с.
8. Сэджвик Р. Алгоритмы на C++. — М.: ООО «И.Д. Вильямс», 2011. — 1056 с.

- Приложение 2 -

Перечень ресурсов Интернета

Русскоязычные ресурсы:

https://code-basics.com/ru/languages/cpp — курс обучения на C++

https://stepik.org/course/363/syllabus — очень хороший курс обучения на C++

https://metanit.com/cpp/ — сайт о программировании на C++

http://cppstudio.com/cat/274/ — сайт о программировании на C++

https://ru.wikipedia.org/wiki/C%2B%2B17 — нововведения стандарта C++17

https://ru.wikipedia.org/wiki/C%2B%2B20 — нововведения стандарта C++20

https://cpp-kt.github.io/cpp-notes/course.html — конспекты курса по C++

https://ru.stackoverflow.com/ — вопросы и ответы по программированию

Англоязычные ресурсы:

https://eel.is/c++draft/ — текущий черновик стандарта

https://en.cppreference.com/w/cpp — лучший справочник по C/C++

https://cplusplus.com/doc/tutorial/ — прекрасный учебник по C++

https://www.stroustrup.com/C++.html — сайт Б. Страуструпа о C++

https://cpp.sh/ — простой онлайн-компилятор

- ПРИЛОЖЕНИЕ 3 -

Установка Code::Blocks

Установка Code::Blocks осуществляется в два этапа:

1. Загрузить установщик среды с сайта разработчиков.
2. Запустить установщик, и он установит Code::Blocks на ваш компьютер.

> ***ПРИМЕЧАНИЕ***
> Программисты обычно говорят не «установщик», а «инсталлятор». Это слово происходит от английского термина installer, которое так и переводится — установщик.

Загрузка установщика в Windows

Запустите браузер и наберите в адресной строке адрес:

`https://www.codeblocks.org/downloads/`

На экране должна появиться страница загрузки (рис. П3.1)

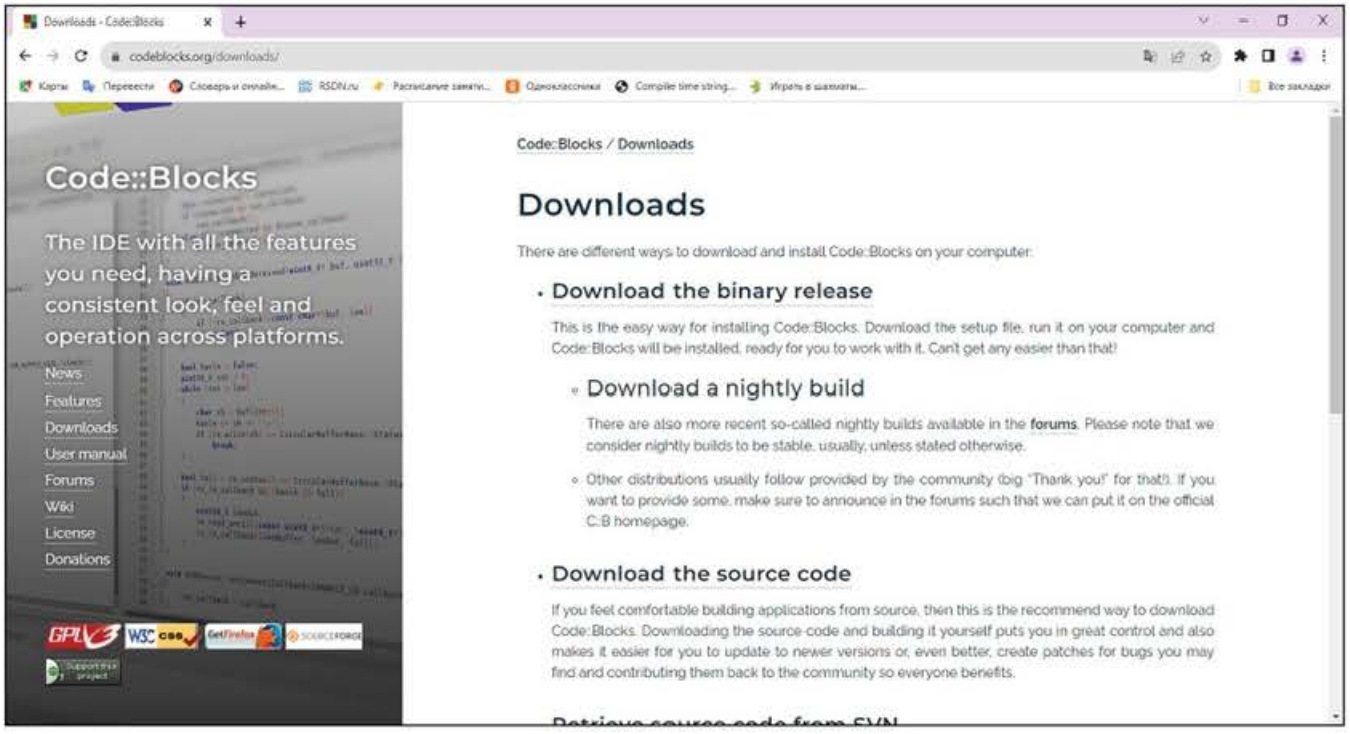


Рис. П3.1. Страница загрузки Code::Blocks

На этой странице щелкните мышью по надписи **Download the binary release**. В результате на экране должна появиться страница с вариантами загрузки для разных операционных систем (рис. П3.2).

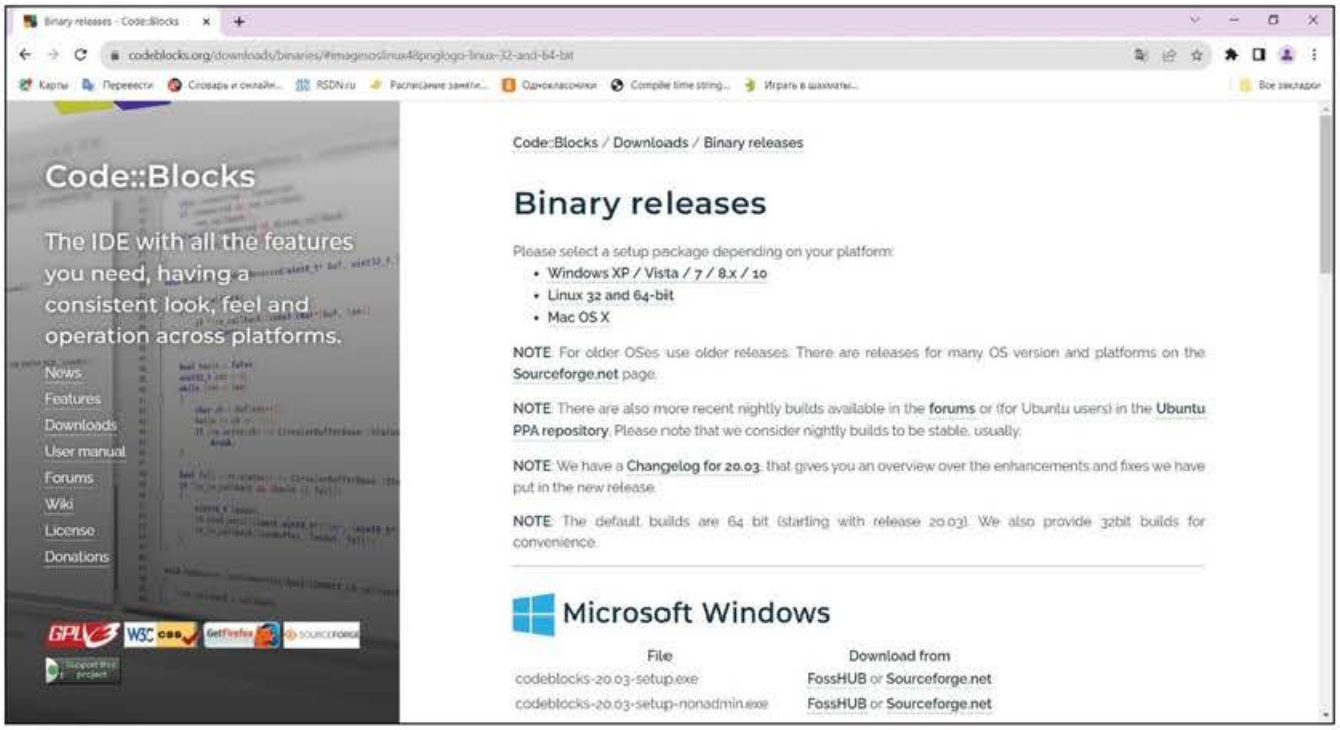


Рис. П3.2. Варианты загрузки для разных операционных систем

Щелкните мышью по надписи **`Windows XP/Vista/7/8.x/10`** и переместитесь к списку файлов для загрузки (рис. П3.3).

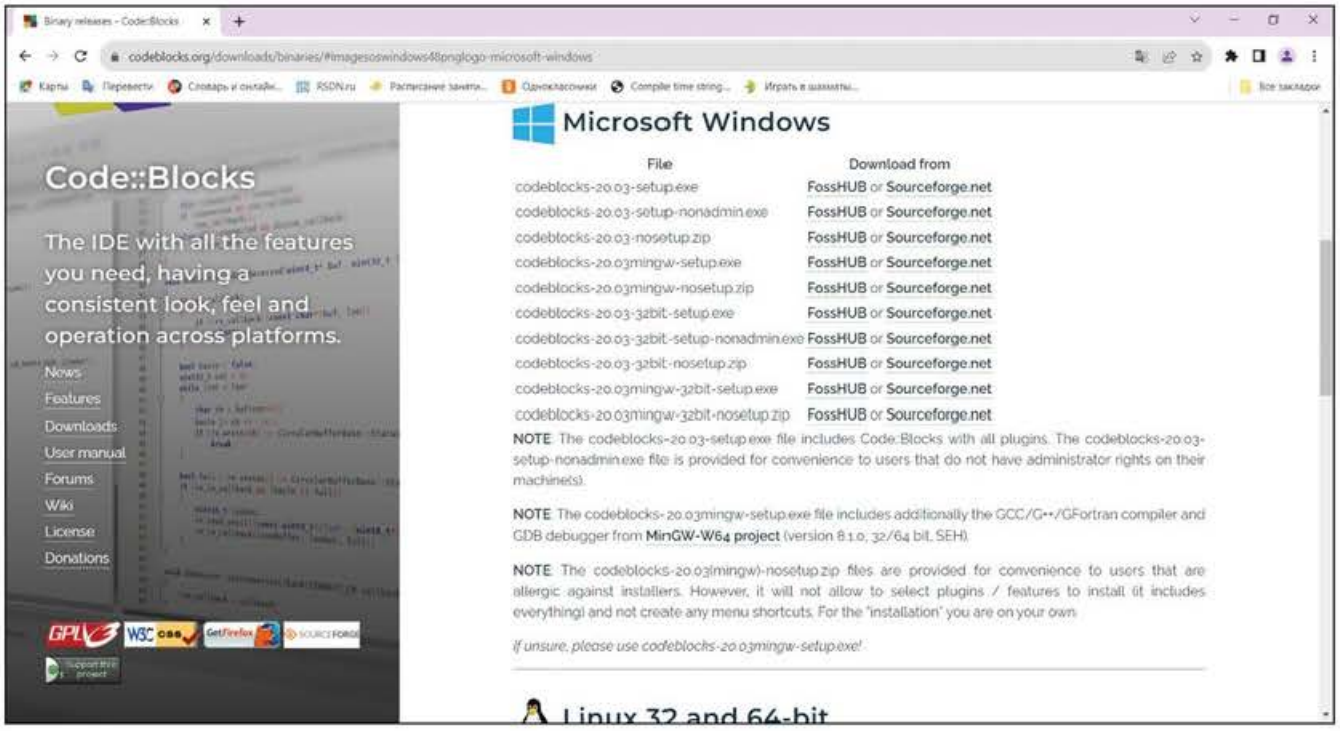


Рис. П3.3. Страница со списком файлов

Из этого списка надо выбрать файл codeblocks-20.03mingw-setup.exe и щелкнуть мышью по ссылке **Sourceforge.net** справа от названия файла. Откроется страница загрузки (рис. П3.4), и через некоторое время автоматически начнется загрузка указанного файла.

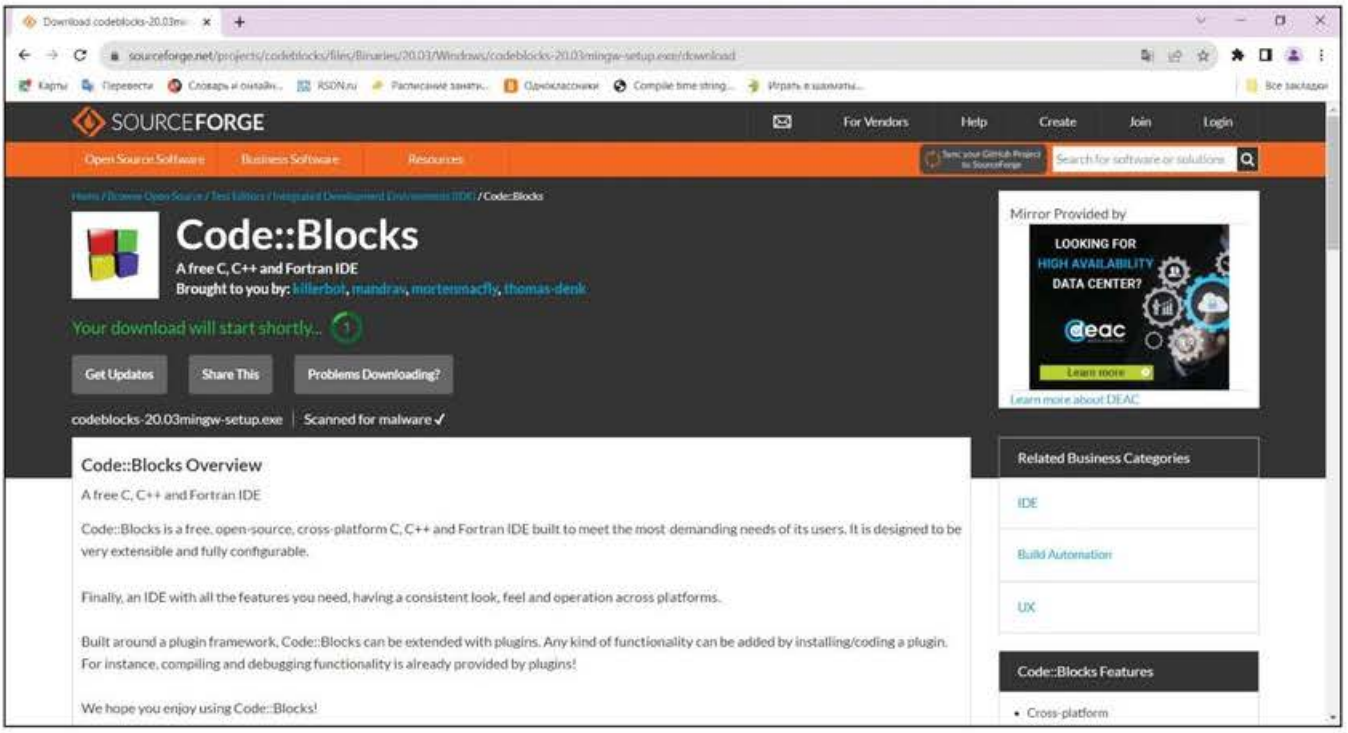


Рис. П3.4. Страница Sourceforge.net загрузки установщика

Если настройки в браузере выставлены по умолчанию, то файл скачается в папку Загрузки (C:\Users\имя_пользователя\Downloads). Если же в браузере выставлен режим **Всегда указывать место для скачивания**, то откроется стандартное окно для выбора папки. Вы можете выбрать любую папку.

Установка Code::Blocks в Windows

Скачанный файл является самораспаковывающимся архивом-установщиком. Его надо запустить на выполнение. Появится окно приветствия (рис. П3.5).

Нужно нажать кнопку **Next** (Следующее). Появится окно с лицензионным соглашением (рис. П3.6).

Нужно нажать кнопку **I Agree** (Я согласен). На экране появится окно для выбора компонентов среды (рис. П3.7).

Мы пока никаких галочек отменять не будем, и по умолчанию установится все, что есть в составе системы. Нажмем кнопку **Next**. Появится окно для выбора папки, в которую вы хотите установить Code::Blocks. При этом предлагается установить систему в стандартную папку (рис. П3.8).



Рис. П3.5. Окно приветствия установщика

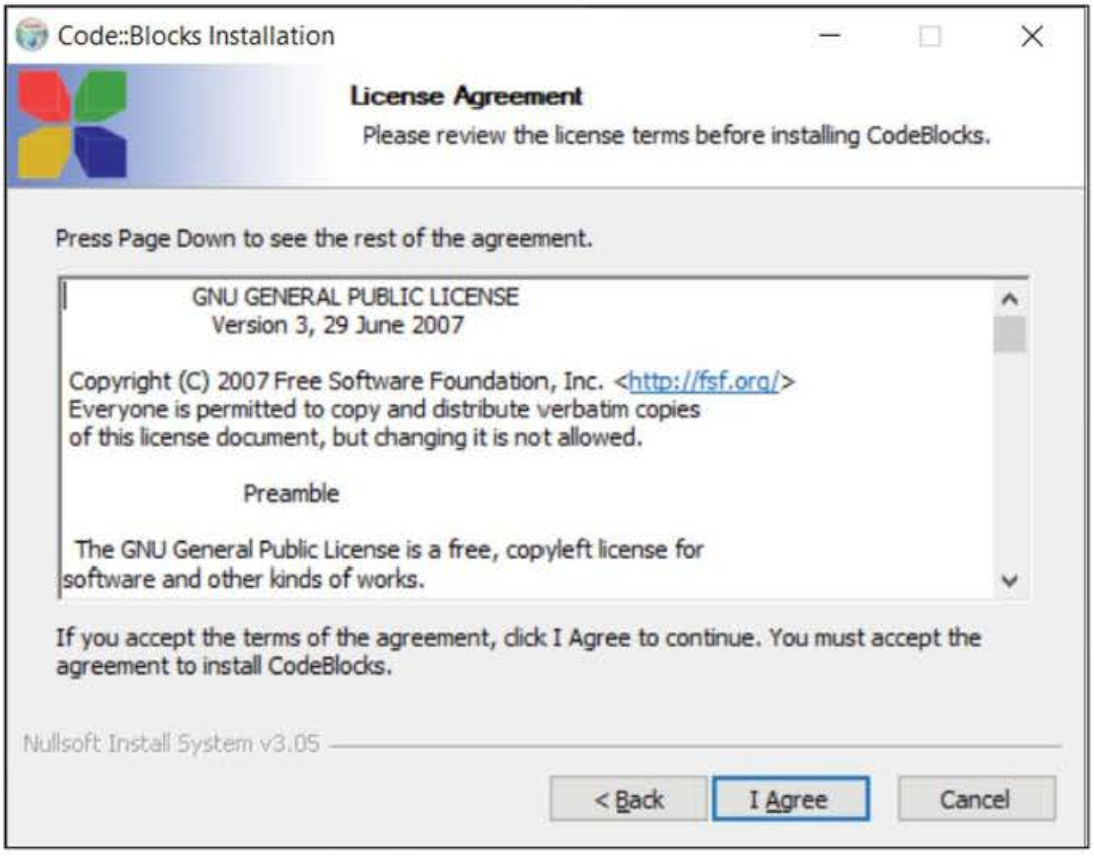


Рис. П3.6. Окно лицензионного соглашения

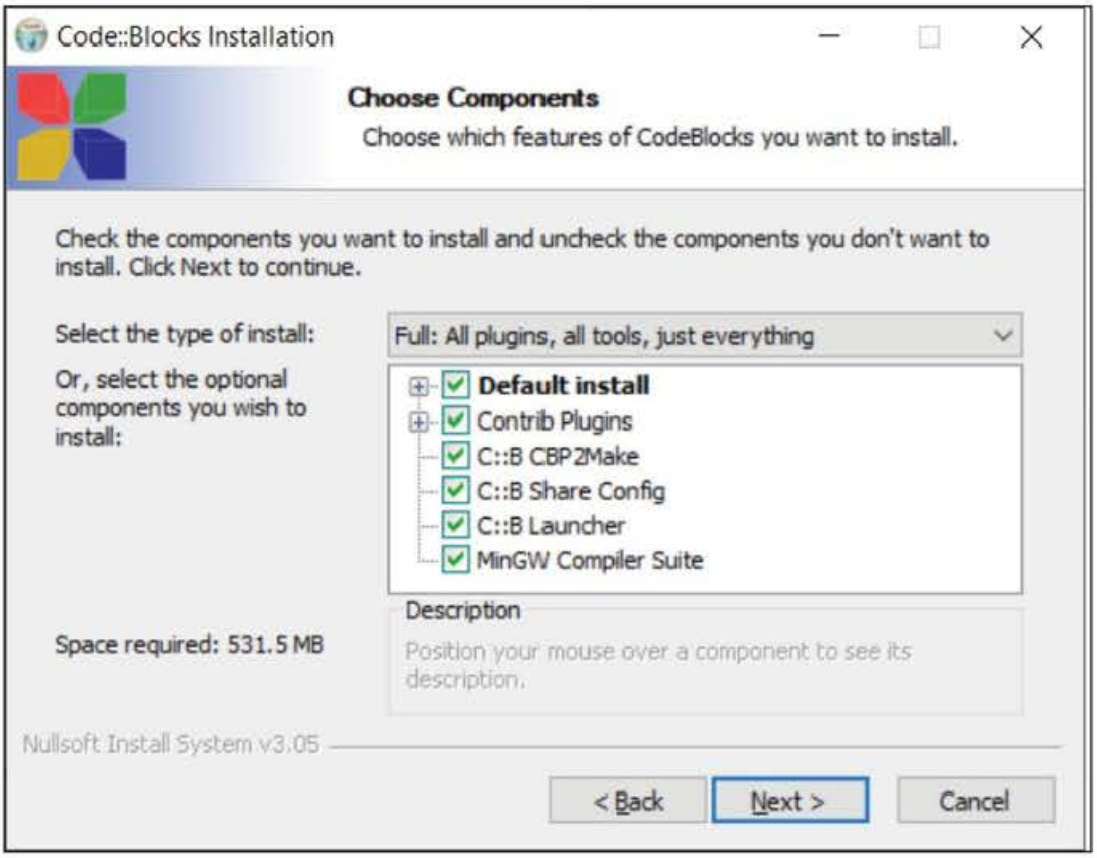


Рис. П3.7. Окно выбора компонентов среды

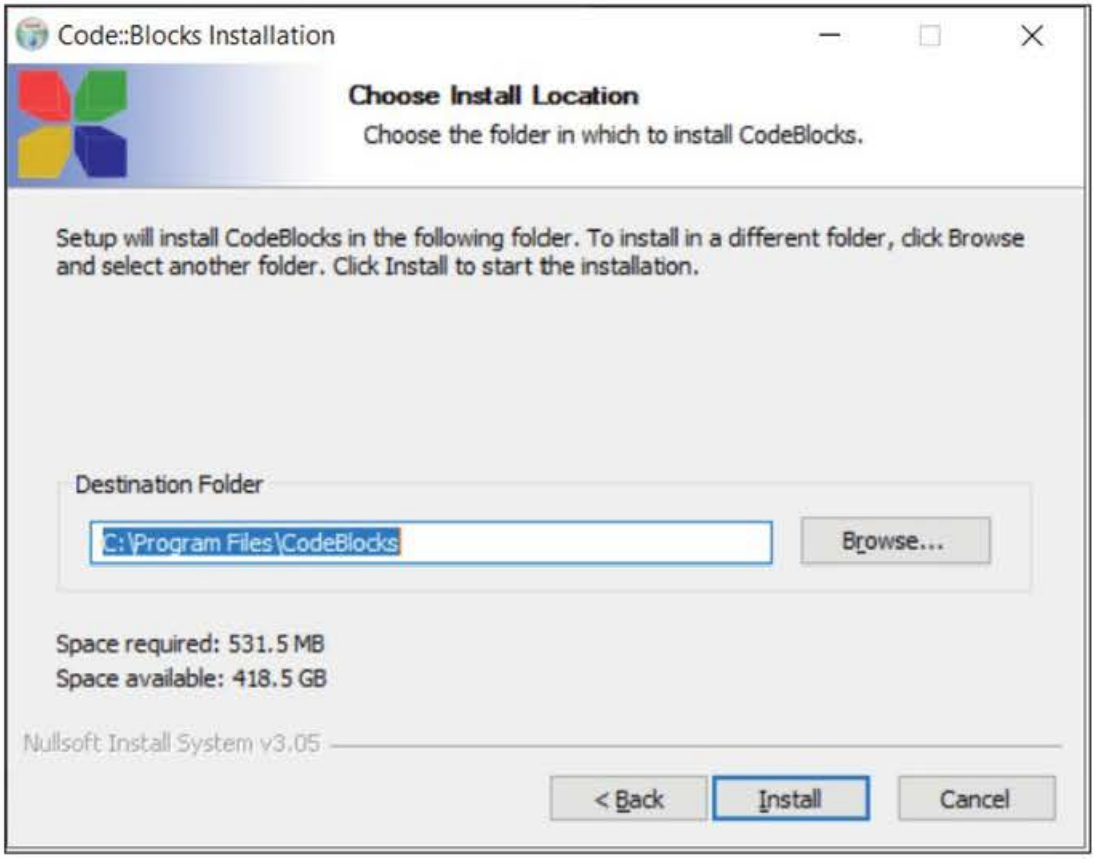


Рис. П3.8. Окно выбора папки для установки системы

Нажимаем кнопку **Install** (Установить). После этого на рабочем столе будет создан ярлык для запуска и в стандартное меню Windows пропишется папка CodeBlocks, откуда тоже можно будет запускать среду.

Установка Code::Blocks в «Альт Образование»

В операционной системе «Альт Образование», которая собрана на основе российского дистрибутива Alt Linux, устанавливать Code::Blocks не нужно, среда там установлена по умолчанию.

Установка Code::Blocks в МОС

В операционной системе МОС, которая собрана на основе российского дистрибутива РОСА Linux, устанавливать Code::Blocks не нужно, среда там установлена по умолчанию.

- Приложение 4 -

Интегрированная среда Code::Blocks

Интегрированная среда Code::Blocks создана еще в 2008 году как свободно распространяемый проект с открытым исходным кодом. Номер версии 20.03 говорит о том, что эта версия была выпущена в марте 2020 года. На стартовой панели видно конкретную дату — 2020-03-12.

Запуск IDE и выход

Запуск IDE осуществляется стандартным для вашей операционной системы способом. Главное окно программы имеет вид, показанный на рис. П4.1.

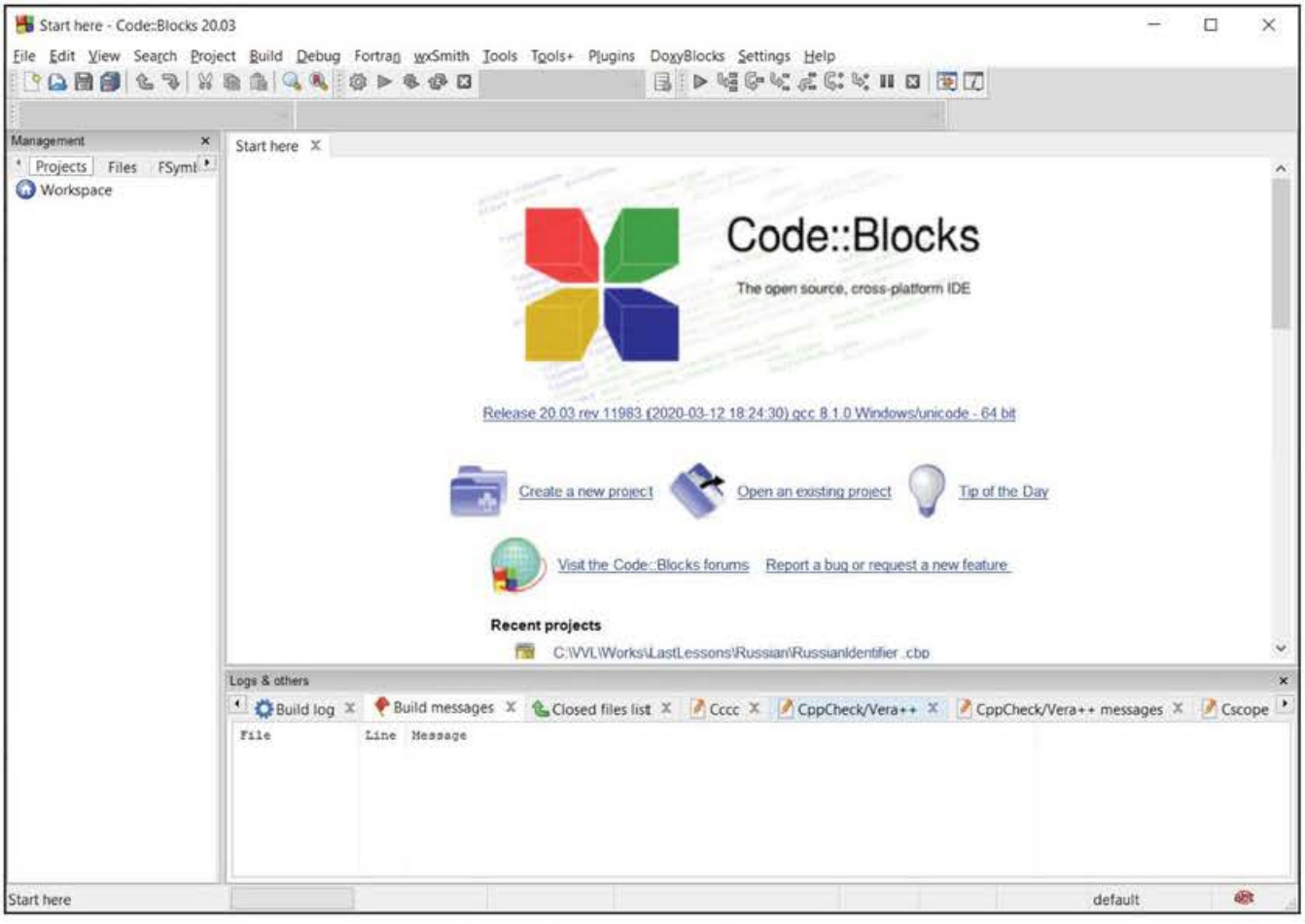


Рис. П4.1. Окно Code::Blocks

Как обычно, сверху (под заголовком окна) расположена строка главного меню. Под ней — панели инструментов (кнопки). Слева — окно проектов, справа — окно редактора кода. Внизу под ним — окно сообщений компилятора (вкладки **Build log** и **Build messages**).

В главном меню много пунктов, но мы будем работать только с некоторыми:

- **File** — работа с файлами и проектами;
- **Edit** — работа с редактором и с буфером обмена (clipboard);
- **View** — настройки внешнего вида Code::Blocks;
- **Search** — поиск и замена (в исходном коде);
- **Project** — работа с проектом;
- **Build** — компиляция и выполнение;
- **Settings** — настройки среды, редактора и компилятора.

Почти все действия в среде, кроме работы с исходным кодом в редакторе, можно выполнять с помощью мыши. Однако для многих операций в среде существуют «горячие клавиши».

Переход в меню выполняется клавишей <F10> (или клавишей <Alt>, если мы не работаем с кодом в редакторе). Для перехода в конкретный пункт главного горизонтального меню надо нажать <Alt>+<буква>. Буква — это подчеркнутая буква пункта главного меню. Например, для перехода в меню **File** нужно нажать <Alt>+<F>.

Переход в конкретный пункт выпадающего вертикального меню выполняется аналогично: нажимается подчеркнутая буква. Например, для перехода в пункт меню **File -> Save file as** надо нажать <Alt>+<F>+<A>.

> ***На заметку***
>
> Переходы по пунктам меню с помощью клавиши <Alt> работают только в английской раскладке.

Для некоторых пунктов меню назначены специальные клавиши, которые уже много лет применяются для этих целей в самых разных программах. Эти клавиши прописаны непосредственно в пункте меню. Например, сохранить файл в любой момент можно, нажав клавиши <Ctrl>+<S>.

Завершение работы IDE и выход из программы — <Ctrl>+<Q> или <Alt>+<F>+<Q>.

Выключение ненужных панелей инструментов

По умолчанию при установке включены все панели инструментов (всего 10 панелей). Нам нужны только три: **Main** (Главная), **Code complete** (Завершение кода) и **Compiler** (Компилятор). Для выключения лишних панелей нужно перейти в меню **View -> Toolbars** и во вложенном меню убрать галочки со всех пунктов, кроме указанных.

Можно то же самое сделать в контекстном меню, открыв его щелчком правой кнопки мыши по свободной от панелей области в верхней части окна.

Создание и сохранение исходного кода

Интегрированная среда позволяет программисту работать либо с проектами, либо с отдельными файлами исходного кода. В обоих случаях нужно сначала создать каталог (папку), в которой будут храниться все проекты и все файлы с исходным кодом программ. Это нужно сделать до начала работы в IDE средствами операционной системы, установленной на вашем компьютере.

> ***На заметку***
>
> Не рекомендуется в названиях каталогов и файлов использовать кириллицу.

Проекты обычно создаются в том случае, если программа состоит из многих файлов. Но мы будем их создавать и для программ из одного файла.

Создание файла

Для создания отдельного файла с исходным кодом нужно перейти в меню **File->New->File…** (именно так — с многоточием). Появится окно выбора типа файла (рис. П4.2).

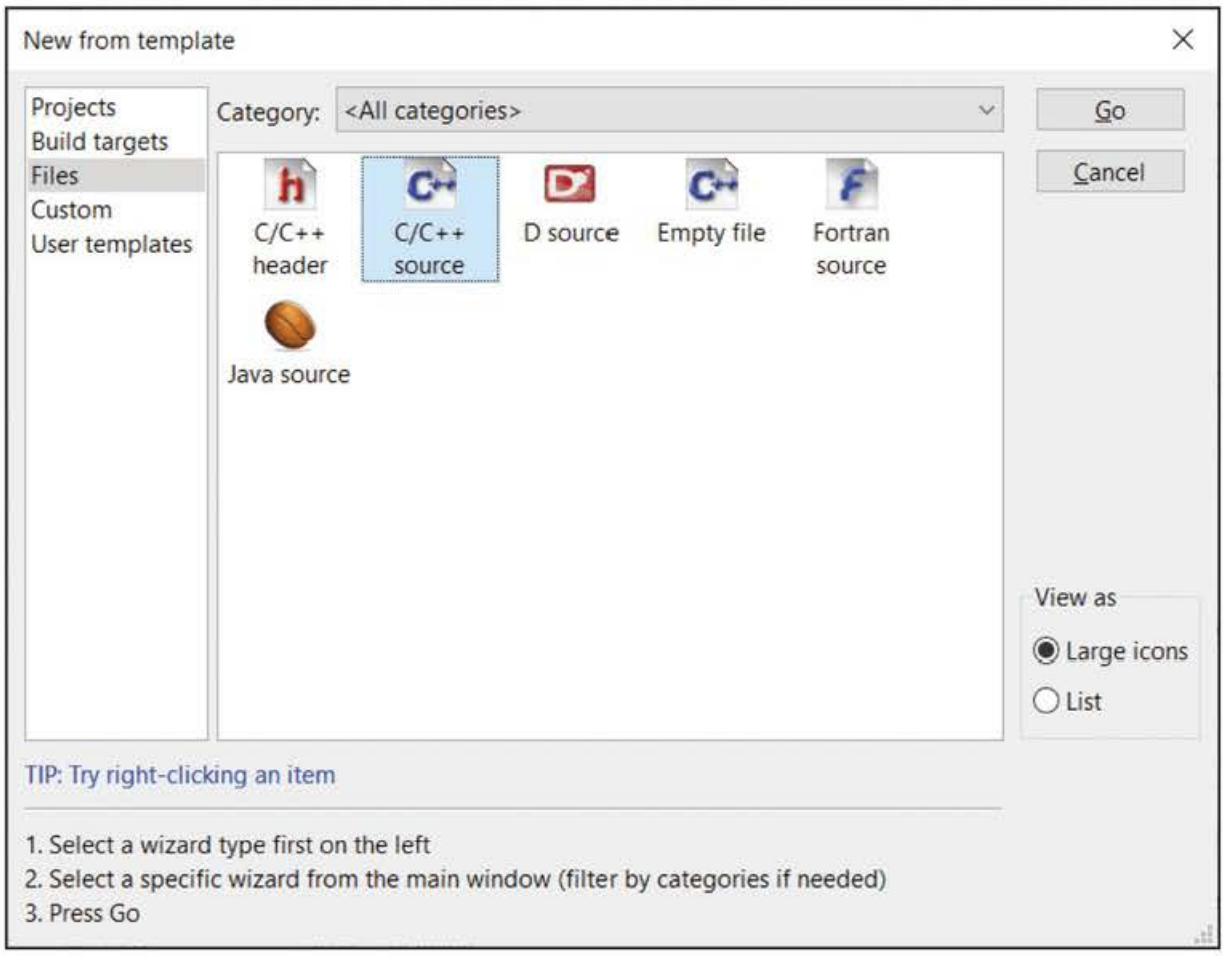


Рис. П4.2. Окно выбора типа файла



Рис. П4.3. Окно выбора языка

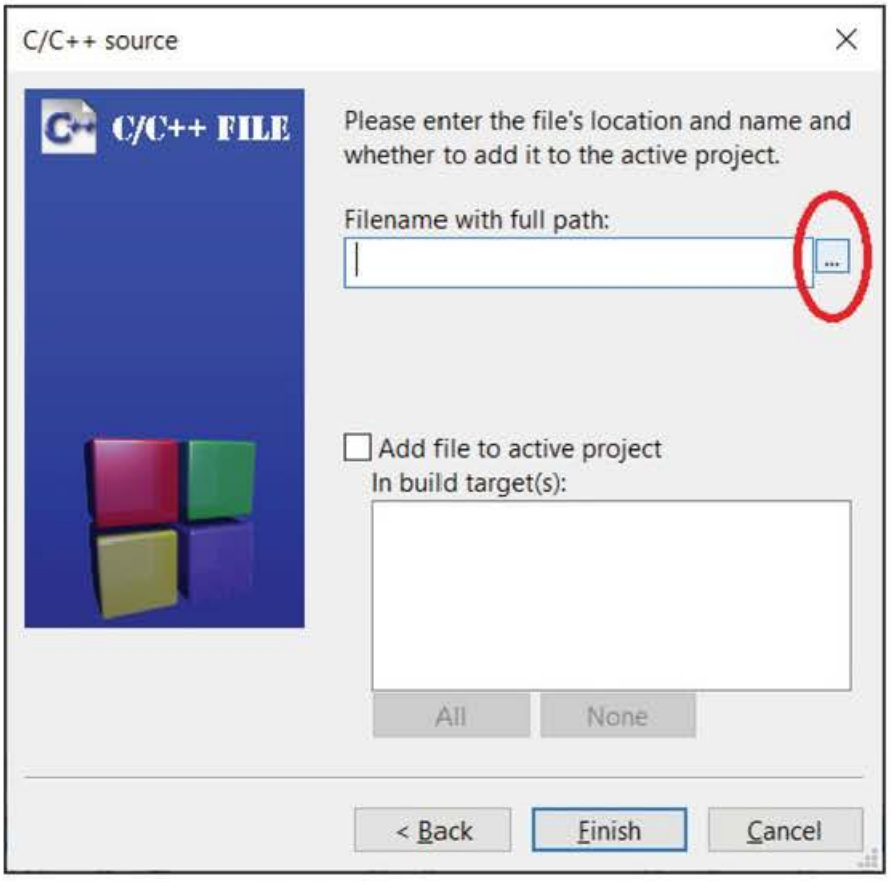


Рис. П4.4. Окно размещения файла

Выбираем иконку с надписью **C/C++ source** и нажимаем кнопку **Go**. На экране появится окно выбора языка (рис. П4.3). Надо выбрать язык C++ и нажать кнопку **Next**.

После этого появится окно, в котором предлагается выбрать место для сохранения создаваемого файла (рис. П4.4).

В этом окне нужно нажать кнопку с многоточием (на рисунке — в красном овале) и выбрать каталог (папку), где будет сохранен создаваемый файл. В окне выбора каталога (папки) нужно будет задать имя файла (например, `First`) без расширения. После этого нажать кнопку **Finish**, и в окне редактора появится пустая вкладка с именем файла, в которой надо будет набрать код. После набора кода файл надо опять сохранить (нажать клавиши <Ctrl>+<S>).

Создание проекта

Для создания проекта нужно перейти в меню **File->New->Project**. Появится окно выбора типа проекта (рис. П4.5).

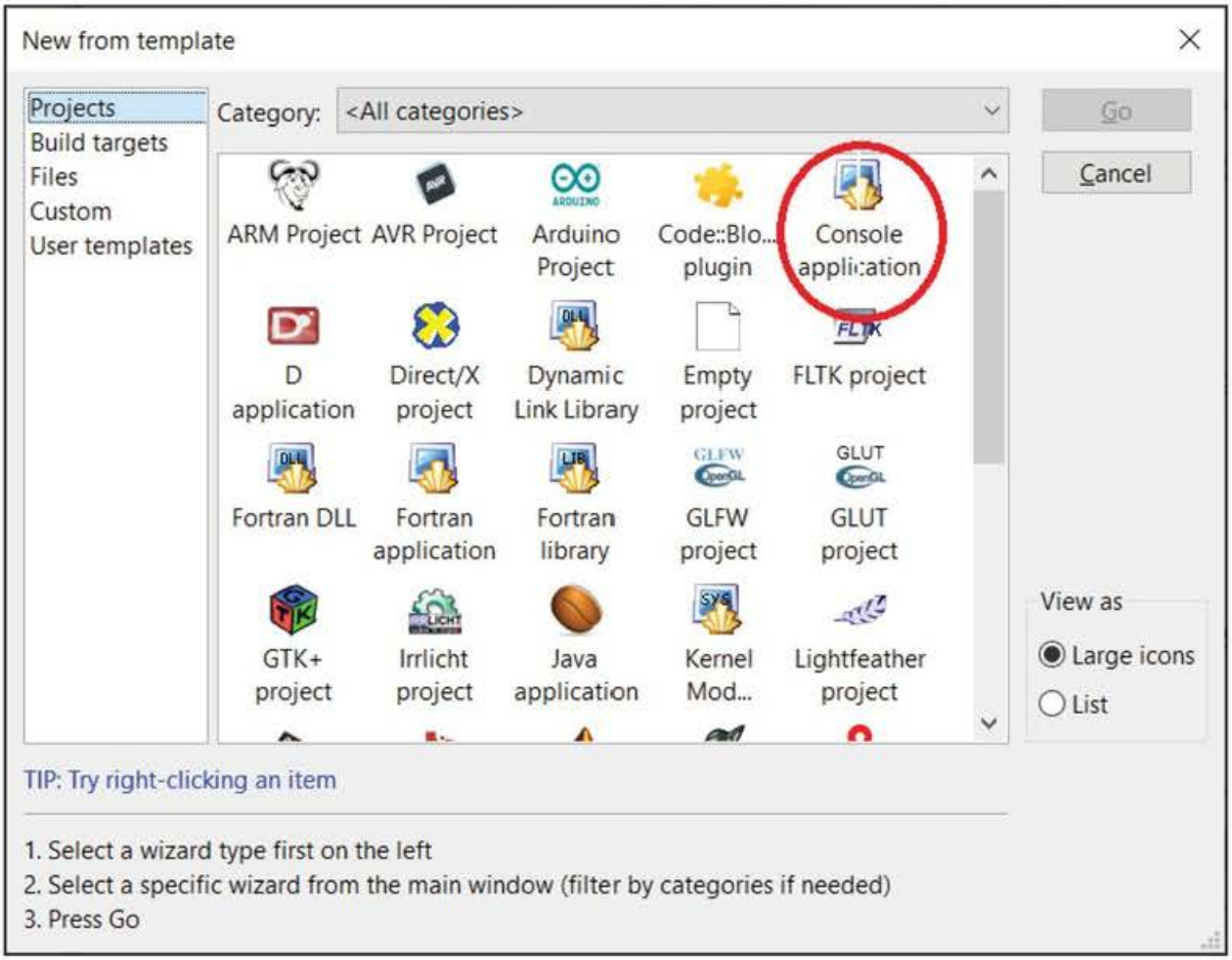


Рис. П4.5. Окно выбора типа проекта

Надо выбрать тип **Console application** и нажать кнопку **Go**. Появится окно для выбора языка (см. рис. П4.3). Выбираем **C++** и нажимаем кнопку **Next**. Появляется

окно для размещения проекта (рис. П4.6). В нем выбирается каталог (папка) — место размещения проекта (на рисунке — кнопка в красном овале).

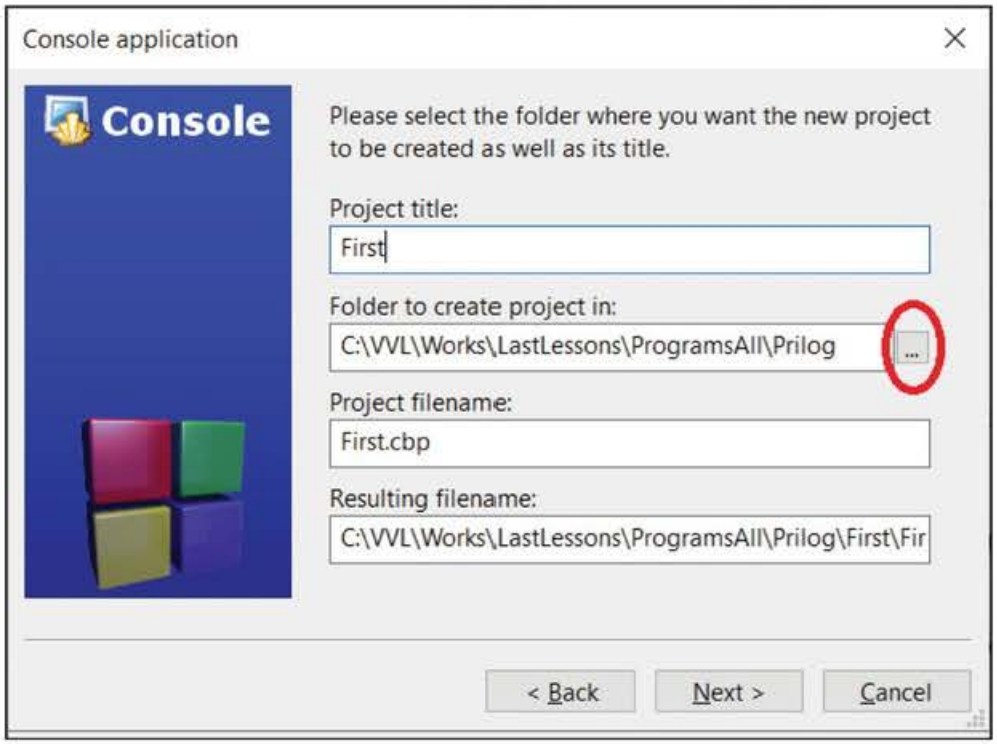


Рис. П4.6. Окно размещения проекта

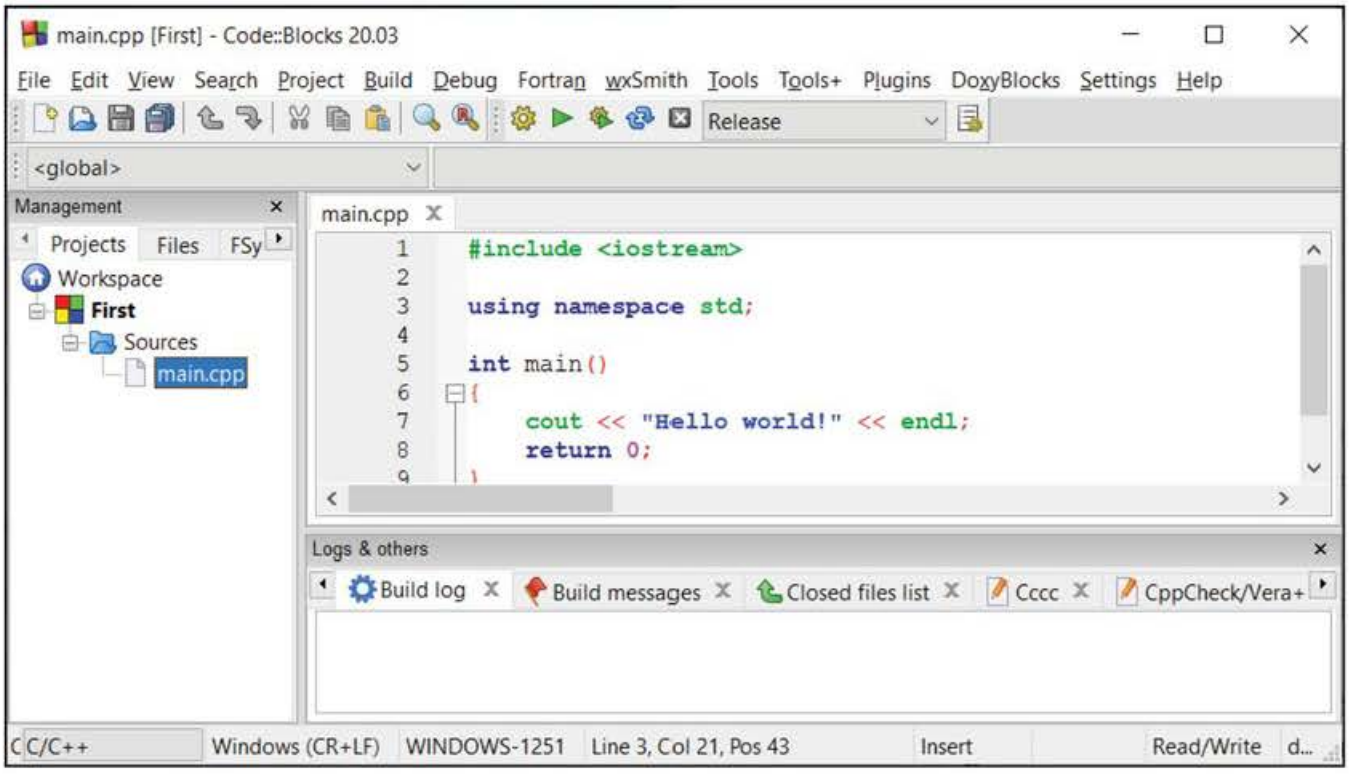


Рис. П4.7. Код проекта `First`

После выбора места нужно в верхней строке ввести имя проекта. Это же имя появится в третьей строке с расширением cbp. В четвертой строке мы видим, что среда

в выбранном нами каталоге создала каталог с именем проекта и внутри него файл с именем проекта.

Нажав кнопку **Next**, попадаем в финишное окно, где надо установить флажок **Create Release configuration** и нажать кнопку **Finish**.

После этого в левом вертикальном окне проектов появится наш проект `First`. Раскрыв его и щелкнув по имени файла **main.cpp**, увидим в окне редактора код программы «Hello World» (рис. П4.7).

Можно модифицировать код (см. *урок 2*). Нажмите клавиши <Ctrl>+<Shift>+<S> после модификации для сохранения всего (Save everything).

Можно оттранслировать и выполнить проект. Но сначала надо установить режимы компиляции (и сборки).

Установка режимов компиляции

Установка режимов компиляции делается в меню **Settings->Compiler**.

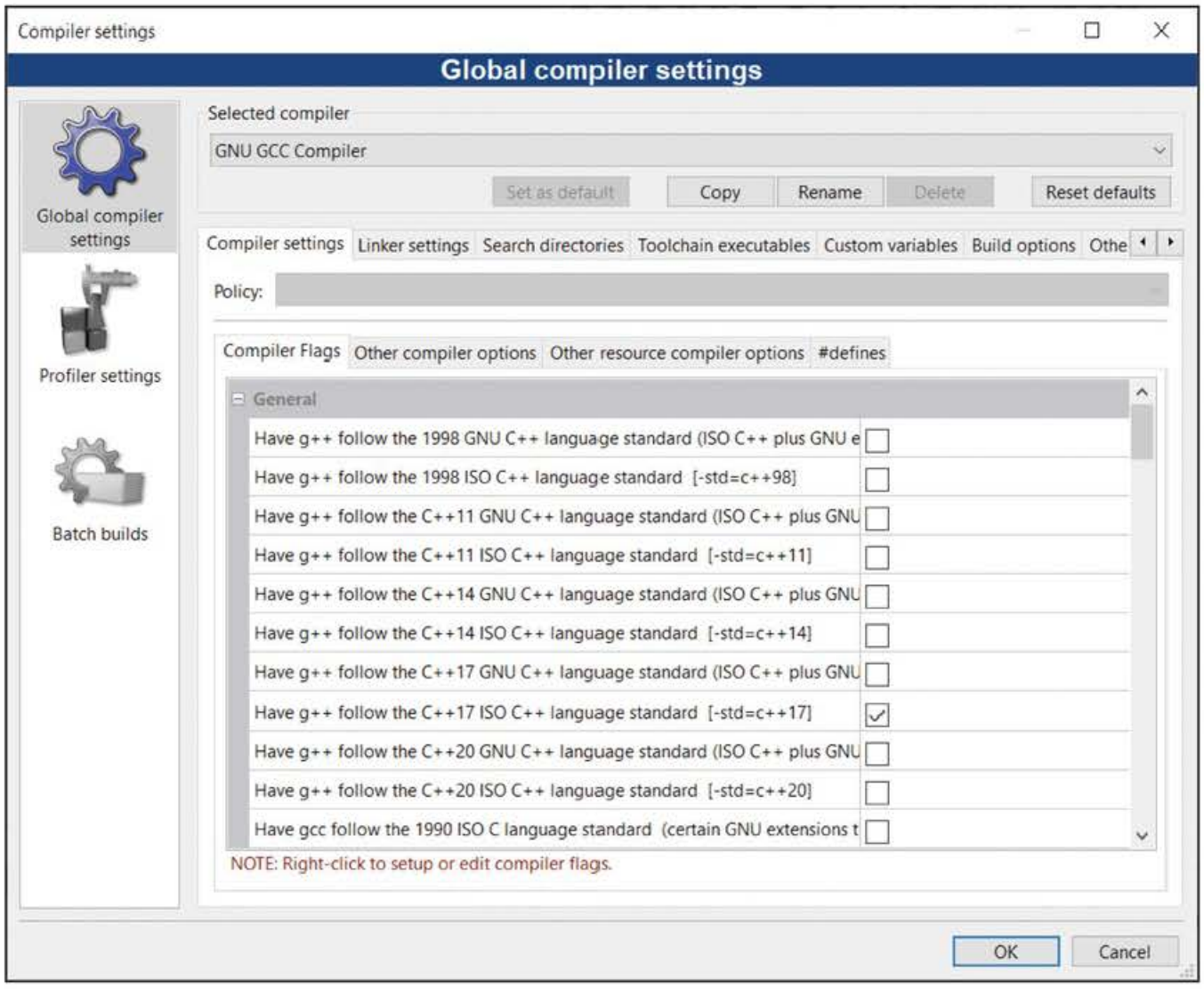


Рис. П4.8. Окно установки режимов компиляции

На экране появится окно **Compiler settings** (рис. П4.8), где надо проставить флажки компиляции и сборки:

- **Have g++ follow the C++17 ISO C++ language standard [-std=c++17]**;
- **Target x86_64 (64bit) [-m64]**;
- **Enable extra compiler warnings [-Wextra]**;
- **Enable warnings demanded by strict ISO C and ISO C++ [-pedantic]**.

Если вы работаете в Windows, можно дополнительно установить флажок **Static linking [-static]**.

При создании файла (не проекта) нужно еще установить флажок **Enable all common compiler warnings (overrides many other settings) [-Wall]**. При создании проекта этот флажок устанавливается автоматически.

Теперь можно запустить компиляцию (и сборку, если нет ошибок).

Компиляция и сборка

Чтобы откомпилировать проект и автоматически собрать (если не будет ошибок), нужно нажать клавиши <Alt>+<B>+<B> или <CTRL>+<F9>. Можно кликнуть по кнопке со значком шестеренки (см. рис. П4.7) в панели инструментов. Этот же значок шестеренки можно видеть в меню **Build**.

Результаты компиляции и сборки — в нижнем окне (под окном редактора) на вкладке **Build log** (рис. П4.9).

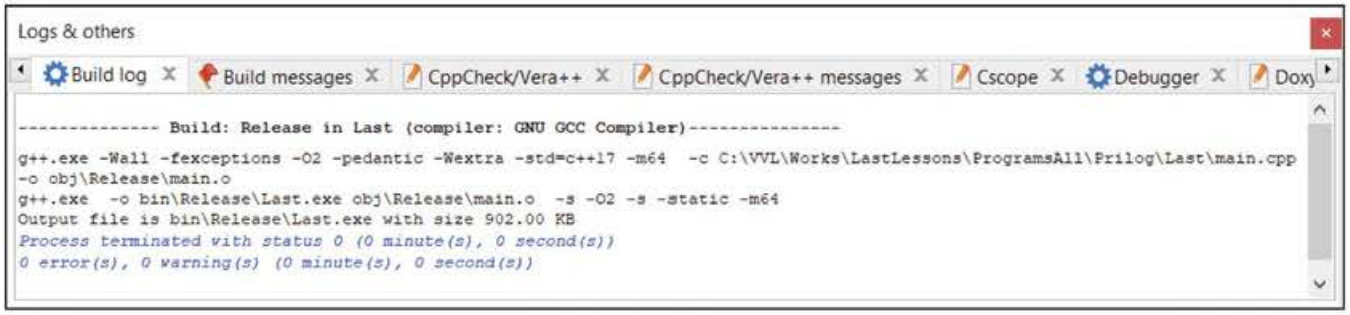


Рис. П4.9. Результаты компиляции и сборки без ошибок

Первая строка, начинающаяся с `g++.exe` — запуск компилятора с режимами компиляции. Вторая (тоже начинается с `g++.exe`) — это запуск линкера. После этого написано, в каком каталоге находится исполняемый файл. И синим цветом — статистика о компиляции и сборке.

Если надо просто откомпилировать отдельный файл (без сборки), то надо нажать клавиши <Ctrl>+<Shift>+<F9>.

Сообщения об ошибках

Если при компиляции возникли ошибки, то в том же нижнем окне на вкладке **Build message** можно прочесть сообщения об ошибках. Например, удалим или за-

комментим директиву `using` и запустим компиляцию. Результаты можно видеть на рис. П4.10.

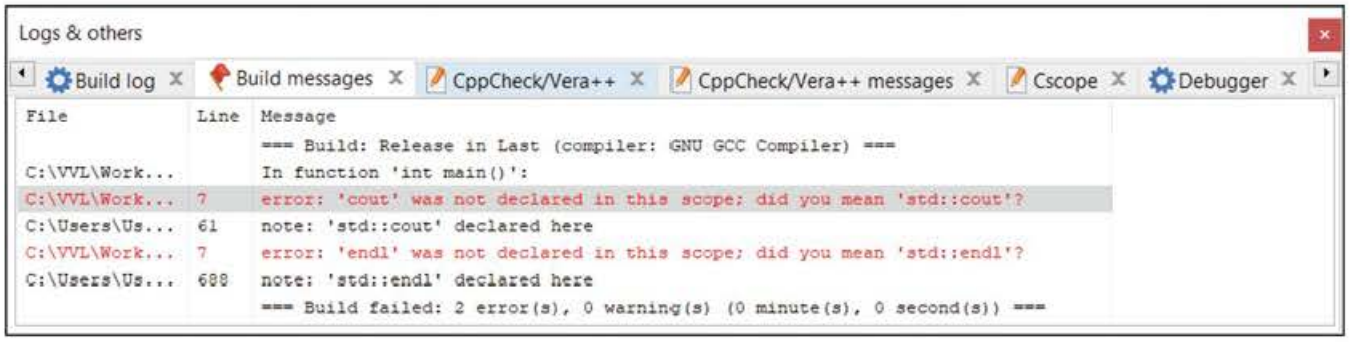


Рис. П4.10. Сообщения об ошибках при отсутствии директивы `using`

Сами сообщения в окне прописаны красным цветом.

Выполнение программы

Если при компиляции не было ошибок, то можно запустить программу. Для этого надо нажать клавиши <Alt>+<B>+<R> или <Ctrl>+<F10>. Можно щелкнуть по кнопке с зеленым треугольником (см. рис. П4.7) в панели инструментов. Этот же значок треугольника можно видеть в меню **Run**.

Интегрированная среда сама откроет консольное окно (терминал), в котором появятся результаты выполнения программы (см. рис. 1.2 и 1.3 в *уроке 1*).

Предметный указатель

A

API, 115

C

Catch-блок, 266
Cp-1251, 222

E

Escape-последовательность, 35

G

GCC, 19

I

IDE, 22, 23
IEEE-754, 61

J

Java, 235

K

KDE, 21

L

L-value, 68

O

ODR, 337, 339

P

Python, 235

R

R-value, 68

S

Scope, 110
STL, 26, 47, 62, 93, 115, 171, 172, 201, 204, 216, 219, 251, 263, 291, 312, 318, 341

T

Try-блок, 266

U

UB, 209, 262, 302
Using-директива, 26, 32, 42, 58
Using-объявление, 40, 74, 143
UTF-8, 222

А

Адрес, 189
Алгоритм, 16, 216
◊ типовой, 165
◊ Евклида, 136, 238
Алиас, 74
Алфавит, 32
Аргумент, 115

Б

Библиотека стандартная, 26, 32
Бит, 275, 282
Блок, 28, 93, 116, 119
Булева алгебра, 81

В

Вектор, 309
Ветвление, 18
Вихрь Мерсенна, 139
Время жизни, 129
Выравнивание, 177, 178

Д

Датчик, 138, 218
Декомпозиция, 175
Делегирование, 313
Деление по модулю, 66
Деструктор, 304
Диапазон, 61
Динамическая память, 231
Директива, 26, 32

Е

Единица трансляции, 337

З

Закон
◊ двойного отрицания, 83
◊ де Моргана, 83
Захват, 298

И

Идентификатор, 43
Инициализация, 44
◊ агрегатная, 156
◊ конструктор, 184
◊ универсальная, 57
Инкапсуляция, 129, 183, 296, 339
Инстанцирование, 214
Интродуктор, 295
Исполнитель, 16
Итератор, 201, 292
◊ обратный, 203
◊ потоковый, 204
◊ прямой, 203

К

Класс, 182
Класс-шаблон, 62
Ключевое слово, 33
Код
◊ исходный, 16, 17
◊ объектный, 17, 20
◊ возврата, 117
Кодировка
◊ cp-1251, 37, 38
◊ cp-866, 37
◊ UTF-8, 37
Командный процессор, 38
Комментарий, 30, 31
◊ многострочный, 30
◊ строчный, 30
Компилятор, 19, 119
◊ ошибка, 20, 111
◊ предупреждение, 20
Компиляция, 17
Композиция, 173
Компоновщик, 20
Конкатенация, 47
Константа, 29, 65
◊ глобальная, 121
◊ поименованная, 47, 56
Конструктор, 184
◊ без аргументов, 185
◊ инициализации, 184
◊ копирования, 185
◊ по умолчанию, 185
Контейнер, 201, 309
Косвенное обращение, 188, 189

Л

Линкер, 20

Литерал, 29, 35, 51, 65, 194
◊ R-литерал, 36, 37
Лямбда, 294

M

Макрос, 135, 194
Манипулятор, 29, 55, 92, 106, 107, 273
Мантисса, 52, 61
Массив, 156, 179
◊ индекс, 158
◊ конечное значение, 200
◊ начальное значение, 200
◊ одномерный, 156
Метка, 98
Метод, 186
Механизм обработки исключений, 263
Многочлен, 279
Модель памяти, 188
Модуль, 149, 332

H

Неопределенное поведение, 209, 225, 262, 302
НОД, 238, 240
Нуль-байт, 227

O

Область видимости, 128
Объект-исключение, 263
Оператор, 28
◊ возврата, 29, 117
◊ вывода, 28, 33
◊ инкремента, 102
◊ объявления, 44, 71
◊ переключатель, 146
◊ перехода, 98
◊ пустой, 109
◊ составной, 87, 93
◊ условный, 86
Операция
◊ бинарная, 66, 81, 84
◊ запятая, 84, 109
◊ левоассоциативная, 66, 84
◊ получения адреса, 190
◊ правоассоциативная, 67
◊ присваивания, 67
◊ разыменования, 190
◊ с присваиванием, 68
◊ унарная, 68, 81
◊ условная, 83
Определенное поведение, 262

П

Палиндром, 77, 222
Память, 44
◊ период хранения, 301
Параметр
◊ фактический, 115, 127
◊ формальный, 115
Перегрузка, 47, 186, 203, 213, 252
◊ операций, 190, 299
◊ функций, 209
Переменная, 43
◊ глобальная, 93
◊ локальная, 93, 129
◊ статическая, 131
Переполнение, 77, 94, 255
◊ порядка, 259
Погрешность, 281
Полином, 274, 277
Поток, 318
Потоковый ввод/вывод, 26
Препроцессор, 32, 344
Прототип, 56, 115, 121
Псевдокод, 17, 18

P

Размер, 61
Реверс, 224
Рекуррентная формула, 238, 239
Рекурсия, 239, 243
◊ глубина, 244, 249
◊ косвенная, 239
◊ прямая, 239
Рефакторинг, 34, 120

C

Связывание, 338
Синоним, 74
Система счисления, 273
◊ позиционная, 274
Система типов, 172

Си-строка, 228, 241
Стандартное пространство имен, 26
Стандартные алгоритмы, 292
Стандартный заголовок, 26
Стек, 129, 243, 249, 302
◊ переполнение, 244
Страж включения, 345
Структура, 172, 178, 179, 182
◊ поле, 172, 180
◊ пустая, 173
◊ размер, 177
◊ тег, 172, 180
◊ точка объявления, 173
Суффикс, 61, 65
Схема Горнера, 277

Т

Таблица, 313
Тестирование, 22, 133
Тест-кейс, 133, 235, 237
Точность, 61
Трассировка, 247, 249

У

Указатель, 189
◊ бестиповый, 192, 194

Ф

Факториал, 112
Формула Бине, 240
Формула Герона, 96
Функтор общности, 239
Функциональные объекты, 293
Функция, 27, 114
◊ главная, 27
◊ заголовок, 27
◊ имя, 115
◊ итеративная, 240
◊ объявление, 115
◊ отладочная, 247
◊ параметры, 27
◊ перегрузка, 131, 142
◊ проектирование, 118
◊ рекурсивная, 239
◊ стандартная, 115
◊ тело, 28, 116, 119
Функция-предикат, 123

Ц

Цикл, 99
◊ итерация, 100
◊ со счетчиком, 104
◊ счетчик, 101
◊ тело, 100
◊ условие продолжения, 100
◊ шаг, 100

Ш

Шаблон, 214, 215

Э

Экспонента, 52
Эпсилон, 269