

Марина Полубенцева

C/C++

**ПРОЦЕДУРНОЕ
ПРОГРАММИРОВАНИЕ**

Санкт-Петербург

«БХВ-Петербург»

2008

УДК 681.3.068+800.92C/C++
ББК 32.973.26-018.1
П53

Полубенцева М. И.

П53 C/C++. Процедурное программирование. — СПб.: БХВ-Петербург, 2008. — 448 с.: ил. — (Внесерийная)
ISBN 978-5-9775-0145-3

Подробно рассмотрены процедурные возможности языков программирования C/C++. Изложены основные принципы строения программы на языке C/C++: раздельная компиляция, функциональная декомпозиция, блоки кода. Описаны синтаксические конструкции языка и показана специфика их использования. Подробно излагаются понятия, связанные с представлением данных: виды данных, их представление в тексте программы, размещение в памяти, время существования и области видимости. Описано назначение и принцип работы пре-процессора. Детально рассмотрены указатели и массивы, а также их взаимосвязь в языке C/C++. Приведена сравнительная характеристика ссылок C++ и указателей. Обсуждаются сложные программные элементы. Рассмотрены агрегатные пользовательские типы данных языка C: структуры, объединения.

Для программистов и разработчиков встраиваемых систем

УДК 681.3.068+800.92C/C++
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Игорь Шишигин</i>
Зав. редакцией	<i>Григорий Добин</i>
Редактор	<i>Римма Смоляк</i>
Компьютерная верстка	<i>Натали Каравановой</i>
Корректор	<i>Виктория Пиотровская</i>
Дизайн серии	<i>Инны Тачиной</i>
Оформление обложки	<i>Елены Беляевой</i>
Зав. производством	<i>Николай Тверских</i>

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 30.11.07.

Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 36,12.

Тираж 2000 экз. Заказ №

"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Санитарно-эпидемиологическое заключение на продукцию
№ 77.99.02.953.Д.006421.11.04 от 11.11.2004 г. выдано Федеральной службой
по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

Оглавление

- Введение..... 1**
- Предисловие2
- Особенности изложения3
- Благодарности5
- Глава 1. Общие принципы процедурного программирования..... 7**
- 1.1. О современном программировании в целом 7
 - 1.1.1. Историческая справка 7
 - 1.1.2. Этапы создания программного продукта.....8
 - 1.1.3. Памятка программисту 9
 - 1.1.4. Критерии хорошего программного продукта 11
- 1.2. Структура программы..... 11
 - 1.2.1. Разбиение на файлы (модульность) и связанные с этим понятия C/C++..... 13
 - 1.2.2. Функциональная декомпозиция и связанные с ней понятия..... 18
 - 1.2.3. Блоки кода 24
 - 1.2.4. Оформление текста программы. Комментарии и отступы..... 26
- Глава 2. Базовые понятия C/C++ 29**
- 2.1. Ключевые слова..... 29
- 2.2. Идентификаторы (имена) 29
- 2.3. Понятия *lvalue* и *rvalue* 30
- 2.4. Операторы..... 31
 - 2.4.1. Арифметические операторы..... 36
 - 2.4.2. Операторы присваивания..... 38
 - 2.4.3. Побитовые операторы 39
 - 2.4.4. Логические операторы и операторы отношения 43
 - 2.4.5. Тернарный оператор ?: 45
 - 2.4.6. Оператор ", " 47

Глава 3. Данные.....	49
3.1. Виды данных	49
3.2. Константы (литералы)	51
3.2.1. Целые литералы	52
3.2.2. Литералы с плавающей точкой	54
3.2.3. Символьные литералы	55
3.2.4. Строковые литералы	61
3.3. Перечисление <i>enum</i>	62
3.4. Переменные	65
3.4.1. Что такое тип переменной	66
3.4.2. Фундаментальные (базовые, встроенные) типы C/C++	67
3.4.3. Оператор <i>sizeof</i> и размеры переменных	69
3.4.4. Знаковость переменной	71
3.4.5. Приведение типов	73
3.4.6. Тип <i>wchar_t</i>	81
3.4.7. Тип <i>bool</i> и <i>BOOL</i>	81
3.5. Понятия объявления и определения	82
3.5.1. Объявление переменной	83
3.6. Способы использования переменных и типы компоновки	86
3.6.1. Безопасная сборка проекта (<i>type-safe linkage</i>)	88
3.7. Размещение и время существования переменных	89
3.7.1. Ключевое слово <i>static</i>	91
3.8. Область видимости переменной (<i>scope</i>)	94
3.8.1. Скрытие (замещение) имени переменной	95
3.8.2. Пространства имен — <i>namespace</i>	97
3.9. Инициализация переменных	108
3.9.1. Явная инициализация переменных (программистом)	108
3.9.2. Неявная инициализация переменных (компилятором)	108
3.10. Модификаторы <i>const</i> и <i>volatile</i>	109
3.10.1. Ключевое слово <i>const</i>	109
3.10.2. Ключевое слово <i>volatile</i>	110
Глава 4. Инструкции (<i>statements</i>) C/C++.....	113
4.1. Общая информация об инструкциях	113
4.2. Инструкции выбора (условия)	115
4.2.1. Инструкции <i>if</i> , <i>if...else</i>	115
4.2.2. Переключатель — <i>switch</i>	118
4.3. Инструкции цикла	122
4.3.1. Инструкция <i>while</i>	123
4.3.2. Инструкция <i>do...while</i>	127
4.3.3. Инструкция <i>for</i>	129
4.4. Инструкции безусловного перехода: <i>break</i> , <i>continue</i> , <i>return</i> , <i>goto</i>	134

Глава 5. Препроцессор. Заголовочные файлы	137
5.1. Директивы препроцессора	137
5.2. Директива <i>#define</i>	138
5.2.1. Использование директивы <i>#define</i>	139
5.2.2. Предопределенные макросы	143
5.2.3. Диагностический макрос <i>assert</i>	144
5.2.4. Рекомендации	145
5.3. Директива <i>#undef</i>	145
5.4. Директивы <i>#ifdef</i> , <i>#ifndef</i> , <i>#else</i> и <i>#endif</i>	147
5.5. Директивы <i>#if</i> , <i>#elif</i> , <i>#else</i> , <i>#endif</i> . Оператор препроцессора <i>defined</i>	149
5.6. Директива <i>#include</i> . Заголовочные файлы	152
5.6.1. Концепция разделения на интерфейс и реализацию. Механизм подключения заголовочных файлов	153
5.6.2. Формы директивы <i>#include</i>	156
5.6.3. Вложенные включения заголовочных файлов (стратегии включения)	157
5.6.4. Предкомпиляция заголовочных файлов	159
5.6.5. Заголовочные файлы стандартной библиотеки	162
5.6.6. Защита от повторных включений заголовочных файлов	166
5.6.7. Что может быть в заголовочных файлах и чего там быть не должно	167
5.7. Директива <i>#pragma</i>	170
5.8. Директива <i>#error</i>	171
Глава 6. Указатели и массивы	173
6.1. Указатели	173
6.1.1. Объявление и определение переменной-указателя	175
6.1.2. Инициализация указателя и оператор получения адреса объекта — <i>&</i>	177
6.1.3. Получение значения <i>объекта</i> посредством указателя: оператор разыменования — <i>*</i>	179
6.1.4. Арифметика указателей	180
6.1.5. Указатель типа <i>void*</i>	182
6.1.6. Нулевой указатель (<i>NULL-pointer</i>)	184
6.1.7. Указатель на указатель	186
6.1.8. Указатель и ключевые слова <i>const</i> и <i>volatile</i>	187
6.1.9. Явное преобразование типа указателя	192
6.2. Массивы	196
6.2.1. Объявление массива	196
6.2.2. Обращение к элементу массива — оператор <i>[]</i>	198
6.2.3. Инициализация массива	200
6.2.4. Массивы и оператор <i>sizeof</i>	206

6.3. Связь массивов и указателей.....	207
6.3.1. Одномерные массивы.....	207
6.3.2. Двухмерные массивы более подробно	210
6.3.3. Многомерные массивы	214
6.3.4. Массивы указателей	216
6.4. Динамические массивы	218
6.4.1. Управление памятью. Низкоуровневые функции языка Си.....	219
6.4.2. Управление памятью. Операторы C++ <i>new</i> и <i>delete</i>	222
6.4.3. Сборщик мусора (<i>garbage collector</i>)	225
6.4.4. Операторы <i>new[]</i> и <i>delete[]</i> и массивы	225
6.4.5. Инициализация динамических массивов	234
Глава 7. Ссылки	235
7.1. Понятие ссылки	235
7.2. Сравнение ссылок и указателей.....	236
Глава 8. Функции.....	241
8.1. Понятия, связанные с функциями.....	241
8.1.1. Объявление (прототип) функции	244
8.1.2. Определение функции (реализация)	246
8.1.3. Вызов функции	248
8.1.4. Вызов <i>inline</i> -функции	252
8.1.5. Соглашения о вызове функции	254
8.2. Способы передачи параметров в функцию	259
8.2.1. Передача параметров по значению (<i>Call-By-Value</i>)	259
8.2.2. Передача параметров по адресу	260
8.2.3. Специфика передачи параметров.....	264
8.2.4. Переменное число параметров.....	272
8.3. Возвращаемое значение.....	287
8.3.1. Виды возвращаемых значений и механизмы их формирования.....	287
8.3.2. Проблемы при возвращении ссылки или указателя.....	290
8.4. Ключевое слово <i>const</i> и функции	292
8.4.1. Передача функции константных параметров	293
8.4.2. Возвращение функцией константных значений.....	294
8.5. Перегрузка имен функций.....	295
8.5.1. Возможные конфликты при использовании параметров по умолчанию.....	298
8.6. Рекурсивные функции	298
8.7. Указатель на функцию.....	301
8.7.1. Определение указателя на функцию.....	301
8.7.2. Инициализация указателя на функцию	302

8.7.3. Вызов функции посредством указателя	302
8.7.4. Использование указателей на функции в качестве параметров.....	303
8.7.5. Использование указателя на функцию в качестве возвращаемого значения	305
8.7.6. Массивы указателей на функции	306
8.8. Ключевое слово <i>typedef</i> и сложные указатели	308
8.8.1. Ключевое слово <i>typedef</i> и указатели на функции	308
8.8.2. Функции, возвращающие сложные указатели.....	308
Глава 9. Структуры	313
9.1. Зачем нужны структуры	313
9.2. Объявление структуры	314
9.3. Создание экземпляров структуры и присваивание значений полям структуры	316
9.4. Ключевое слово <i>typedef</i> и структуры	318
9.5. Совмещение объявления и определения. Анонимные структуры	319
9.6. Инициализация структурных переменных	320
9.7. Действия со структурами	321
9.8. Поля структуры пользовательского типа.....	322
9.9. Вложенные (<i>nested</i>) структуры	323
9.10. Указатели и структуры	324
9.11. Упаковка полей структуры компилятором. Оператор <i>sizeof</i> применительно к структурам	326
9.12. Структуры и функции	329
9.12.1. Передача структуры в функцию в качестве параметра	329
9.12.2. Возврат структуры по значению	332
9.13. Что можно использовать в качестве поля структуры	333
9.14. Поля битов	334
Глава 10. Объединения (<i>union</i>)	345
10.1. Понятие объединения	345
10.2. Использование объединений.....	346
10.3. Размер объединения.....	349
10.4. Инициализация объединений.....	350
10.5. Анонимные объединения (специфика Microsoft)	351
ПРИЛОЖЕНИЯ	353
Приложение 1. Представление данных.....	355
П1.1. О системах счисления и изображении количеств	355
П1.2. Перевод чисел из одной системы счисления в другую	357

П1.3. Использование различных систем счисления при технической реализации средств цифровой вычислительной техники	361
П1.4. Особенности выполнения арифметических операций в ограниченной разрядной сетке.....	362
П1.5. Изображение знакопеременных величин.....	363
П1.6. Выявление переполнений при выполнении сложения и вычитания....	367
П1.7. Смена знака целого знакопеременного числа	370
П1.8. Действия с повышенной разрядностью	370
П1.9. Особенности умножения и деления целых двоичных чисел	371
П1.10. Приведение типов данных.....	372
П1.11. Числа с плавающей точкой	375
П1.11.1. Неоднозначность представления и нормализованная форма.....	376
П1.11.2. Форматы представления чисел ПТ двоичным кодом	380
П1.11.3. Стандарт на числа ПТ ANSI/IEEE 754-1985	381
П1.12. О понятии старшинства арифметических типов данных	384
П1.13. Битовые поля и операции над ними	385
П1.13.1. Подробнее об операциях сдвига.....	390
Приложение 2. Язык Си и низкоуровневое программирование	392
П2.1. Низкоуровневая (регистровая) модель вычислительного ядра	398
П2.1.1. Оптимизация фрагмента кода по скорости	399
П2.1.2. Определение положения программы в пространстве адресов.....	401
П2.1.3. Использование средств уровня языка Ассемблера в программах на Си	401
П2.1.4. Работа с регистрами периферийных устройств.....	404
П2.1.5. Синхронизация программы с внешним событием	406
П2.2. Программирование обработчиков прерываний.....	408
П2.2.1. Запрет/разрешение прерываний процессору	409
П2.2.2. Приоритеты и управление ими.....	410
П2.3. Программирование без операционной системы.....	412
Предметный указатель	415

Введение

— Сынок, будешь хорошо учиться — купим тебе компьютер.

— А если буду плохо учиться?

— Тогда купим пианино.

Я хочу стать программистом, когда вырасту большим, потому что это классная работа и простая. Поэтому в наше время столько программистов и все время становится больше.

Программистам не нужно ходить в школу, им нужно учиться читать на компьютерном языке, что бы они могли с компьютером разговаривать. Думаю, что они должны уметь читать тоже, что бы знать в чем дело, когда все напереполах.

Программисты должны быть смелыми, чтобы не пугаться, когда все перепуталось так, что никто не разберет, или если придется разговаривать на английском языке по-иностранному, чтобы знать, что надо делать.

Еще мне нравится зарплата, которую программисты получают. Они получают столько денег, что не успевают их все тратить. Это происходит потому, что все считают работу программиста трудной, кроме программистов, которые знают, как это просто.

Нет ничего такого, что бы мне не понравилось, кроме того, что девочкам нравятся программисты и все хотят выйти за них замуж, и поэтому женщин надо гнать, чтобы не мешали работать.

Надеюсь, что у меня нет аллергии на офисную пыль, потому что на нашу собаку у меня аллергия. Если у меня будет аллергия на офисную пыль, программиста из меня не получится и придется искать настоящую работу.

Сочинение 7-летнего Тараса по теме: "Кем я хочу стать, когда я буду большим"

Предисловие

Вряд ли можно сказать что-либо новое о программировании на C/C++, тем более страшно подумать, сколько книг написано на эту тему, и какими авторами! В свое оправдание могу сказать лишь следующее: писать книги вообще (и эту в частности) мне бы никогда не пришло в голову, если бы мои студенты постоянно не упрекали меня в том, что они успевают одно из двух: или понимать, или конспектировать. Механическую часть работы я решила взять на себя, поэтому однажды летом, находясь в отпуске (эта ремарка для моего начальства), я села за компьютер...

Ну, а если серьезно, то хочу сказать, что материал этой книги является как обобщением накопленного личного практического опыта программирования на C++, так и результатом моей преподавательской деятельности.

Далеко не сразу мне удалось сформулировать причины, по которым, несмотря на огромное количество блестящих книг по этой теме, я все-таки рискнула написать еще одну.

За многие годы программирования на C/C++ я сама сталкивалась с многочисленными проблемами, на понимание и разрешение которых уходила уйма времени. Не хочется, чтобы это время пропало даром.

Концепция преподавания одного и того же предмета у каждого преподавателя своя. Я лично всегда пыталась понять и объяснить, в первую очередь, себе самой: *как и почему*. В книге постаралась структурировать ответы на эти вопросы. В частности, для понимания того, как некоторые свойства языка реализуются компилятором, иногда приходится разбираться с низкоуровневым кодом, в который компилятор превращает текст программы на C/C++, или анализировать, каким образом компилятор располагает данные в памяти. Без подобного понимания, конечно же, можно обойтись, но тогда остается просто заучивать правила. Однажды получила забавный отзыв на свой курс: "Три года программировал на Си, только теперь начал понимать, что делаю".

Большинство современных авторов книг, посвященных C++, довольно подробно описывают объектно-ориентированные возможности языка, уделяя минимум внимания базовым понятиям языка C/C++. Возможно, начинающий программист найдет в данном пособии структурированный материал по этой теме, а программист, уже имеющий опыт программирования на C/C++, откроет для себя возможности языка, которыми он до сих пор не пользовался (или пользовался неосознанно). Надеюсь, эта книга будет полезна разработчикам программного обеспечения для встраиваемых приме-

нений (в настоящее время программы для микроконтроллеров пишут в основном на языке Си).

Я попыталась написать книгу, в которой рассматриваются не только сами по себе основные понятия процедурного программирования на C/C++, но и взаимосвязи между этими понятиями. В программировании (как и в любой другой области) количество взаимосвязей между элементами существенно превышает количество самих элементов, а изложение, увы, линейно. Невозможно придумать способ, с помощью которого все можно было бы объяснить последовательно. Поэтому в книге много ссылок как вперед, так и назад. Это может даже несколько раздражать читателя, однако именно наличие таких ссылок позволяет быстрее преодолеть трудность начального этапа, когда изучающему еще почти ничего не известно, и поэтому мало что понятно. И, наоборот, бывает очень полезно вернуться по ссылке назад к тому материалу, который был не понят или пропущен. В результате в голове у учащегося формируется грубая (приближенная) модель внутренней структуры изучаемого предмета, а после этого существенно облегчается восприятие последующего материала, поскольку начинает активно работать ассоциативная память.

При чтении лекций я стараюсь пояснять материал разнообразнейшими рисунками, т. к. мне кажется, что таким образом материал воспринимается и запоминается лучше. Этот принцип изложения попыталась реализовать и в книге.

Особенности изложения

Данная книга содержит описание процедурных возможностей языков программирования Си и C++. Язык Си является чисто процедурным, а язык C++ был создан на базе Си, поэтому он совмещает традиционный процедурный подход с подходом объектно-ориентированным (в книге рассматриваются процедурные возможности одновременно для обоих языков).

При создании этих языков разработчики преследовали, прежде всего, цель повышения эффективности. Оба языка обладают низкоуровневыми возможностями, присущими языкам Ассемблера, и требуют от программиста большей ответственности (по сравнению с программированием на других языках высокого уровня).

В языке C++ появились понятия, которых не было в языке Си. Так как эти понятия (ссылки, перегрузка имен функций и т. д.) расширяют, в частности,

процедурные возможности C++, то они рассмотрены в данной книге наравне с другими базовыми понятиями. Также согласно стандарту языка C++ ISO/IEC 14882 "Standard for the C++ Programming Language" рассмотрены новые операторы явного приведения типа (`static_cast`, `reinterpret_cast`, `const_cast`) и пространства имен (`namespace`).

Языки C/C++ похожи, в первую очередь, тем, что являются синтаксически сложными, поэтому одна из целей данной книги — адаптация к нетривиальному (в отличие от других высокоуровневых языков) синтаксису, определяющему широкие возможности рассматриваемых языков.

Хочется обратить внимание читателя на некоторые моменты, прежде чем он перейдет к основным главам:

- ❑ большинство рассматриваемых в книге понятий справедливы как для языка Си, так и для языка C++. В тех случаях, когда существуют отличия, явно указывается, для какого языка эти понятия реализованы;
- ❑ термин "объект" используется в данной книге для обозначения любого низкоуровневого понятия C/C++ и не имеет отношения к объектно-ориентированному программированию;
- ❑ примеры низкоуровневого кода для пояснения действий компилятора приводятся для VC.net 2005;
- ❑ чтобы привлечь внимание читающего, некоторые существенные элементы в листингах подчеркнуты;
- ❑ по ходу изложения материала читателю предлагаются задания, над которыми он должен подумать самостоятельно, они сопровождаются пиктограммой

граммой  ;

- ❑ помимо основных 10 глав в книге имеются два приложения, в которых рассмотрены механизмы низкоуровневой реализации в цифровых процессорах многих элементов и конструкций языка Си. Знание этих механизмов необходимо при написании низкоуровневых драйверов аппаратных устройств, а также при оптимизации кода с целью повышения его эффективности (по быстродействию или объему требуемой памяти).

Благодарности

Не могу не выразить свою признательность:

- ❑ своему мужу, Новицкому Александру Петровичу, за написанные для данной книги приложения;
- ❑ еще раз своему мужу за роль самого строгого технического редактора и мужество меня критиковать (наш брак не распался, а книга, несомненно, стала лучше);
- ❑ своей дочери за поиски программистских "приколов" в Интернете для эпиграфов к главам и за время, потраченное на прочтение моего труда;
- ❑ студентам, которые обучались и обучаются у меня языку C++ и постоянно "подкидывают" все новые программистские головоломки;
- ❑ своей собаке, которая отрывала меня от компьютера и выводила погулять.



Глава 1

Общие принципы процедурного программирования

Вчера написал программу. Работает нормально и не глючит... Может быть, я что-то не так делаю?

1.1. О современном программировании в целом

Время диктует новые требования к создаваемому программному продукту. А если программист не учитывает изменение условий, то созданный им продукт становится неконкурентоспособным.

1.1.1. Историческая справка

Чтобы убедить начинающего программиста в том, что современные условия требуют от него знаний и умений, далеко выходящих за рамки освоения любого языка программирования, в табл. 1.1 приведена сравнительная характеристика той ситуации, что была на заре программирования и имеется в настоящее время.

Таблица 1.1. Эволюция отношения программиста к создаваемой программе

Давным-давно	Сейчас
Компьютер был предметом роскоши (являлся редкостью и стоил очень дорого), поэтому его ресурсы ценились гораздо дороже труда программиста	Компьютер занял место в ряду бытовых приборов, а оплата труда программиста составляет большую часть стоимости программного продукта
Возможности компьютера были скромными, программы — относительно небольшими, узкоспециализированными	Возможности компьютера сказочно выросли (и продолжают расти), программы стали универсальнее

Таблица 1.1 (окончание)

Давным-давно	Сейчас
С программой работал в большинстве случаев только сам автор, поэтому он текст программы писал для себя, как правило, воздерживаясь от документирования и игнорируя структуру программы. А в результате модифицировать старую программу было сложнее, чем написать новую, работа же над проектом в команде была практически невозможна	Сформированы правила хорошего стиля создания программных продуктов. Проекты становятся большими и сложными, поэтому необходимыми требованиями являются структурирование текста и документирование. Все меньше требуется одноразовых программ, поэтому писать их приходится с учетом будущих модификаций и возможности постороннего сопровождения. Прошли времена программистов-одиночек, обычным требованием при приеме на работу является умение корректно и эффективно взаимодействовать с другими участниками проекта
Пользователь к работе с программой не допускался (заказчик приносил исходные данные и забирал результат), поэтому на интерфейс программист времени не тратил	Главным действующим лицом при работе с большинством программ стал <i>пользователь</i>, поэтому появился пресловутый термин "интуитивно понятный пользовательский интерфейс"

1.1.2. Этапы создания программного продукта

Может быть, сегодня и существуют еще программисты, которые, получив задачу, сразу же приступают к кодированию, но на работу в серьезной фирме они вряд ли могут рассчитывать. Разработка *хорошей* программы происходит в соответствии с жизненным циклом программного обеспечения. Поэтому даже начинающему программисту неплохо бы представлять себе, что его ожидает, и сразу же привыкать к правилам хорошего тона, потому что, только освоив все понятия, связанные с разработкой программного обеспечения, можно перейти с уровня простого кодировщика на уровень системного аналитика или менеджера проекта. Приведенная иллюстрация (рис. 1.1) не учитывает итеративности процесса, но содержит все основные рекомендуемые этапы разработки программного продукта.

Этапы создания хорошего программного продукта

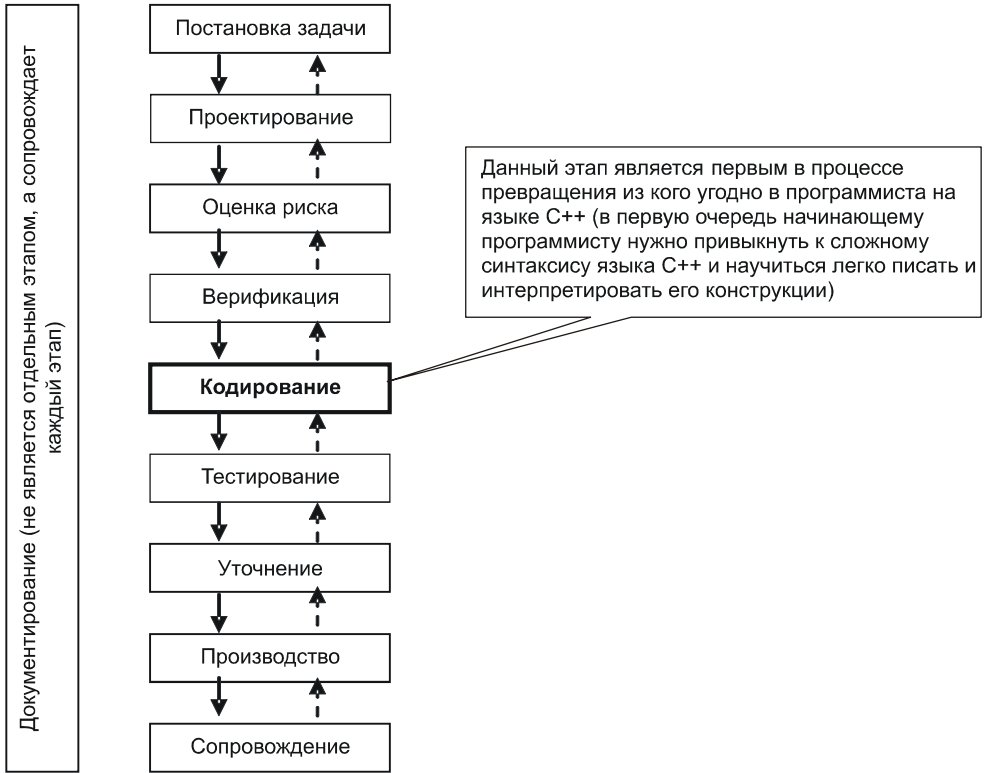


Рис. 1.1

1.1.3. Памятка программисту

Причины, по которым стоит уделять особое внимание структуре текста программы:

- каким бы образом ни велась разработка (по правилам или вопреки оным), результатом является программный продукт, стоимость которого складывается из стоимости ресурсов компьютера и оплаты труда программиста (рис. 1.2);

Следствие

Чем хуже организованы этапы разработки и чем хуже *структурирован* текст, тем больше времени затрачивает программист, увеличивая стоимость программы;

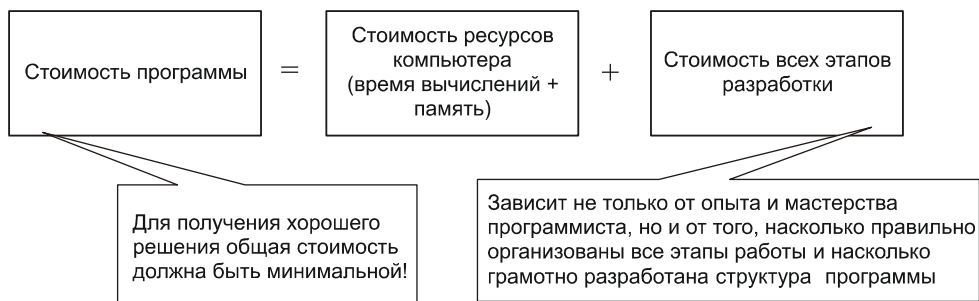


Рис. 1.2

- ❑ в условиях конкуренции программист должен как можно быстрее получить конечный продукт (всегда есть вероятность того, что пока вы будете создавать нечто сверхэффективное, другой программист предложит вашему заказчику не такую супер, но довольно сносно решающую задачу программу);

СЛЕДСТВИЕ

Чтобы минимизировать общее время разработки, программист должен *хорошо структурировать* текст программы и документировать каждый этап разработки.

- ❑ если текст будет хорошо *структурирован*, разработанные фрагменты можно легко использовать при решении других аналогичных задач;
- ❑ не стоит ожидать от заказчика грамотно сформулированного технического задания на программный продукт (таких заказчиков не бывает). Обычно заказчик лишь смутно представляет себе, чего он хочет, поэтому, скорее всего, вам придется не раз менять структуру вашей программы, пока, наконец, сами не сформируете себе техническое задание на программный продукт и не объясните вашему заказчику, чего он хочет;
- ❑ даже когда вы получите деньги за свою работу, не обольщайтесь тем, что удалось создать нечто вечное (чудес не бывает). Скорее всего, вам еще долго придется это произведение сопровождать (т. е. доделывать и переделывать). Чем лучше структурирована и документирована программа, тем меньше усилий потребует от вас этот этап;
- ❑ очень плохо действует на заказчика синий экран, возникающий во время демонстрации вашей программы, поэтому в структуру программы сразу же следует заложить обработку нештатных ситуаций.

Кроме того, необходимо учитывать интересы заказчика, который обычно считает себя профессионалом и предполагает, что для управления программой достаточно одной кнопки, при нажатии на которую программа сама выпол-

нит все требуемые действия. Если же вам удастся убедить его в том, что одной кнопкой никак не обойтись, то в ваших же интересах создать для взаимодействия пользователя с программой предельно интуитивно понятный интерфейс.

1.1.4. Критерии хорошего программного продукта

Подводя итог, можно сформулировать критерии хорошего современного программного продукта (рис. 1.3). Как и прежде, существенную роль играют минимизация времени выполнения и рациональное использование памяти (особенно при программировании для встроенных применений). Современные компиляторы самостоятельно умеют оптимизировать многие языковые конструкции, в то время как исправлять структуру вашей программы они не могут, вот поэтому требования к структурному построению текста программы в большинстве случаев становятся определяющими.



Рис. 1.3

1.2. Структура программы

Если мне еще не удалось убедить вас в важности создания хорошо структурированного текста программы, то приведу цитату гуру программирования Б. Страуструпа: "Вы можете написать небольшую программу (скажем, 1000 строк),

используя грубую силу и нарушая все правила хорошего стиля. Для программы большого размера вы не сможете этого сделать. Если структура программы, состоящей из 100 000 строк, плоха, вы обнаружите, что новые ошибки появляются с той же скоростью, с которой исправляются старые. Язык программирования C++ разрабатывался таким образом, чтобы предоставить возможность рационально структурировать большие программы, и чтобы один человек мог работать с большим объемом кода".

Под структурным программированием понимается метод программирования, обеспечивающий создание текста программы, структура которого:

- отражает структуру решаемой задачи (логическую структуру);
- хорошо читаема не только его создателем, но также и другими программистами.

Так как структурный подход охватывает все стадии разработки проекта, предполагается, что квалифицированный программист, прежде чем приступить собственно к написанию текста программы, продумывает логическую структуру решаемой задачи (сверху вниз). Для этого применяется подход (интуитивно понятный), при котором исходная задача делится на несколько крупных подзадач, каждая из которых, в свою очередь, может быть тоже разделена на подзадачи и т. д. (рис. 1.4). Такая процедура называется декомпозицией задачи.



Рис. 1.4

Следовательно, разрабатывая программу, следует придерживаться определенных правил. Очевидно, что некоторые правила являются общими и не зависят от языка реализации, а некоторые определяются возможностями конкретного языка программирования.

1.2.1. Разбиение на файлы (модульность) и связанные с этим понятия C/C++

В принципе, весь текст программы на Си всегда можно поместить в один файл, однако при написании хоть сколько-нибудь значительных по размеру программ оказывается полезным сгруппировать логически связанные между собой понятия (код и данные, соответствующие подзадачам) и хранить такие совокупности в отдельных файлах.

Разбиение программы на файлы помогает:

- улучшить структуру (программисту удобнее ориентироваться в собственной или чужой программе при внесении изменений);
- уменьшить общее время получения нового загрузочного модуля при внесении изменений только в один из исходных файлов.

Следует отметить, что само физическое разбиение программы на модули реализуется довольно просто, а серьезная проблема при этом состоит в том, что между модулями неизбежно возникают зависимости (пример внешних зависимостей см. на рис 1.6), поэтому следует обеспечить безопасное, удобное и эффективное взаимодействие этих модулей.

Для обеспечения модульности C/C++ (впрочем, как и большинство современных средств разработки) предоставляет возможность компиляции каждого файла по отдельности с последующей стыковкой полученных частей в единый загрузочный (исполняемый) модуль.

Этапы получения загрузочного модуля

Рассмотрим этапы получения загрузочного (исполняемого) файла (рис. 1.5):

1. Программист с помощью текстового редактора формирует исходный файл на C/C++ (обычно такие файлы имеют расширение *.c или *.cpp). При этом в современной интегрированной среде разработки (Integrated Development Environment IDE) программисту помогает система подсказок — IntelliSense. При разбиении программы на отдельные файлы в тексте каждого файла разработчик должен объяснить компилятору, каким образом тот должен обращаться с теми внешними понятиями (данными или функциями), которые определены в других файлах, а используются в данном файле.

Этапы получения загрузочного (исполняемого) файла



Рис. 1.5

- Далее производится предварительная обработка текста исходного файла программой-препроцессором (см. главу 5). Препроцессор всегда запускается автоматически перед компиляцией файла. Результат обработки препроцессором исходного файла называется единицей компиляции (это окончательно сформированный текст, с которым уже может работать компилятор).
- Затем происходит компиляция, когда в результате синтаксического, затем лексического анализа, а потом и собственно трансляции получается промежуточный формат файла, называемый объектным форматом (обычно он имеет расширение *.obj). Для того чтобы создать его, компилятору достаточно знать только свойства внешних понятий (таких, как тип переменной или прототип функции). Если программист предоставил такую информацию компилятору, последний уже может сгенерировать код (последовательность процессорных команд), но не может сформировать адреса внешних (по отношению к данному файлу) переменных или адреса внешних функций. При создании объектного файла компилятор "откладывает на потом" разрешение внешних для данного исходного файла зависимостей.
- И, наконец, производится компоновка (синонимы: редактирование связей, линковка, сборка). Это соединение всех ранее откомпилированных частей

(не только ваших, но и кода статических библиотек *.lib) в единый исполняемый модуль (для Windows и DOS — обычно файл с расширением *.exe или *.dll). На этом этапе все объектные модули необходимо обрабатывать совместно, чтобы произвести окончательное распределение памяти и сформировать для всех команд адресные части.

5. Кроме того, компилятор и компоновщик по требованию программиста могут в исполняемом файле добавить к коду дополнительную отладочную информацию (такую, например, как соответствие символических имен переменных их машинным адресам). Эта информация используется программой-отладчиком и позволяет производить отладку на уровне исходного текста программы.

Соответственно перечисленным этапам программисты получают:

- ☐ на этапе компиляции — синтаксические ошибки и ошибки неописанных внешних объектов;
- ☐ на этапе компоновки — ошибки неразрешенных или неуникальных внешних зависимостей.

ЗАМЕЧАНИЕ

С логическими ошибками программист должен бороться самостоятельно (ни компилятор, ни компоновщик не помогут!).

Раздельная компиляция

Для того чтобы стало возможным разбиение исходного текста на отдельные файлы, в C/C++ реализован механизм раздельной компиляции (каждый исходный файл обрабатывается компилятором независимо от других).

Но на практике обычно в каждом файле используются понятия, определенные в других файлах — внешние зависимости. Например:

- ☐ код функции может находиться в одном файле, а вызов функции — совсем в другом;
- ☐ переменная определена в одном файле, а использовать ее нужно в другом.

Простейший пример возникновения внешних зависимостей приведен на рис. 1.6. В примере показано, как одно и то же выражение на языке высокого уровня в зависимости от свойств понятий, внешних по отношению к данному файлу (в примере имеются в виду переменные X, Y и Z), компилятор превращает в совершенно разные последовательности низкоуровневых команд. Очевидно, что прежде чем использовать внешние понятия, программист должен объяснить компилятору, каким образом следует с ними обращаться (описать свойства X, Y и Z, чтобы он (компилятор) мог сгенерировать соответствующую

последовательность низкоуровневых команд), а компоновщик сформирует адреса внешних переменных.

Что такое внешние зависимости (на примере внешних данных)

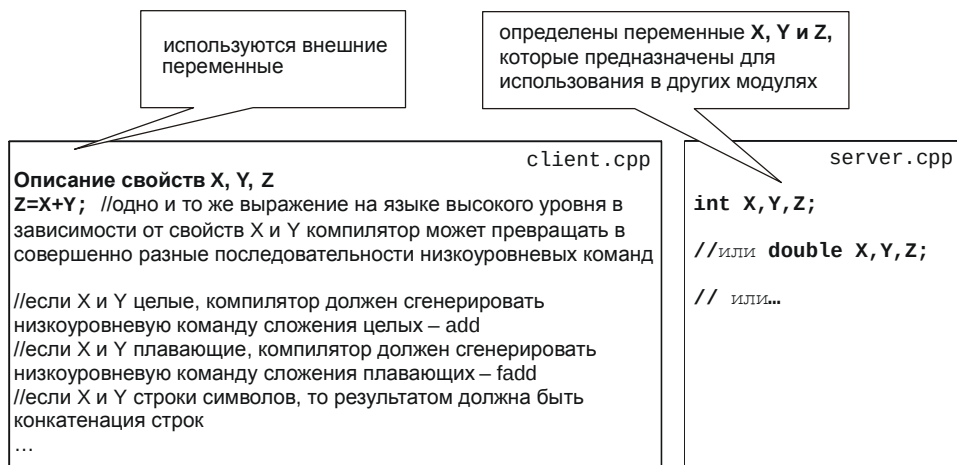


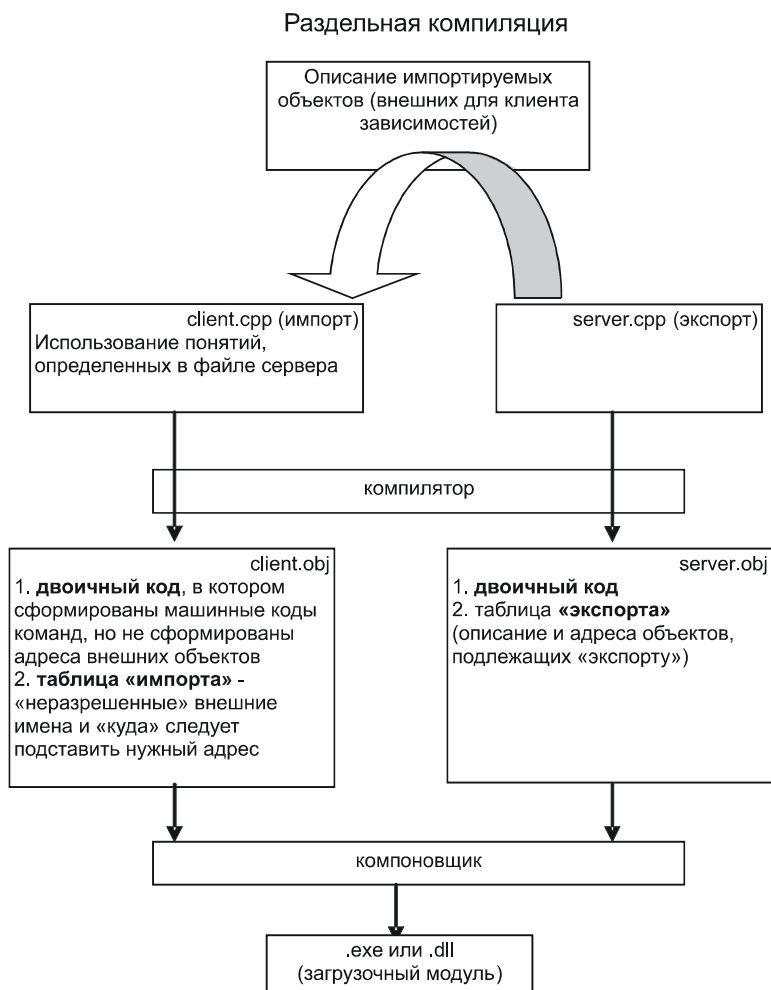
Рис. 1.6

Рассмотрим обязанности по разрешению внешних зависимостей:

- ❑ *программист* должен в тексте программы, во-первых, описать свойства всех внешних для данного файла понятий, а во-вторых — обеспечить уникальность каждого описания (интерфейса). Если сравнивать программу с аппаратной системой, то каждый файл похож на блок, подключаемый к другому блоку с помощью разъема (правильно спроектированные разъемы на двух стыкуемых частях должны обеспечить уникальность соединения, чтобы их невозможно было перепутать, и вся система в целом не сгорела);
- ❑ *компилятор* по описанию внешних зависимостей генерирует в объектном модуле (рис. 1.7) последовательность низкоуровневых (процессорных) команд, а также таблицу описания входов/выходов (где указано, куда нужно подставить адрес каждого внешнего объекта, плюс описание свойств самого объекта, чей адрес нужно подставить). Если продолжить аналогию с аппаратурой, то можно сказать, что компилятор формирует отдельный блок плюс разъем для подключения данного блока к другим;

ЗАМЕЧАНИЕ

Сам по себе каждый отдельный блок работать не будет, их нужно стыковать;

**Рис. 1.7**

- *компоновщик* анализирует таблицы, созданные компилятором (рис. 1.7), и ищет для каждого входа соответствующий выход (уникальный). Нашел — "соединяет проводами" (подставляет адрес внешнего объекта), не нашел — выдает ошибку.

ЗАМЕЧАНИЕ 1

На самом деле каждый файл может быть одновременно и клиентом, и сервером, поэтому обычно в объектном модуле формируются две таблицы: одна для импортируемых понятий, другая — для экспортируемых.

ЗАМЕЧАНИЕ 2

Термины экспорт/импорт в данном контексте не имеют никакого отношения к динамически подключаемым библиотекам (*.dll).

Понятие проекта

Программисты называют проектом совокупность файлов с исходными текстами и служебных файлов, в которых содержится дополнительная информация для средств трансляции.

Процесс компиляции каждого исходного файла может иметь свои особенности. Это означает, что в параметрах командной строки при компиляции каждого файла как компилятору, так и препроцессору можно указать разные опции (ключи).

В параметрах командной строки компоновщику нужно указать, из каких объектных модулей и объектных библиотек собирать исполняемый файл, а также можно перечислить особенности сборки исполняемого файла.

Если бы мы работали из командной строки, то все эти особенности пришлось бы указывать в опциях командной строки вручную или писать командные (*.bat) файлы. Но в настоящее время большую часть такого рода работы выполняет интегрированная среда разработки (IDE). Все перечисленные (и многие другие) параметры, задающие свойства исполняемого модуля, объединяются в *опциях проекта* и сохраняются в служебных файлах. А встроенные в некоторые IDE средства автоматизации (т. н. Wizard) облегчают программисту работу с интегрированной средой тем, что предоставляют разработчику заготовку (template) программы требуемого типа. При этом большая часть опций (а может быть, даже и все) генерируется автоматически.

1.2.2. Функциональная декомпозиция и связанные с ней понятия

Функция C/C++ представляет собой фрагмент кода, на который можно передать управление (вызвать) из любого места программы. По окончании функции выполнение будет продолжено за точкой вызова (см. подробнее главу 8).

Функции позволяют использовать один и тот же код для работы с разными наборами данных (с этой точки зрения можно рассматривать функции как нижний уровень абстракции программирования). Они являются краеугольным камнем процедурного программирования. Функциональная декомпозиция — это представление программы в виде иерархии вложенных вызовов функций. Посредством использования функций решаются две задачи:

- ☐ улучшается структура текста программы;
- ☐ программисту предоставляется средство, позволяющее не дублировать код.

Продолжая разбиение программы на все более мелкие части (см. рис. 1.4), мы стремимся к тому, чтобы каждая такая часть содержала какое-либо законченное действие. Физически это выражается в разбиении исходных файлов на более мелкие единицы — функции. При процедурном подходе программу можно представить в виде вложенных (иногда рекурсивных) вызовов функций (рис. 1.8).

Функциональная декомпозиция

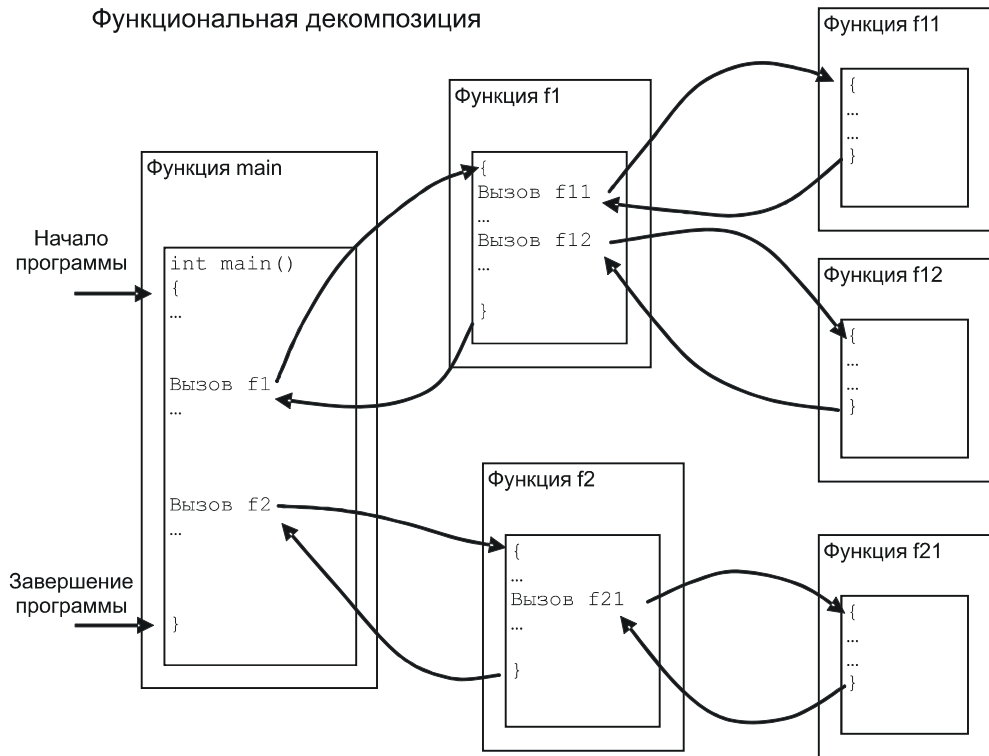


Рис. 1.8

Обычно программист помещает несколько функций в один файл, объединяя их по какому-то логическому принципу, при этом код вызываемой функции может находиться в одном файле, а вызов — в другом (рис. 1.9). В этом случае программисту необходимо объяснить компилятору, каким образом тот должен формировать вызов внешней по отношению к данному модулю функции, оставив компоновщику обязанность подставить адрес точки вызова (куда передать управление).

Внешние функции

```
server.cpp
void f(int a,double b,char c)
{
    тело функции
}
```

```
client.cpp
//описание свойств функции
...
f(1, 2.2, 'A');
//для того чтобы компилятор сгенерировал
вызов функции, он должен знать: имя функции,
количество и типы параметров, тип
возвращаемого значения. Тогда он сможет
сгенерировать низкоуровневую
последовательность команд с точностью до
указания адреса - куда передать управление.
А адрес сформирует компоновщик.
```

Рис. 1.9

Функции C/C++ вообще

Функция C/C++ — это фрагмент кода, оформленный определенным образом (в частности, ограниченный фигурными скобками) и выполняющий некоторое законченное действие. На рис. 1.10 с помощью псевдокода показано, как следует оформлять функцию на C/C++ и что может быть в теле функции.

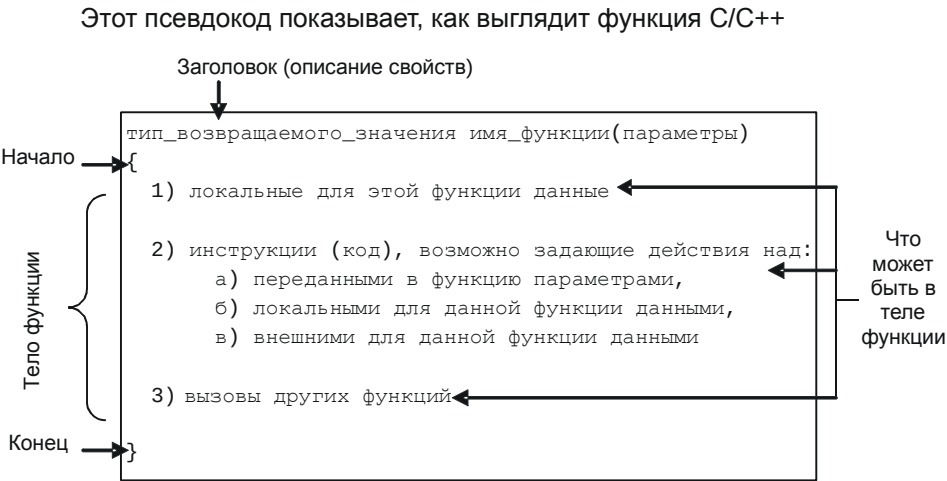


Рис. 1.10

ВАЖНО!

Программист посредством фигурных скобок дает синтаксическое указание компилятору о начале и завершении функции. Компилятор же подставляет вместо открывающей и закрывающей фигурных скобок низкоуровневые команды, обеспечивающие корректный вызов функции и возврат управления вызывающему коду.

После того как функция написана (и отлажена), программист может забыть о том, как она устроена внутри, а посторонний пользователь и вовсе не должен задумываться о внутренней начинке, ему надо знать только назначение и интерфейс вызова этой функции (имя функции, типы параметров). Такую функцию можно рассматривать как новую мощную команду, выполняющую действия над заданными программистом значениями.

Функция *main* в частности

Начало специальной функции с предопределенным именем `main()` является точкой входа для программ на C/C++. Хотя наличие этой функции обязательно, она не генерируется компилятором автоматически — программист должен обеспечить ее наличие в тексте программы явно!

Минимальной программой на C/C++ является:

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

Так как начало функции `main()` является точкой входа программы, то у нее есть особенности:

- каждая программа на C/C++ обязательно должна содержать функцию с именем `main`;



Подумайте, кто выдаст ошибку, если программист забыл определить функцию с таким именем?

- имя `main` может быть только у единственной функции в вашей программе;



Подумайте, кто выдаст ошибку, если программист определил несколько функций с таким именем?

- открывающая скобка функции `main()` является началом вашей программы, закрывающая — выходом (во всяком случае, корректным);
- функция `main()` на самом деле является не абсолютной, а относительной точкой входа, т. к. перед вызовом этой функции (в отличие от всех других) компилятор генерирует невидимый программисту стартовый блок кода (пролог), а после завершения — эпилог всей программы (рис. 1.11);
- компилятор понимает несколько форм функции `main()`:

- `int main()` //не принимает параметров
- `int main(int argc[, char *argv[ ] ] [, char *envp[ ] ])`
//см. разд. 8.2.3
- `int wmain()` //поддержка UNICODE, специфика Microsoft
- `int wmain(int argc[, wchar_t *argv[ ] ] [, wchar_t *envp[ ] ])`

Как вызывается функция main

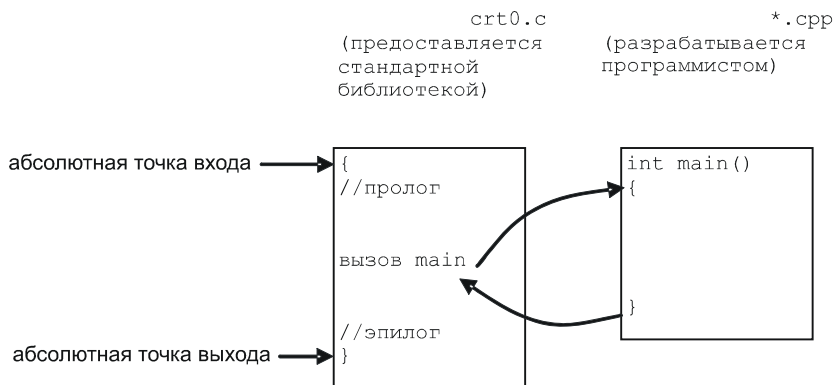


Рис. 1.11

- хорошим стилем является возврат функцией `main()` системе кода завершения (нулевое значение обозначает нормальное завершение программы). Альтернативный (нерекомендуемый) путь — определить функцию с ключевым словом **void** (см. разд. 8.1.1):

```
void main() {...}
```

- несколько ограничений, относящихся к функции `main()` (но не касающиеся всех остальных функций):
 - не должна быть перегружена программистом, т. е. можно использовать только приведенные ранее формы, а свои придумывать нельзя (см. разд. 8.5);
 - не может быть объявлена с ключевым словом **inline** (см. разд. 8.1.4);
 - не может быть объявлена с ключевым словом **static** (см. разд. 3.7.1);
 - нельзя (без дополнительных ухищрений) рекурсивно вызвать эту функцию.

Завершение программы

Если точка входа только одна, то возможностей завершения программы в C/C++ (рис. 1.12) имеется несколько:

- по закрывающей скобке функции `main()`;
- выполнение инструкции **return** из функции `main()`;
- вызов библиотечной функции `exit()` в любом месте программы;
- вызов библиотечной функции `abort()` в любом месте программы.

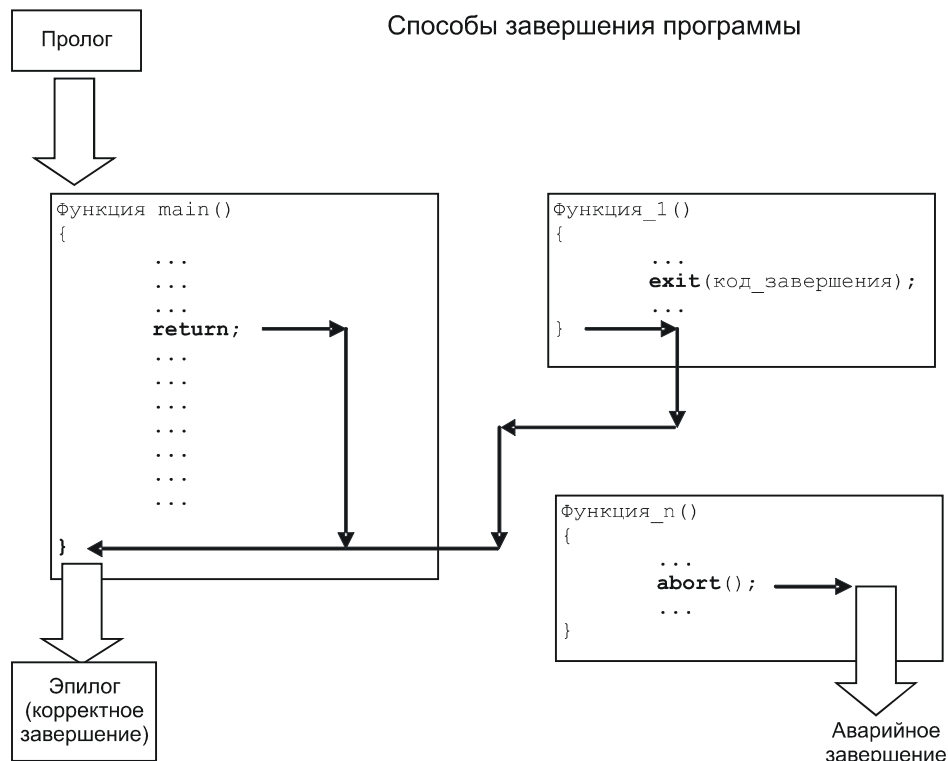


Рис. 1.12

Первые три способа позволяют завершить программу корректно. Функция `abort()` завершает программу аварийно.

ЗАМЕЧАНИЕ 1

Несмотря на то, что способов корректного завершения несколько, вы, тем не менее, можете предусмотреть и какие-то конкретные действия, которые гарантированно будут выполняться при корректном завершении программы. Для этого следует оформить эти действия как самостоятельную функцию и с помощью библиотечной функции `atexit()` задать ее вызов (в случае корректного завершения программы) в эпилоге, предоставляемом стандартной библиотекой.

ЗАМЕЧАНИЕ 2

Вы можете предусмотреть не одно, а несколько последовательно выполняемых завершающих действий, создав стек вызываемых функций посредством нескольких вызовов библиотечной функции `atexit()` (листинг 1.1). Эти функции будут вызываться при корректном завершении программы в обратном порядке (последняя указанная будет вызвана первой).

Листинг 1.1. Дополнительные действия при корректном завершении программы

```
#include <cstdlib>
void fn1( void );
void fn2( void );
int main( void ) {
    atexit( fn1 );
    ...
    atexit( fn2 );
} //в эпилоге будут вызваны функция fn2, а затем fn1
```

Аварийность выхода по `abort()` заключается в том, что функция `abort()` вызывает немедленное прекращение выполнения программы с кодом завершения 3, при этом:

- ☐ не закрываются открытые или временные файлы;
- ☐ не очищаются буферы потоков ввода/вывода;
- ☐ не вызываются деструкторы для глобальных и статических объектов;
- ☐ не выполняются дополнительные специализированные действия, предусмотренные программистом посредством `atexit()`.

ЗАМЕЧАНИЕ

Если функция `abort()` вызывается в консольном Win32-приложении, в Debug-версии проекта появляется окно сообщений (Message Box) с диагностикой об аварийном завершении приложения. Программисту предоставляются три возможности отреагировать на ситуацию: прервать, повторить, пропустить.

1.2.3. Блоки кода

Продолжим рассмотрение процесса декомпозиции задачи (см. рис. 1.4).

В каждой функции, в свою очередь, можно выделить логически обособленные фрагменты (блоки) кода. Такой фрагмент, заключенный в фигурные скобки, называется составной инструкцией.

На рис. 1.13 с помощью псевдокода показано, как следует оформлять блок кода на C/C++ и что может быть в теле составной инструкции. Приведенный в качестве примера блок кода может выполняться только при определенном условии. Для того чтобы сообщить об этом компилятору, программист должен заключить такую последовательность в фигурные скобки.

Внутри каждого такого блока могут быть определены элементы данных, локальные для этого блока (используемые только в пределах текущего блока)

(см. разд. 3.7 и 3.8), и инструкции, выполняющие действия над локальными данными и, возможно, над данными, внешними для текущего блока.

Этот псевдокод показывает, как выглядит составная инструкция C/C++

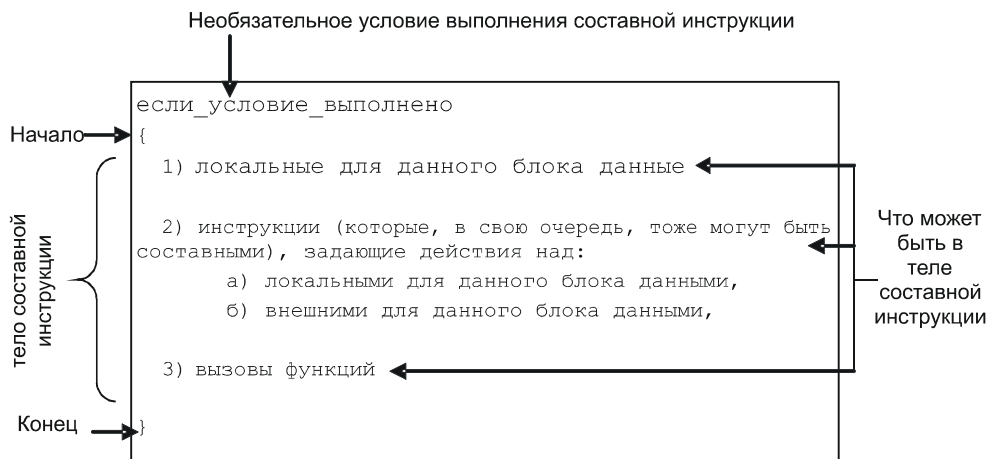


Рис. 1.13

ЗАМЕЧАНИЕ 1

Вложенность блоков достаточно велика и определяется свойствами компилятора.

Для того чтобы улучшить визуальное восприятие текста программы, рекомендуется смещать текст каждого вложенного блока на несколько позиций вправо. Именно так поступает текстовый редактор в современных интегрированных средах разработки (если только вы ему в этом не препятствуете):

```

{ // начало блока
    ...
    { // вложенный блок
        ...
        { // еще вложенный блок, хорошо видно, где начало
            ...
        } // и где окончание каждого блока
        ...
    }
    ...
}
  
```


Таким образом, с помощью оформления относительно независимых частей программы в виде функций и выделения логических блоков внутри функций, создается хорошо читаемая иерархическая структура программы.

1.2.4. Оформление текста программы. Комментарии и отступы

Давным-давно стиль оформления текста программы зависел исключительно от желания и аккуратности программиста. Сейчас оформление кода (code style) является одним из основных требований и в большинстве организаций является обязательным. В частности, удачно подобранный и грамотно написанный набор комментариев является существенной частью хорошей программы.

ЗАМЕЧАНИЕ 1

Написание правильных комментариев может оказаться не менее сложной и трудоемкой задачей, чем написание самой программы. Полезно начинать писать программу с комментариев, где вы естественным языком описываете, что должна делать комментируемая часть кода.

ЗАМЕЧАНИЕ 2

Комментарии пишутся для программиста, компилятору они не нужны, поэтому препроцессор исключает комментарии из того окончательно сформированного текста, с которым работает компилятор.

ЗАМЕЧАНИЕ 3

В современных интегрированных средах разработки текстовый редактор выделяет закомментированные участки текста другим цветом.

В C/C++ имеются два вида комментариев:

- `//` — это комментарий первого вида (в стиле C++). Препроцессор исключит весь текст, следующий за `//` до конца строки;
- `/* */` — а это комментарий второго вида (в стиле стандартного Си). Препроцессор исключит весь текст, находящийся между `/* ... */`.

ЗАМЕЧАНИЕ

Комментарии второго вида не могут быть вложенными, но в них можно вложить комментарий первого вида!

Пример 1.

```
/*  
    /* эта открывающая скобка будет проигнорирована  
    */ первая встреченная закрывающая скобка будет означать  
        конец комментария второго вида  
*/а этот фрагмент уже будет отправлен компилятору и вызовет  
    ошибку
```

Пример 2.

```
/*  
    // ОК! Здесь никаких ошибок не произойдет!  
*/
```

Несколько рекомендаций:

- ☐ комментарии должны быть! — это часть документирования программы;
- ☐ плохой комментарий хуже, чем его отсутствие! (Б. Страуструп);
- ☐ не стоит комментировать то, что и так очевидно;
- ☐ обязательно комментировать понятия, которые используются разными единицами трансляции (тем более, если ими может воспользоваться другой программист);
- ☐ всегда полезно комментировать непереносимый или неочевидный код (хотя такового быть не должно!);
- ☐ хорошим тоном считается начинать с комментария каждый исходный файл и определение каждой функции;
- ☐ использование комментариев позволяет на время исключить (закрывать от компилятора) отдельные части исходного текста программы, и этим весьма широко пользуются при отладке.

ЗАМЕЧАНИЕ

Комментарии к фрагментам, приводимые в данной книге, не всегда являются хорошим примером того, как следует комментировать рабочую программу, поскольку они всего лишь преследуют цель обучения и предназначены для начинающих изучение программирования на C/C++.



Глава 2

Базовые понятия C/C++

2.1. Ключевые слова

Ключевые слова — это слова, зарезервированные для использования только компилятором C/C++. Программист не может использовать ключевые слова языка для именования своих *объектов*¹ (использовать их позволено только по прямому назначению).

Примеры ключевых слов:

if, int, static...

ЗАМЕЧАНИЕ

В современных интегрированных средах разработки редактор текста помогает программисту отличать ключевые слова, выделяя их другим цветом.

2.2. Идентификаторы (имена)

Обычно программист присваивает объектам своей программы имена (идентификаторы).

Специфика:

- ☐ идентификатор C++ состоит из последовательности букв и цифр;
- ☐ первым символом должна быть буква;
- ☐ символ подчеркивания "_" приравнивается к букве. Но имена, начинающиеся с символа "_", обычно зарезервированы для специфических нужд

¹ Здесь и далее термин *объект* используется для обозначения любого низкоуровневого понятия C/C++ и не имеет отношения к объектно-ориентированному программированию.

среды или используются стандартной библиотекой, поэтому такие имена не стоит использовать в прикладных программах;

- ❑ компилятор C/C++ различает символы в верхнем и нижнем регистре, поэтому идентификаторы `foo` и `Foo` оказываются для компилятора двумя совершенно разными именами.

РЕКОМЕНДАЦИЯ

Не слишком разумно выбирать имена, отличающиеся только регистром. Это может привести к трудно выявляемым ошибкам;

- ❑ для некоторых компиляторов C++ есть ограничение на количество символов в имени (в языке Си компилятор учитывает только 31 символ);
- ❑ в качестве имен нельзя использовать ключевые слова языка C/C++;
- ❑ чем шире область использования, тем осмысленнее должны быть имена;
- ❑ согласно Венгерской нотации рекомендуется в имени переменной указывать ее тип.

Примеры правильных имен:

```
Hello    var1    this_is_a_very_long_name    _u_name
```

Примеры последовательностей символов, которые нельзя использовать в качестве идентификаторов:

```
123    a fool    your.func    .func    int
```

2.3. Понятия *lvalue* и *rvalue*

Термин `lvalue` (от `left value`) имеет следующий смысл: "нечто, что может быть использовано слева от знака равенства".

В простейшем случае:

```
x=1;    //переменная x является lvalue
```

А в общем случае `lvalue` может быть любым выражением, результат которого ассоциируется компилятором с выделенной областью памяти.

`Lvalue` бывают двух типов:

- ❑ модифицируемые (которым можно присвоить значение сколько угодно раз);
- ❑ немодифицируемые (значение присваивается один раз при инициализации).

Что может выступать в качестве `lvalue`:

- ❑ в большинстве случаев слева от знака равенства находится имя (идентификатор) переменной:

```
x=выражение; //OK
x=7; //OK
```

```
//но!
```

```
x+y = выражение; //ошибка: слева не lvalue
```

```
7=выражение; //ошибка: слева не lvalue
```

- иногда слева от знака равенства используются довольно сложные конструкции типа:

```
*p[x+y] = выражение; //если результату выражения слева от знака
                      равенства соответствует выделенная
                      компилятором память, то выражение слева
                      является lvalue
```

ЗАМЕЧАНИЕ

Таким образом, сложная конструкция всего лишь объясняет компилятору, как ему следует вычислять адрес в памяти, куда должен быть записан вычисленный результат;

- результат выражения справа от знака равенства, в свою очередь, тоже может быть lvalue для следующего выражения, например:

```
x=y=выражение
```

ЗАМЕЧАНИЕ

Термин rvalue был введен для того, чтобы как-то назвать все то, что, в свою очередь, может находиться справа от знака равенства. При этом любое lvalue всегда может находиться справа от знака равенства, обратное справедливо далеко не всегда!

2.4. Операторы

В русскоязычных переводах используются разные названия для обозначения одних и тех же понятий. В данной книге русское "оператор" соответствует английскому "operator", а русскому "инструкция" соответствует английское "statement".

С операторами связаны следующие понятия:

- *количество операндов*. Оператор может совершать действие над:
- одним операндом (унарный оператор);
 - двумя операндами (бинарный оператор);
 - тремя операндами (тернарный оператор);
- *приоритет операторов* (или, как говорили в школе, порядок действий). Если в выражении участвуют несколько операторов, компилятор должен точно знать, в каком порядке требуется выполнять (вычислять) отдельные

части выражения. Такой предопределенный порядок называется приоритетом операторов:

$a+b+c$ — эквивалентно $(a+b)+c$;

$a+b*c$ — эквивалентно $a+(b*c)$.

Для изменения порядка вычисления можно использовать скобки:

$a+b*c$ — эквивалентно: $a+(b*c)$;

$(a+b)*c$ — явно изменили порядок вычислений;

- *ассоциативность операторов*. Если в выражении содержатся операторы с одинаковым приоритетом (без использования скобок), вычисление производится согласно правилам ассоциативности справа налево или слева направо.

ПРАВИЛО

Унарные операторы и операторы присваивания правоассоциативны (т. е. компилятор организует порядок выполнения действий по правилу "справа налево"), а все остальные левоассоциативны. Это означает:

$a=b=c$ — эквивалентно: $a=(b=c)$

$a+b+c$ — эквивалентно: $(a+b)+c$

Для изменения порядка вычисления также можно использовать скобки:

$a=b=c$ — эквивалентно: $a=(b=c)$;

$(a=b)=c$ — явно изменили порядок вычислений;

- *порядок синтаксического разбора* выражения компилятором. Это правила, по которым действует компилятор при синтаксическом разборе выражений, где возможна неоднозначная трактовка выражения (см. пример 4 разд. 2.4.1).

В табл. 2.1 приведены операторы C++ (большинство операторов также имеют место в языке Си). В каждом блоке таблицы (блоки разделены двойной горизонтальной линией и расположены по убыванию приоритета) представлены операторы с одинаковым приоритетом.

Таблица 2.1. Операторы C++

Оператор	Название	Синтаксис использования	Ассоциативность
::	Разрешение области видимости	::имя – глобально имя_пространства::имя – посредством имени пространства имен имя_класса::имя_члена_класса – посредством имени класса	Нет

Таблица 2.1 (продолжение)

Оператор	Название	Синтаксис использования	Ассоциативность
[]	Доступ к элементу массива по индексу	Идентификатор[выражение] или выражение[идентификатор]	Слева направо
()	Вызов функции	Имя_функции(список_параметров)	
()	Конструирование значения	Тип(список_параметров)	
.	Обращение к члену структуры или класса посредством объекта или ссылки	Объект.член_класса Ссылка_на_объект. Член_класса	Слева направо
->	Обращение к члену структуры или класса посредством указателя на объект	Указатель-> член_класса	
++	Постфиксный инкремент	lvalue++	Нет
--	Постфиксный декремент	lvalue--	
new	Динамическое создание объекта	new type — создать в куче объект типа type	Нет
delete	Уничтожить объект (освободить память)	delete указатель_на_объект	
delete[]	Уничтожить массив объектов	delete[] указатель_на_массив_объектов	
++	Префиксный инкремент	++lvalue	
--	Префиксный декремент	--lvalue	
*	Разыменование	* выражение	
&	Получение адреса объекта	&lvalue	
+	Унарный плюс	+ выражение	
-	Унарный минус (арифметическое отрицание)	- выражение	

Таблица 2.1 (продолжение)

Оператор	Название	Синтаксис использования	Ассоциативность
!	Логическое отрицание (not)	! выражение	Нет
~	Поразрядная инверсия	~ выражение	
sizeof	Размер (в байтах)	sizeof(объект) — размер объекта sizeof (type) — размер типа	
typeid()	Идентификация типа	typeid (type) — идентификация типа времени выполнения	
(type)	Приведение (преобразование типа)	(type) выражение — приведение типа выражения к типу, указанному в скобках (старый стиль)	Справа налево
const_cast	Константное преобразование типа	const_cast <type> (выражение)	Нет
dynamic_cast	Преобразование типа с проверкой во время выполнения	dynamic_cast <type> (выражение)	
reinterpret_cast	Преобразование типа без проверки во время компиляции	reinterpret_cast <type> (выражение)	
static_cast	Преобразование типа с проверкой во время компиляции	static_cast <type> (выражение)	
.*	Выбор члена класса посредством объекта	объект.*указатель_на_член_класса	Слева направо
->*	Выбор члена класса посредством указателя на объект	указатель_на_объект ->* указатель_на_член_класса	
*	Умножение	выражение * выражение	Слева направо
/	Деление	выражение / выражение	
%	Остаток от деления (деление по модулю)	выражение % выражение	
+	Сложение	выражение + выражение	Слева направо
-	Вычитание	выражение - выражение	

Таблица 2.1 (окончание)

Оператор	Название	Синтаксис использования	Ассоциативность
<<	Сдвиг влево	выражение << выражение	Слева направо
>>	Сдвиг вправо	выражение >> выражение	
<	Меньше	выражение < выражение	Слева направо
>	Больше	выражение > выражение	
<=	Меньше или равно	выражение <= выражение	
>=	Больше или равно	выражение >= выражение	
==	Равно	выражение == выражение	Слева направо
!=	Не равно	выражение != выражение	
&	Побитовое И (and)	выражение & выражение	Слева направо
^	Побитовое исключающее ИЛИ (or)	выражение ^ выражение	Слева направо
	Побитовое ИЛИ (or)	выражение выражение	Слева направо
&&	Логическое И (and)	выражение && выражение	Слева направо
	Логическое ИЛИ (or)	выражение выражение	Слева направо
e1?e2:e3	Условное	выражение ? выражение : выражение	Справа налево
=	Простое присваивание	lvalue = выражение	Справа налево
*=	Совмещенное присваивание	lvalue *= выражение	
/=		lvalue /= выражение	
%=		lvalue %= выражение	
+=		lvalue += выражение	
-=		lvalue -= выражение	
<<=		lvalue <<= выражение	
>>=		lvalue >>= выражение	
&=		lvalue &= выражение	
=		lvalue = выражение	
^=		lvalue ^= выражение	
throw	Генерация исключения	throw выражение	
,	Запятая (последовательность)	выражение , выражение	Слева направо

РЕКОМЕНДАЦИЯ

Если вы не уверены в порядке выполнения операторов, пользуйтесь скобками. Обычно скобки лишними не бывают.

2.4.1. Арифметические операторы

Язык C/C++ включает стандартный набор арифметических операторов:

- ☐ сложение (+);
- ☐ вычитание (-);
- ☐ умножение (*);
- ☐ деление (/).

ЗАМЕЧАНИЕ

Некоторые особенности выполнения действий с целыми числами в ограниченной разрядной сетке рассмотрены в приложении (см. разд. П1.4—П1.10).

Кроме того, в языке C/C++ имеются операторы:

- ☐ остаток от целочисленного деления (%),
- ☐ оператор увеличения на единицу (++);
- ☐ оператор уменьшения на единицу (--).

Остаток от целочисленного деления — оператор %

В примере демонстрируется разница выполнения операторов целочисленного деления и получения остатка от деления нацело:

```
int x=5, y=2, z;  
z = x / y;      //частное от целочисленного деления (z=2)  
z = x % y;      //остаток от деления нацело (z=1)
```

Операторы инкремента ++ и декремента --

Оператор инкремента (++) увеличения переменной на 1 удобно использовать в следующих случаях:

- ☐ вместо записей типа `x=x+1` — запись типа `x++`;
- ☐ вместо записей типа `x+=1` — запись типа `++x`.

Аналогично выглядит использование оператора декремента (--).

Специфика операторов ++ и -- заключается в том, что они могут быть как *префиксными* (инкремент или декремент происходит до вычисления выражения), так и *постфиксными* (инкремент или декремент происходит после

вычисления выражения). В предыдущем примере результат обоих выражений будет одинаковым, т. к. выражение очень простое.

Рассмотрим примеры использования инкремента в более сложных выражениях.

Пример 1.

Постфиксный инкремент:

```
x=1;
y=x++; //сначала вычисляется выражение, т. е. переменной y
        присваивается текущее значение переменной x.
        В результате y=1, а после вычисления выражения
        значение переменной x увеличивается на 1.
        Таким образом, x=2;
```

Пример 2.

Префиксный инкремент:

```
x=1;
y=++x; //сначала модифицируется значение x (x=2), а потом
        переменной y присваивается измененное значение x (y=2);
```

Пример 3.

Результатом префиксного инкремента (++x) является lvalue, а постфиксного (x++) — не lvalue, поэтому:

```
x=1;
// (x++)++; //ошибка, т. к. выражение x++ не является lvalue
(++x)++; //правильно, т. к. результат ++x - lvalue
++x +=y; //тоже корректно с точки зрения компилятора, но писать
        такое выражение вряд ли имеет смысл. Гораздо понятнее
        выглядит x += y+1;
```

Пример 4.

Порядок синтаксического разбора выражения

```
z=x+++y; //программист мог иметь в виду (x++)+y или x+(++y), а у
        компилятора существуют правила, согласно которым он
        должен интерпретировать любое выражение однозначно.
```

Правило

Анализируя выражение, компилятор пытается выделить самое длинное корректное выражение.

В данном примере он будет поступать следующим образом:

$x+$ — корректно;

$x++$ — тоже корректно;

поэтому такое выражение будет интерпретироваться: $z=(x++)+y$.



Исходя из вышеприведенного правила, попробуйте проинтерпретировать следующее выражение:

$z=x+++++y$.

РЕКОМЕНДАЦИЯ

Не используйте операторы инкремента/декремента в сложных выражениях, т. к. разные компиляторы могут интерпретировать такие выражения по-разному и даже один и тот же компилятор может в DEBUG- и RELEASE-версиях генерировать для таких выражений разный код.

Например, два компилятора (VC.net и BorlandC 3.1) вычисляют приведенный ниже пример по-разному:

☐ $x=1; y= (++x) * (++x);$ // VC: $y=9$;

☐ $x=1; y= (++x) * (++x);$ // BC: $y=6$;

2.4.2. Операторы присваивания

Простое присваивание

В простейшем случае для компилятора оператор присваивания означает, что значение, полученное в результате вычисления выражения справа от знака равенства, нужно скопировать в область памяти, на которую ссылается lvalue слева от знака равенства:

$X=\text{выражение};$ //компилятор вычислит выражение и сгенерирует
mov [X],результат

При этом очевидно, что слева от знака равенства должно быть модифицируемое lvalue. Этот оператор только на первый взгляд кажется самым простым (в языке Си он таковым и является), но на самом деле для C++ присваивание — это дело серьезное и нетривиальное.

Сейчас рассмотрим только самые простые применения оператора присваивания, когда смысл выполняемого действия предельно ясен:

$a=b=c=d;$ //эквивалентно $a=(b=(c=d))$, т. к. оператор
присваивания правоассоциативен. Если хотим изменить
порядок вычислений, можно использовать скобки

```
((a=b)=c)=d; //1. a=b  
                2. a=c  
                3. a=d
```



В каком порядке компилятор будет вычислять следующее выражение:

```
(a=b)=c=d;
```

Совмещенные операторы присваивания

Язык C/C++ предоставляет большое количество операторов, которые совмещают присваивание с другой операцией (см. табл. 2.1).

Рассмотрим некоторые примеры.

Пример 1.

`x+=выражение` эквивалентно `x=x+выражение`.

Следует иметь в виду, что такая сокращенная запись не даст выигрыша в эффективности, т. к. большинство компиляторов просто сгенерируют в обоих случаях одинаковый низкоуровневый код.

```
a=1;
```

```
a+=3; //эквивалентно a=a+3. В результате a=4
```

Пример 2.

Совмещенные операторы присваивания правоассоциативны и приоритет всех совмещенных операторов присваивания одинаков:

```
a+=b+=c; //эквивалентно a+=(b+=c);
```

```
(a+=b)+=c; //для изменения порядка вычислений пользуйтесь скобками
```



В каком порядке компилятор будет вычислять следующее выражение:

```
a*=b+=c;
```

ЗАМЕЧАНИЕ

Оператор типа "+=" является отдельной лексемой (токеном) языка, поэтому никакие разделители не допускаются, а выражение `a+ =b`; вызовет ошибку компилятора.

2.4.3. Побитовые операторы

Побитовые операторы (рис. 2.1) позволяют производить действия над отдельными разрядами многобитовых операндов (установка отдельных битов, сброс отдельных битов или инвертирование отдельных битов), а также осу-

ществлять поразрядный сдвиг всего значения вправо или влево. Эта тема обычно вызывает затруднения у программистов, имеющих слабое представление об архитектуре компьютера и о внутреннем представлении данных, поэтому в приложении (см. разд. П1.12) приведено более подробное описание понятий, связанных с побитовыми операторами.

ЗАМЕЧАНИЕ

Компилятор C/C++ разрешает применять побитовые операторы только к целым типам: **char**, **short**, **int**, **long**, **long long**, возможно с модификатором **unsigned** (см. гл. 3). Результатом побитовых операций являются также целые типы.



Рис. 2.1

Каждый из операторов "**|**"(ИЛИ), "**&**"(И), "**^**"(исключающее ИЛИ) независимо выполняет действия над одноименными битами двух многоразрядных двоичных слов:

- **|** — для установки битов;
- **&** — для сброса битов;
- **^** — для инверсии отдельных битов.

Правила истинности для четырех основных логических функций: конъюнкции, дизъюнкции, исключающего ИЛИ, а также инверсии, которые обычно реализованы в системах команд универсальных процессоров, приведены в табл. 2.2.

ЗАМЕЧАНИЕ 1

Язык C/C++ предоставляет побитовые операторы, совмещенные с присваиванием:

```
x |= y;
```

Таблица 2.2. Таблица истинности основных логических функций

Операнд1	Операнд2	Результат		
		AND (&)	OR ()	XOR (^)
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Операнд	Результат NOT (~)
0	1
1	0

ЗАМЕЧАНИЕ 2

Не путайте побитовые операторы: &, |, ~ с логическими: &&, ||, !. Последние возвращают true (1) или false (0) и применяются в основном в инструкциях `if`, `while` и `for`.

ЗАМЕЧАНИЕ 3

Побитовые операции (кроме сдвига вправо) не зависят от того, знаковый операнд или без знака. В приложении (см. разд. П1.12) приведены дополнительные сведения об особенностях команд сдвигов, реализованных в системах команд процессоров.

Пример 1.

Побитовое отрицание:

<code>char x = 5;</code>	x: 0000 0101
<code>char y = ~x</code>	y: 1111 1010

Пример 2.

Побитовое сложение:

<code>char x = 5;</code>	x: 0000 0101
<code>char y = 12;</code>	y: 0000 1100
<code>char z = x y;</code>	z: 0000 1101

Пример 3.

Побитовое умножение:

char x = 5;	x: 0000 0101
char y = 12;	y: 0000 1100
char z = x & y;	z: 0000 0100

Пример 4.

Побитовое исключающее ИЛИ (сложение по модулю 2):

char x = 5;	x: 0000 0101
char y = 12;	y: 0000 1100
char z = x ^ y;	z: 0000 1001

Пример 5.

Сдвиги.

Первый операнд содержит значение, которое нужно сдвинуть, второй — указывает, на сколько разрядов нужно сдвинуть значение первого операнда.

- При сдвиге влево младшая часть всегда заполняется нулями, независимо от знаковости операнда:

char x = 5;	x: 0000 0101
char y = x << 2;	y: 0001 0100
unsigned char x = 5;	x: 0000 0101
unsigned char y = x << 2;	y: 0001 0100

- При сдвиге вправо беззнаковых целых старшая часть всегда заполняется нулями:

unsigned char x = 251;	x: 1111 1011
unsigned char y = x >> 2;	y: 0011 1110

- При сдвиге вправо знаковых целых старшая часть заполняется значением старшего (знакового) разряда:

char x = -5;	x: 1111 1011
char y = x >> 2;	y: 1111 1110
char x = 5;	x: 0000 0101
char y = x >> 2;	y: 0000 0001



Выполните задания, приведенные в табл. 2.3.

Таблица 2.3. Задания на побитовые операторы и операторы сдвига

Номер	Задание
1	Дана целая переменная (<code>int x;</code>), которая может содержать любое значение. Поменяйте в этой переменной местами значения старшего и младшего байтов
2	Дана целая переменная (<code>int x;</code>). Проинвертируйте значения разрядов младшего байта

ЗАМЕЧАНИЕ 1

Сдвиг влево эквивалентен умножению на 2, сдвиг вправо эквивалентен делению на 2. Зачастую компилятор оптимизирует операции целочисленного деления и умножения на степень двойки, генерируя низкоуровневые команды сдвига.

ЗАМЕЧАНИЕ 2

Используя операторы сдвига, имейте в виду следующее: при выполнении низкоуровневой команды сдвига аппаратно может быть задействовано только ограниченное число разрядов второго операнда, поэтому для процессоров X86 (где аппаратно задействованы только 5 младших разрядов второго операнда) выражение (`x<<32`) не будет иметь никакого действия.

2.4.4. Логические операторы и операторы отношения

Логические операторы (табл. 2.4) и операторы отношения используются при формировании логических выражений, дающих в результате только два значения: `true` (1) и `false` (0). Наиболее часто такие выражения встречаются в инструкциях управления потоком вычислений: циклах, условиях.

Таблица 2.4. Логические операторы `!`, `&&`, `||`

Оператор	Операнд 1	Операнд 2	Результат
логическое NOT (<code>!</code>)		<code>true</code>	<code>false</code>
		<code>false</code>	<code>true</code>
логическое AND (<code>&&</code>)	<code>true</code>	<code>true</code>	<code>true</code>
	все остальные комбинации		<code>false</code>
логическое OR (<code> </code>)	<code>false</code>	<code>false</code>	<code>false</code>
	все остальные комбинации		<code>true</code>

Операторы отношения `<`, `<=`, `>`, `>=`, `==`, `!=` сравнивают значения выражений слева и справа от оператора и формируют значения `true` или `false` в зависимости от соотношения выражений.

ЗАМЕЧАНИЕ 1

Так как нулевое значение выражения соответствует `false`, а ненулевое соответствует `true`, то следующие выражения будут эквивалентны:

`if(x!=0)` эквивалентно `if(x)`

`if(x==0)` эквивалентно `if(!x)`

ЗАМЕЧАНИЕ 2

Если результат логического выражения используется в арифметическом выражении, то вместо `false` компилятор подставит `0`, а вместо `true` `1`.

Например: `int x=(y<100);` //если `y<100`, то `x=1`, иначе `x=0`.

Пример.

Пусть требуется определить, находится ли значение переменной `x` в диапазоне `[0, 100]`:

```
int x;  
//сформировали значение x  
//если пытаемся написать на C/C++ условие таким образом,  
как писали в школе:  
if(0<=x<=100)... //то получаем всегда true, т. к. компилятор будет  
интерпретировать это выражение следующим образом:  
сначала он сравнивает значение переменной x  
с нулем и получает false (0) (если x<0),  
и true (1) (если x>=0). Затем сравнивает  
полученный результат с 100. Поскольку оба  
значения всегда меньше 100, то такое выражение  
всегда будет true.  
//Как такое условие написать на C/C++ правильно:  
if( (x>=0) && (x<=100) )...
```

ЗАМЕЧАНИЕ

Для повышения эффективности вычислений компилятор оптимизирует вычисления выражений, в которых используется `&&` или `||`:

`if(выражение1 && выражение2)` — если результатом вычисления первого выражения является `false`, то второе уже не вычисляется;

`if(выражение1 || выражение2)` — если первое выражение дает `true`, то вычислять второе тоже не имеет смысла.



Выполните задания, приведенные в табл. 2.5.

Таблица 2.5. Задания на логические операторы и операторы отношения

Номер	Задание
1	Какой результат дадут два нижеприведенных выражения при исходном значении <code>a=0</code> ? <pre>if(a==0) if(a++>0) if(++a>0)</pre>
2	Каким будет результат следующего выражения? <pre>if(-1)</pre>
3	Сравните два выражения. Поясните разницу при вычислении обоих условий и предскажите результат: <pre>if(x==5) if(x=5)</pre> <i>Подсказка:</i> это обычная ошибка использования оператора присваивания (=) вместо оператора проверки на равенство (==). Второе выражение в примере всегда будет истинно, т. к. результат отличен от нуля
4	Каким будет результат выражения: <code>100 + (x < 40)</code> • при <code>x=0</code> ? • при <code>x=100</code> ?
5	Напишите условие: <code>x > 0</code> и нечетное
6	Чему будут равны значения <code>y</code> и <code>z</code> : <pre>int x=-1; int y = !x; int z = -x;</pre>
7	Вспомните о приоритетах операторов и укажите порядок вычисления выражения: <pre>x=0; y=!++x;</pre>

2.4.5. Тернарный оператор ?:

По смыслу тернарный оператор идентичен конструкции `if...else`, но выглядит компактнее, что вовсе не означает, что компилятор генерирует более эффективный код.

Синтаксис:

```
условие ? true-выражение : false-выражение;
```

Сначала вычисляется условие.

Если условие принимает значение `true`, то вычисляется `true`-выражение, иначе вычисляется `false`-выражение.

Пример 1.

Последовательность команд:

```
if(a<=b) max = b;  
else max=a;
```

можно переписать с помощью тернарного оператора:

```
max = (a<=b) ? b : a; //заключать условие в скобки необязательно,  
но читается так лучше.
```

Пример 2.

Результатом тернарного оператора может быть lvalue. В таком случае последовательность команд:

```
if(a<=b) b=max;  
else a=max;
```

можно переписать с помощью тернарного оператора:

```
((a<=b) ? b : a) = max; //внешние скобки обязательны, иначе  
компилятор решит, что присваивание  
является частью false-выражения
```

Пример 3.

При целочисленном делении на ноль генерируется ошибка времени выполнения. Посредством тернарного оператора можно предусмотреть защиту от такой ситуации:

```
a = b ? 100/b : 0; //если b!=0 (т. е. условие принимает  
значение true), то делить безопасно,  
если же b=0, то деление производиться  
не будет, а переменной a будет просто  
присвоено нулевое значение
```

Пример 4.

Вывод на печать наибольшего из двух значений:

```
std::cout << ( i > j ? i : j ) <<" is greater" << std::endl;  
//скобки являются существенными, т. к. без скобок  
смысл выражения для компилятора будет совершенно  
другим
```

Пример 5.

Тернарный оператор может быть вложенным.

Пусть требуется переменной y присвоить значение:

- 1 — если $x > 0$;
- -1 — если $x < 0$;
- 0 — если $x = 0$.

В таком случае можно записать:

```
y=(x<0) ? -1 : ( (x>0) ? 1:0);
```

//в данном примере для внешнего тернарного оператора **true**-выражение выполняется в случае отрицательного значения **x**, а **false**-выражение в свою очередь является вложенным тернарным оператором и формирует оставшиеся два условия



Даны переменные **x**, **y**, **z**.

С помощью тернарного оператора найдите минимальное из трех значений.

ЗАМЕЧАНИЕ

Для оптимизации вычислений компилятор вычисляет только одно из выражений. Если условие принимает значение **true**, вычисляется только **true**-выражение, если **false**, то только **false**-выражение. Например:

```
int x=1, y=2, z;
```

```
z=(x<y) ? x++ : y++;
```

//сначала переменной **z** будет присвоено значение **x** (**z=1**), а потом значение только этой переменной будет инкрементировано (**x=2**).

2.4.6. Оператор ",", "

Оператор **", "** позволяет сгруппировать несколько выражений там, где компилятор ожидает только одно. Не стоит злоупотреблять этим оператором, поскольку читабельность программы ухудшается.

Пример 1.

```
int a=1, b=2, c;  
c=a,b; //c=1  
c=(a,b); //c=2  
c=a++,b++; //c=1, a=2, b=3
```

Пример 2.

```
int x;  
if(std::cin>>x, x>0){...} //компилятор вычислит все выражения,  
указанные через оператор (,) запятую, но формировать  
условие будет по результату последнего выражения
```



Глава 3

Данные

3.1. Виды данных

С точки зрения программиста видов данных много: есть простые и очевидные (целые и плавающие для представления чисел), часто используются данные для представления символов и строк, кроме того, для каждой конкретной задачи программист может комбинировать простые данные, создавая таким образом бесконечное число специфических пользовательских видов данных.

А с точки зрения процессора все объекты в памяти представляются совокупностью битов, содержащих 0 и 1 (двоичное представление данных). Но:

- разным объектам может соответствовать разное количество битов;
- даже если количество битов одинаково, эти 0 и 1 для каждого вида данных по замыслу программиста имеют совершенно разный смысл (например, форматы хранения коротких плавающих **float** и целых **int** принципиально разные!), поэтому действия процессора должны быть разными.

Возможности процессора по работе с данными разного типа ограничены. Обычно в системе команд процессора имеются низкоуровневые команды для работы с одно-, двух- и четырехбайтовыми целыми, а также специальные низкоуровневые команды для работы с данными в плавающем формате и команды, которые позволяют обращаться к отдельным разрядам. Поэтому, естественно, существуют правила, позволяющие устанавливать взаимосвязь между двоичным представлением данных и их сущностью:

- для некоторых видов данных эти правила просты (например, для целых);
- для некоторых правила достаточно сложны, но всю работу по отображению берет на себя компилятор (например, для плавающих);
- для некоторых (пользовательских типов) соответствие должен установить программист.

По большому счету элементы данных в программе можно разделить на две разновидности:

- *константы* — программист при написании программы точно знает значения, которые будет использовать, не собирается изменять эти значения во время выполнения программы и может заранее сообщить их компилятору;
- *переменные* — данные изменяются в процессе выполнения программы, а программист может задать только начальные значения (проинициализировать элементы данных). Может быть и так, что начальные значения тоже неизвестны заранее, а вычисляются в процессе выполнения программы.

И те, и другие (кроме переменных, содержащих адреса, т. е. указывающих местоположение в памяти других программных элементов) могут быть следующего вида (рис. 3.1):

- *числа: целые или с плавающей точкой* (арифметические типы) — элементы данных, выражающие количества. Арифметические типы имеют различные форматы хранения и размеры. До написания программы программист должен оценить возможный диапазон изменения значений своих данных и, исходя из этой оценки, выбрать подходящий тип для машинного представления данных. С помощью правильного выбора в конкретной ситуации конкретного типа данных программист может минимизировать объем используемой памяти или время выполнения.

ЗАМЕЧАНИЕ1

Переменные целого типа могут быть беззнаковыми или знакопеременными. В *приложении* рассмотрены более подробно способы представления чисел двоичными кодами, их особенности и свойства.

ЗАМЕЧАНИЕ2

Элементы целочисленных данных после трансляции могут занимать в памяти компьютера разное количество байтов. Элемент типа **char** занимает в памяти один байт, элемент типа **short** — два байта, элемент типа **long** — 4 байта. Еще один тип — **int** имеет размер, который зависит от особенностей процессора, и обычно равен разрядности его регистров. Например, в процессорах семейства x86 тип **int** имел длину два байта в младших 16-разрядных моделях, а в современных процессорах тип **int** — четырехбайтовый. Подробную информацию о типах и разрядности элементов данных см. в разд. 3.4.2. В некоторых компиляторах используется нестандартный тип **long long**, длина которого 8 байтов;

- *символы* — целочисленные элементы, предназначенные для хранения кодов отдельных символов, и *строки* — совокупности символов;
- *логические* — для хранения результата логических операций — одного из двух значений: true (истина) и false (ложь);

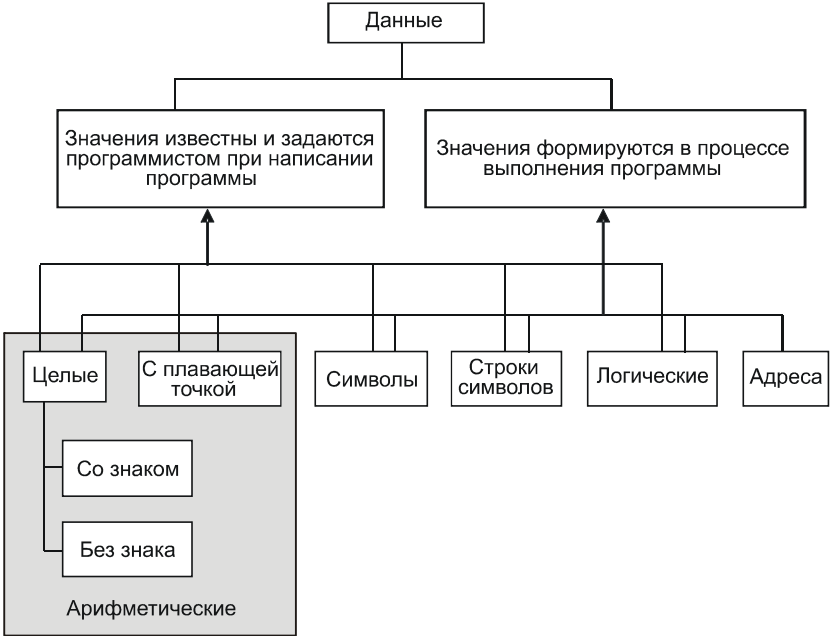


Рис. 3.1

❑ *адреса* — формируются компоновщиком и имеют смысл только во время выполнения программы.

Для того чтобы компилятор мог генерировать низкоуровневый код, он должен знать тип каждого элемента данных.

Тип константы распознается компилятором по ее записи в тексте программы.
Тип переменной программист должен указать компилятору явно.

3.2. Константы (литералы)

Литералом называют текстовое представление значения константы (это тот вид, в котором программисту в тексте программы привычно и удобно задавать значения). Компилятор согласно своим правилам переводит это текстовое представление в двоичный вид, с которым уже может работать процессор. Константа может быть числовой, например:

3
-23.76

0.123e3

а также символьной или строковой, например:

'A'
"Hello World"

3.2.1. Целые литералы

В языке C/C++ можно использовать десятичные, шестнадцатеричные и восьмеричные целые литералы.

ЗАМЕЧАНИЕ

В данном разделе упоминаются различные системы счисления, о них можно более подробно прочитать в *разд. П1 приложения*.

Для того чтобы компилятор мог различать целые литералы по написанию в тексте программы, приняты соглашения, представленные в табл. 3.1.

УТОЧНЕНИЕ

Префиксы для шестнадцатеричных и восьмеричных литералов начинаются с цифры 0 (а не с буквы O).

Таблица 3.1. Префиксы представления целых литералов в разных системах счисления

Система счисления	Префикс	Примеры представления		
десятичная	без префикса	8	10	256
шестнадцатеричная	0x или 0X	0x8 или 0X8	0xA или 0Xa	0x100 или 0X100
восьмеричная	0	010	012	0400

Например:

x = 10; //эта константа (10) задана в десятичной системе
x = 010; //а эта (8) - в восьмеричной системе
x = 0x10; //константа (16) задана в шестнадцатеричной системе

ЗАМЕЧАНИЕ

Программист может задать в программе константу произвольной длины, но в каждом компиляторе на количество цифр существуют ограничения.

По умолчанию (если программист не указал специальным образом) компилятор сам определяет тип и размер целой константы, учитывает этот тип при выполнении действий с константой и для хранения ее значения выделяет

в области кода (т. е. в той части памяти, где будут храниться машинные команды) количество байтов согласно следующим простым правилам:

- ❑ если значение константы помещается в диапазон представления **int** — то столько байтов, сколько занимает на данной платформе **int** (2 либо 4 байта);
- ❑ если значение выходит из диапазона представления **int** — то столько байтов, сколько занимает **long** (4 байта);
- ❑ если значение находится за пределами диапазона **long** — то столько байтов, сколько занимает **long long** (8 байтов);
- ❑ если значение выходит за диапазон **long long**, то компилятор его "обрезает" до 8 байтов. При этом хороший компилятор должен выдавать предупреждение о литералах, слишком длинных для внутреннего представления.

ПРИМЕЧАНИЕ

Тип **long long** в настоящее время не является стандартным и поддерживается не во всех компиляторах и не на всех платформах. В таком случае компилятор ограничивает длину константы размером **long** (4 байта).

Посредством суффиксов (табл. 3.2) программист может предоставить компилятору дополнительную информацию о том, как следует интерпретировать указанное значение.

Таблица 3.2. Суффиксы для задания разрядности целых литералов

Суффикс	Чему соответствует	Пример представления
Если суффикс отсутствует	По приведенным выше правилам	3
L или l (от long)	long (32-разрядное целое)	3L
ll или LL i64 или I64	long long (64-разрядное целое)	3LL 3i64

В зависимости от суффиксов констант в следующих примерах получим разные результаты:

выражение `(0xFFFFFFFF + 1)` даст `0x00000000`, т. к. при сложении двух 32-разрядных значений произойдет переполнение разрядной сетки, и старший разряд результата будет потерян

выражение `(0xFFFFFFFFi64 + 1i64)` даст `0x0000000100000000`, поскольку складываться будут 64-разрядные значения и переполнения не произойдет

Также можно явно указать компилятору, каким образом он должен интерпретировать константу — как знаковую или как беззнаковую (табл. 3.3). В зависимости от этого компилятор может генерировать разные низкоуровневые команды при использовании такой константы в выражениях.

Таблица 3.3. Суффиксы для задания знакопеременности целых литералов

Суффикс	Интерпретация значения	Примеры представления
Без суффикса	Компилятор интерпретирует константу как знаковую, если ее значение помещается в диапазон знакового <code>int</code> , иначе как беззнаковую	-1 (<code>0x0FFFFFFF</code>) — знаковое 4294967295 (<code>0xFFFFFFFF</code>) — беззнаковое
U или u (от unsigned)	Компилятор интерпретирует константу как беззнаковую	255U

В зависимости от суффиксов констант в следующих примерах получим разные результаты.

Пример 1.

```
int n=31, m=3;
int res = ((1<<n)>>m); //res=0xf0000000, т. к. при сдвиге влево
                        в старшем знаковом разряде окажется единица, которая
                        при последующем сдвиге вправо знакового значения будет
                        распространена на три старших разряда
```

Пример 2.

```
int n=31, m=3;
int res = ((1u<<n)>>m); //res=0x10000000, т. к. при сдвиге вправо
                        беззнакового значения старшая часть заполняется нулями
```

ЗАМЕЧАНИЕ

Любой из суффиксов `L`, `int64`, `LL` можно комбинировать с суффиксом `U`, например: `12345678912345Ui64`.

3.2.2. Литералы с плавающей точкой

Если в литеральной записи числа содержится точка (разделитель целой и дробной части), то компилятор интерпретирует ее как константу с плавающей точкой. Стандартные форматы чисел с плавающей точкой рассмотрены в разд. П1.11 приложения.

Примеры записи литералов с плавающей точкой:

1.25
0.25
.25
1.

Примеры записи чисел с плавающей точкой в инженерном (экспоненциальном) формате:

1.25E10 (мантиссаЕпорядок)
1.25e-10

ЗАМЕЧАНИЕ 1

По умолчанию литералы с плавающей точкой являются константами типа **double**.

ЗАМЕЧАНИЕ 2

В записи литералов с плавающей точкой не должно быть пробелов! Например: 1. 25 e -10, скорее всего, будет воспринято компилятором как набор из четырех различных литералов (что вызовет синтаксическую ошибку).

В языке C/C++ имеется несколько плавающих форматов. Посредством суффиксов программист может предоставить компилятору дополнительную информацию о том, как интерпретировать значение и сколько памяти (в области кода) компилятор должен задействовать для представления значения (табл. 3.4).

Таблица 3.4. Суффиксы для задания типа плавающего литерала

Суффикс	Количество байтов для представления значения	Пример представления
Если суффикс отсутствует	8	0.25
F или f (от float)	4	0.25f или .25F
L или l (от long double)	Пока 8, в дальнейшем возможно больше	0.25l или .25L

3.2.3. Символьные литералы

Одним из самых распространенных видов данных является текстовая информация: символы и строки. Символьным литералом (символьной константой) называется последовательность, содержащая один или несколько символов и заключенная в одинарные кавычки, например:

'A'

Компилятор, встретив в тексте программы символьный литерал, заменяет его короткой двоичной кодовой комбинацией (кодом символа).

Использование символьных литералов вместо их числовых эквивалентов (непосредственно кодов символов) обеспечивает:

- облегчение восприятия текста программы;
- возможность переносимости программ на другие процессоры (вычислительные системы).

Кодирование символов

Каждому символу, используемому в компьютерных программах, во внутреннем представлении соответствует уникальное значение (двоичный код). Сколько требуется отводить памяти под такой числовой эквивалент и какое значение следует сопоставить символу, зависит от способа кодировки.

ЗАМЕЧАНИЕ

Кроме печатных знаков, необходимо еще некоторое количество кодовых комбинаций, используемых для управляющих целей (переместиться в тексте на новую строку или вернуться к началу строки и т. п.). Для того чтобы компилятор мог вставить в двоичный код программы такую кодовую комбинацию, программист должен специальным образом ее изобразить в тексте программы (см. разд. "Управляющие или escape-последовательности" данной главы).

Рассмотрим способы кодирования символов.

- В те далекие времена, когда большинство программ не поддерживали диалогового режима с пользователем, программисту для представления текстовой информации вполне хватало строчных и прописных букв латинского алфавита (даже в том случае, когда программист был не очень-то и силен в английском языке), знаков препинания, знаков арифметических действий, цифр и некоторых общеиспользуемых символов (таких, как @, #, \$ и т. д.). Для уникального представления всех перечисленных значений достаточно 7 битов. Такой вид кодировки известен как ASCII (American Standard Code for Information Interchange). При 7-битовой кодировке для представления печатных символов отводится диапазон значений от 0x20 до 0x7F (от 32₍₁₀₎ до 127₍₁₀₎). Кроме того, коды в диапазоне от 0x00 до 0x1F используются для специфических (управляющих) целей: перевод строки, возврат каретки и т. д. (см. разд. "Управляющие или escape-последовательности" данной главы).
- Проблема проявилась позже, когда оказалось, что пользователь предпочитает общаться с программой на своем родном языке. Решения для локализации приложений (как заставить приложение воспринимать ввод/вывод на разных языках) появлялись разные. Для символов нелатинских алфавитов стали использовать в рамках байтовой разрядной сетки диапазон 128—255 (например, кодировка КОИ8Р, в которой эти значения были

задействованы для кодов символов русского алфавита). Такой вид кодировки называют словосочетанием SBCS (Single Byte Character Set) — 8-битовая кодировка.

- ❑ Посредством SBCS полностью проблему решить не удалось, т. к. существуют языки (например, японский), в которых знаков гораздо больше, чем можно закодировать однобайтовыми комбинациями. Поэтому появился способ кодировки MBCS (MultiByte Character Set), при котором часть символов кодируется одним байтом, а остальные — двумя. Очевидно, что при таком способе кодировки должна быть возможность определения количества байтов, занимаемых кодом символа (значение первого байта содержит признак того, что символ кодируется двумя байтами). А для того чтобы воспользоваться таким значением, необходимо программно анализировать, сколько байтов занимает код символа. Кроме того, кодировка MBCS предполагает применение разных кодовых страниц для разных языков, а это означает, что в одной программе пользоваться одновременно несколькими языками оказывается затруднительно.
- ❑ Для унификации кодирования текстовой информации в 90-х годах XX века был разработан стандарт кодирования символов, известный под названием UNICODE, при котором любой символ кодируется двумя байтами. Среди 65 536 возможных (при двухбайтовом кодировании) кодовых комбинаций для каждого из языков был выделен свой диапазон (примеры в табл. 3.5). При таком способе кодировки ничто не мешает использованию в одной программе представления текстовой информации на нескольких языках. Для совместимости со старыми программами, диапазон кодов 0x0000...0x007F был оставлен для традиционной кодовой таблицы ASCII. Для символов кириллицы был выделен диапазон кодов 0x0400...0x4FF.

Таблица 3.5. Некоторые диапазоны представления символов для UNICODE

16-битный код	Символы
0x0000 – 0x007f	ASCII
...	...
0x0370 – 0x03ff	Греческий
0x0400 – 0x04ff	Кириллица
0x0530 – 0x058f	Армянский

Преимущества использования UNICODE по сравнению с MBCS:

- ❑ позволяет использовать в одном приложении сколько угодно разных языков (интернационализация приложений);

- увеличивает скорость выполнения прикладных программ, обрабатывающих текстовую информацию, т. к. все символы кодируются единообразно двумя байтами.

Недостаток:

- увеличивает расход памяти.

ЗАМЕЧАНИЕ

На сегодняшний день UNICODE является стандартом (полное описание приведено в документе "The Unicode Standard: WorldWide Character Encoding" издательства Addison-Wesley). Операционные системы Windows семейства NT сравнительно давно ориентированы на использование UNICODE, поэтому если вы собираетесь разрабатывать Windows-приложения, то лучше сразу привыкать использовать этот тип кодировки.

Изображение символьных констант в C/C++-программе

Желая вставить в текст программы символьную константу (символьный литерал), программист просто должен записать один или два символа, заключив их в одинарные кавычки, если он использует однобайтовое кодирование (например, вот так: 's').

Если же предполагается использование UNICODE, то перед открывающей кавычкой следует поставить префикс L (вот так: L's').

Встретив в тексте программы символьный литерал, компилятор должен заменить его соответствующим числовым эквивалентом. Этот числовой эквивалент зависит от способа кодировки (табл. 3.6).

ЗАМЕЧАНИЕ

При любом способе кодировки диапазон 0—127 (ASCII) отводится под буквы латинского алфавита, цифры и некоторые знаки @, #, \$ и т. д.

Например:

```
std::cout<<'A';    //будет выведен символ A
std::wcout<<L'A'; //для расширенных символов используется
                  соответствующий поток вывода wcout
```

ЗАМЕЧАНИЕ 1

Синтаксис языка C/C++ позволяет задать в символьном литерале сразу два символа следующим образом: 'ab'. При этом компилятор располагает коды двух указанных символов в двух смежных байтах памяти: код первого символа — по старшему адресу, код второго — по младшему: 0x6162. Рекомендуется избегать такого задания символьных литералов — это ухудшает читабельность текста программы.

Таблица 3.6. Использование символьных литералов

	Способ кодировки			Escape – последовательность
	SBCS	MBCS	UNICODE	
Пример изображения	'А' 'Ф' '1' '*'	'а' '+' 'Б' Т. к. коды всех букв русского алфавита помещаются в диапазон 128—255, то пример символа кириллицы, который будет занимать 2 байта, привести затруднительно	L'А' L'Ф'	'\a' '\n' '\"' '\\' '\08' '\0x10' '\\' L'\n' L'\\' L'\08'
Сколько занимает памяти	1 байт	1 или 2 байта	2 байта	Несмотря на внешний вид, это одиночный символ, поэтому: без префикса L — 1 байт, с префиксом — 2 байта
Что делает компилятор, встречая в тексте символьный литерал	Подставляет вместо него байт с числовым эквивалентом данного символа из набора SBCS 'а' – 0x61 'ф' – 0xd4	Подставляет вместо него один или два байта из набора MBCS: 'а' – 0x61 'ф' – 0xd4	Подставляет значение из набора UNICODE L'ф' – 0x424	Подставляет значение в соответствии с табл. 3.7. '\n' – 0x0A (10) L'\n' – 0x000A (10)

ЗАМЕЧАНИЕ 2

В некоторых языках коды, которые в ASCII соответствуют специальным символам (#, [, {, }, |...), заняты кодами букв, которых нет в английском алфавите (Å, Æ...). Если в таком случае требуется вывести специальный символ, то используются *триграфы*. Например: если код символа '#' в данном языке занят, то для вывода # можно использовать триграф '??='.

Управляющие или escape-последовательности

Есть несколько особых кодовых комбинаций:

- которым не соответствуют отображаемые символы (перевод строки, звуковой сигнал...). Такой код является управляющим для устройства, т. е. заставляет устройство выполнять некоторые специфические действия;
- изображение отсутствует на клавиатуре, а отображать хочется (♥, ☺, ♣...);
- отображаемый символ используется компилятором в своих целях (" , \...).

Так как такие кодовые комбинации имеют специальное назначение, то они в тексте программы записываются последовательностью символов, начинающейся с символа обратной косой черты \. Несмотря на свой внешний вид, каждая такая последовательность на самом деле соответствует одной кодовой комбинации. Примеры часто используемых escape-последовательностей приведены в табл. 3.7.

Таблица 3.7. Управляющие (escape) последовательности

Символьное представление	Шестнадцатеричный код	Что делает компилятор
'\a'	0x07	Звуковой сигнал
'\b'	0x08	Возврат на знакоместо назад
'\f'	0x0C	Перевод страницы
'\n'	0x0A	Перевод строки
'\r'	0x0D	Возврат каретки
'\t'	0x09	Горизонтальная табуляция
'\v'	0x0B	Вертикальная табуляция
'\\'	0x5C	Символ обратной косой черты
'\''	0x27	Символ апострофа
'\"'	0x22	Символ кавычки
'\?'	0x3F	Символ вопросительного знака
'\0101' (явное задание кода символа в восьмеричной системе)	0x41	Код символа A Эквивалентно 'A'
'\x42' (явное задание кода символа в шестнадцатеричной системе)	0x42	Код символа B Эквивалентно 'B'
'\0'	0x00	Нулевое значение байта

Примеры (эти фрагменты программы печатают на экране):

```
std::cout<<'\\x41'<<'\\'<<'\\n'; //будет напечатано: A\
и произойдет переход на новую строку
std::cout<<'\\3'; //в MS_DOS или Win32 Console приложении будет
напечатан символ ♥.В других ОС ASCII-коду 3 может
соответствовать другое графическое изображение
или оно может вовсе отсутствовать.
```

ЗАМЕЧАНИЕ

За символом `\\` могут следовать только предусмотренные компилятором символы, иначе результат интерпретации такой escape-последовательности не определен. Например, компилятором VC для такой непредусмотренной комбинации будет выдано предупреждение, а символ `\\` будет просто проигнорирован.

3.2.4. Строковые литералы

Строковый литерал — это заключенная в двойные кавычки последовательность символов:

"Пример строковой константы"

Специфика:

- ❑ строковый литерал — это массив (тип такого массива — **char[]**), в котором компилятор хранит коды указанных программистом символов;
- ❑ следует помнить, что компилятор при генерации такого массива автоматически добавляет завершающий нулевой байт, т. к. такое нулевое значение в C/C++ является признаком конца строки;
- ❑ компилятор отводит память для хранения такого массива в той же области, в которой находится код программы, поэтому в защищенном режиме (для процессоров семейства x86) эта область оказывается аппаратно защищенной от записи. Во время выполнения программы попытка модификации такой области памяти вызовет срабатывание аппаратного механизма защиты и передачу управления операционной системе;
- ❑ строка может быть пустой ("") — эта запись означает, что строковый литерал состоит из единственного нулевого байта;
- ❑ можно представить одну строку в виде совокупности строк (конкатенация строковых литералов) следующим образом:

```
"12" "34" "56" эквивалентно "123456" //компилятор воспринимает
такую запись как один строковый литерал. В конце
массива добавляется один завершающий ноль;
```

- строка с префиксом `L` (например, `L"My wide chars"`) является строкой символов, закодированных двухбайтовыми кодами UNICODE (ее тип: `wchar_t[]`):

```
std::wcout<<L"My wide chars"; //для вывода расширенного строкового
                               литерала используется соответствующий
                               поток расширенного вывода
```

- если строка оказывается слишком длинной, а программисту не хочется прокручивать окно редактирования, то для переноса остатка строкового литерала на следующую строку в тексте программы нужно использовать обратный слеш `\`. Компилятор этот обратный слеш проигнорирует и воспримет такую запись как одну строку.



Что будет напечатано?

```
std::cout<<"ABC\
    DFG";
```

- для того чтобы в строку вставить символ `"` (двойную кавычку) или `\` (обратный слеш) — эти символы являются служебными для компилятора, нужно использовать *escape*-последовательности.



Что будет напечатано?

```
std::cout<<"A\"BC\\D";
```

3.3. Перечисление *enum*

В практике программирования встречаются случаи, когда выражение может принимать значения только из заранее определенного конечного множества значений. Для таких задач язык Си предоставляет программисту способ задания именованных целочисленных констант: **enum** (перечисление).

Это средство облегчает жизнь программисту следующим образом: вместо того, чтобы помнить и использовать в тексте программы числовые значения, можно один раз сопоставить каждой константе имя и далее пользоваться этими именами (которые запомнить гораздо легче), а компилятор сам в каждом нужном месте просто подставит значение соответствующей константы.

Синтаксис:

```
enum [имя_пользовательского_типа] {список_именованных_констант};
```

Пример 1.

Допустим, что программист должен предоставить пользователю выбор из трех вариантов действий, а программа, соответственно, должна в каждом случае отреагировать по-разному.

Это можно было бы решить следующим образом: каждому варианту сопоставить значение: 0, 1, 2. При этом программист должен помнить, какому варианту действий программы соответствует каждое абсолютное значение (листинг 3.1).

Листинг 3.1. Использование абсолютных значений для обозначения констант

```
{
    int n; //в этой переменной будет сформирован выбор пользователя
    //Формирование значения переменной n
    if(n == 0){...} //если значение переменной совпадает с нулем,
                    то пользователь может продолжить выполнение
                    программы (при этом программист должен помнить,
                    чему соответствует этот ноль)
}
```

Гораздо удобнее для этих же целей ввести именованные константы, а имя подскажет, чего хочет пользователь (листинг 3.2).

Листинг 3.2. Использование enum для обозначения констант

```
{
    enum{CONTINUE, CANCEL, RETRY}; //встретив такое объявление,
    компилятор запомнит указанный в скобках список
    имен и сопоставит каждому имени значение
    по умолчанию: 0, 1, 2. Позже, встретив в тексте
    программы CONTINUE, компилятор подставит 0,
    вместо CANCEL подставит 1, вместо RETRY – 2
    //или enum{CONTINUE=-1, CANCEL=0, RETRY=1}; //значения констант
    определены программистом явно
    //или enum{CONTINUE=-1, CANCEL, RETRY}; //значения констант
    будут по умолчанию сформированы следующим образом:
    CONTINUE соответствует -1, а дальше компилятор
    формирует значение очередной константы, прибавляя
    единицу, т. е. CANCEL=0, RETRY=1 (пока снова не
    встретит явный инициализатор)

    int n; //в этой переменной будет сформирован выбор пользователя
    //Формирование значения переменной n
    if(n == CONTINUE){...} //гораздо понятнее
}
```

ЗАМЕЧАНИЕ

Именованными константами, определенными внутри блока, можно пользоваться до закрывающей скобки блока.

Пример 2.

С помощью перечисления можно ввести свой пользовательский тип и далее создавать и использовать переменные такого типа (листинг 3.3).

ЗАМЕЧАНИЕ

Переменные типа **enum** аналогичны другим целочисленным переменным, с той лишь разницей, что без принудительного приведения типа (см. разд. 3.4.5) компилятор позволит присваивать им значения только посредством именованных констант, указанных программистом в списке инициализаторов.

Листинг 3.3. Использование переменных типа enum

```
{  
    enum REASON{ OK, CANCEL, RETRY}; //REASON является  
                                     пользовательским типом.  
//в C++  
    REASON res; //объявление переменной типа REASON подсказывает  
                программисту и компилятору, как предполагается  
                использовать переменную res  
//в языке Си обязательно требуется уточнить компилятору посредством  
                ключевого слова enum, что REASON является  
                перечислением  
    enum REASON res1;  
    res= CANCEL; //корректно – переменная res будет содержать 1  
    res = OK;    //корректно, присвоили res значение 0  
    //OK = 4;    //ошибка, т. к. OK – это константа  
    //res = 0;   //ошибка, несмотря на то, что значение OK=0,  
                поскольку переменной такого типа можно присвоить  
                значение только посредством именованной константы.  
                Чтобы компилятор не выдавал ошибки, требуется явное  
                приведение типа (см.разд. 3.4.5)  
}
```

ЗАМЕЧАНИЕ 1

При вычислении выражений вместо именованных констант компилятор подставляет соответствующие целые значения.



Проинтерпретируйте результат приведенных выражений:

```
int sum = OK + CANCEL + RETRY;
int m = 10;
if(m> OK){...}
int n= OK;
```

ЗАМЕЧАНИЕ 2

В стандартном Си тип данных **enum** эквивалентен знаковому целому (**signed int**). Стандарт языка C++ разрешает программисту самому указывать один из целых типов, на базе которого компилятор создает перечисление: **char**, **short**, **int** или **long**, а также можно уточнить знаковость перечисления посредством **signed** или **unsigned**: **enum REASON : unsigned char {OK, CANCEL, RETRY};**

ЗАМЕЧАНИЕ 3

В списке инициализации значения могут повторяться. Встретив очередной явный инициализатор, компилятор по умолчанию формирует значение следующей именованной константы, прибавляя единицу:

```
enum{a=1, b, c, A=1, B, C}; //a=1, b=2, c=3, A=1, B=2, C=3
```

или

```
enum{a=1, b, c, A=a, B, C}; //результат будет таким же
```

3.4. Переменные

Если значения констант программист должен указать компилятору непосредственно в тексте программы, то значения переменных вычисляются в процессе выполнения программы, поэтому эти два вида данных имеют принципиальные отличия (табл. 3.8).

Таблица 3.8. Сравнение констант и переменных

Константы	Переменные
Тип константы распознается компилятором по ее написанию в тексте программы: по умолчанию или по использованным программистом префиксам и суффиксам	Тип переменной программист должен указать компилятору явно

Таблица 3.8 (окончание)

Константы	Переменные
Встретив в тексте программы значение константы, компилятор помещает указанное значение непосредственно в тело процессорной команды	Переменные — это поименованные программистом области памяти для хранения значений. Встретив в тексте программы имя переменной, компилятор на самом деле генерирует процессорную команду, которая обращается к ассоциированной с этим именем области памяти
Константы можно использовать в выражениях только справа от знака равенства (модифицировать их просто невозможно)	Переменные являются lvalue. Их можно использовать как справа от знака равенства, так и слева, за исключением переменных, объявленных с ключевым словом const (см. разд. 3.10.1)

3.4.1. Что такое тип переменной

Программист пишет текст программы, оперируя понятиями языка высокого уровня. Для того чтобы программа могла выполняться, написанный текст должен быть преобразован компилятором в двоичный код, с которым может работать процессор. При этом один и тот же текст на языке высокого уровня (одни и те же действия с данными разного типа) компилятор превращает в разные последовательности низкоуровневых команд:

```
int x1=1, y1=2, z1;
z1 = x1 + y1; //компилятор сгенерирует низкоуровневую команду
              целочисленного сложения — add

double x2=1.2, y2=2.33, z2;
z2 = x2 + y2; //компилятор сгенерирует низкоуровневую команду
              плавающего сложения — fadd
```

На рис. 3.2 представлено разделение обязанностей между программистом и компилятором для получения результирующего двоичного кода. Роль компилятора:

- выделить нужное количество памяти для хранения значения переменной;
- распознать, в каком виде использует данные программист, и перевести их в двоичное представление, которое понимает процессор;
- при выполнении действий с переменными разного типа сгенерировать разные низкоуровневые команды.

ЗАМЕЧАНИЕ

Программисту удобнее в отдельных случаях по-разному задавать значения, которые в двоичном виде будут выглядеть одинаково.

Разделение обязанностей для получения двоичного кода

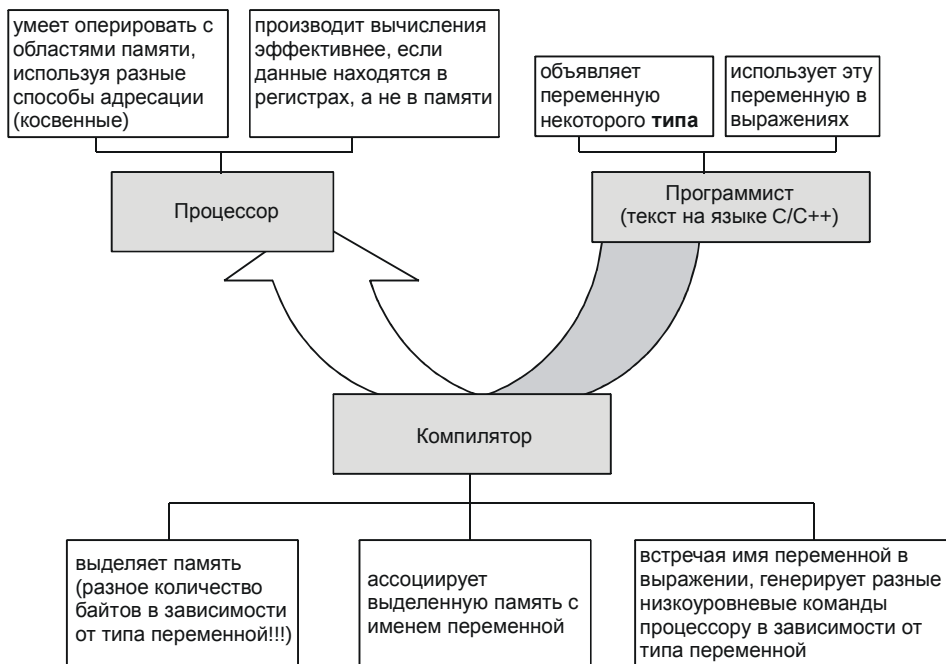


Рис. 3.2

Следующие, разные с точки зрения программиста, представления данных для процессора выглядят одинаково:

```

char c = 'A'; //посредством символьного литерала в памяти будет
               храниться 0100 0001 - двоичный код символа A
c = 65;       //или то же самое значение, заданное программистом
               в десятичной системе счисления
c = 0x41;     //то же самое - в шестнадцатеричной системе
c = 0101;     //то же самое - в восьмеричной системе
  
```

3.4.2. Фундаментальные (базовые, встроенные) типы C/C++

Основные потребности программиста отражены в фундаментальных типах C/C++ (рис. 3.3). Эти типы поддерживаются большинством современных компьютеров на аппаратном уровне (имеется в виду организация памяти и набор машинных инструкций для операций над данными фундаментальных

типов). Если на аппаратном уровне поддержка не реализована, она осуществляется программно.

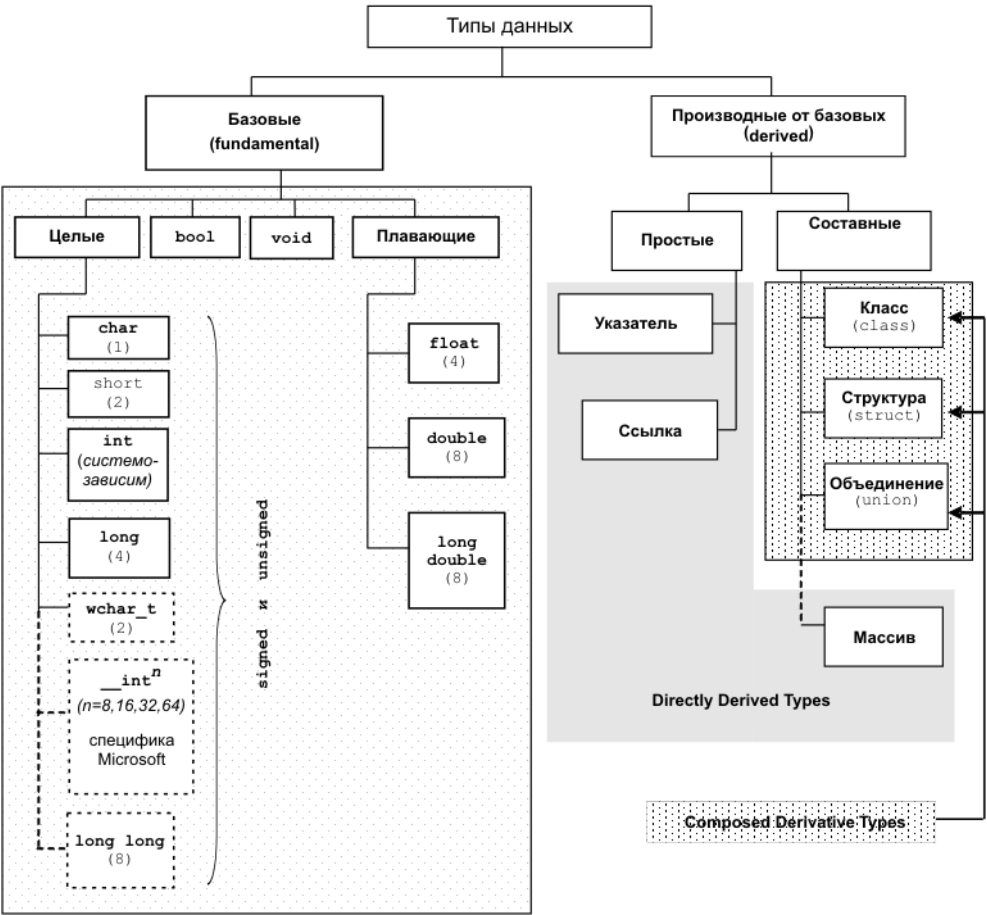


Рис. 3.3

ЗАМЕЧАНИЕ 1

Если в программе используются только встроенные типы данных, компилятор без каких-либо дополнительных объяснений программиста сам знает, какие низкоуровневые инструкции генерировать! На то они и встроенные!

ЗАМЕЧАНИЕ 2

Пунктиром на рис. 3.3 отмечены фундаментальные типы C++, которых в стандарте нет, но многие компиляторы так или иначе такие типы поддерживают (например, VC).

3.4.3. Оператор *sizeof* и размеры переменных

Для определения размера переменной служит оператор **sizeof**. Результатом является количество байтов, занимаемых переменной (или выражением).

ЗАМЕЧАНИЕ 1

Результат оператора **sizeof** вычисляется на этапе трансляции компилятором, а не во время выполнения программы.

ЗАМЕЧАНИЕ 2

Для обозначения типа результата, вычисляемого оператором **sizeof**, в заголовочном файле `<stddef>` (в нестандартизованных версиях — `<stddef.h>`) введен специфический тип **size_t**, который на самом деле является синонимом **unsigned int**.

Формы оператора **sizeof** (синтаксис):

```
size_t num = sizeof(выражение); //скобки могут быть опущены
size_t num = sizeof(тип);        //скобки обязательны. Тип может
                                быть как базовым, так и пользовательским
```

Специфика:

```
sizeof (char) == 1 //по определению
sizeof (массива) == количество байтов, занимаемых массивом
sizeof (class, struct, или union) == количество байтов, занимаемых
                                объектом пользовательского типа, + затраты
                                на выравнивание см. разд. 9.11 и 10.1)
```

Например:

```
double d;
size_t n = sizeof( d ); //8

n = sizeof( int ); //2 или 4 в зависимости от типа процессора
n = sizeof( имя_массива ); //количество байтов, занимаемых
                                массивом, причем элементы массива
                                могут быть любого типа
```



Подумайте, чему будет равен размер переменной `ld`?

```
long double ld;
n= sizeof( ld );
```

ЗАМЕЧАНИЕ

В общем случае по поводу размеров фундаментальных типов гарантируются только следующие соотношения (Б. Страуструп):

```
1==sizeof(char)<=sizeof(short)<=sizeof(int)<=sizeof(long)
sizeof(float)<=sizeof(double)<= sizeof(long double)
```

Свойства (размеры) базовых типов (numerical limits) в зависимости от реализации

Иногда в программе требуется учитывать свойства фундаментальных типов, которые зависят от реализации. Например, размер переменной типа `int` или то, каким образом преобразуется длинный плавающий формат к целому формату (округляется или обрезается) и т. д. Такие особенности лучше учесть при написании программы, нежели потом искать возникающие нетривиальные ошибки во время выполнения.

Одним из способов борьбы с особенностями реализации является использование стандартной библиотеки, когда это возможно. Соответствующие средства стандартной библиотеки реализованы посредством *шаблонов* и сосредоточены в заголовочном файле `<limits>`. А т. к. речь о заголовочных файлах (см. разд. 5.6) и шаблонах пойдет позже, просто приведем примеры, как этими средствами можно пользоваться.

Пример 1.

Пусть требуется определить, можно ли присвоить значение, хранящееся в переменной типа `int`, переменной меньшего размера `char` без потери значения.

Способ 1 — без использования средств стандартной библиотеки (листинг 3.4).

Листинг 3.4. Проверка диапазона значений `char` без использования средств стандартной библиотеки

```
{
    int n=значение; //это значение может лежать в пределах диапазона
                    //изменения char, а может и выходить за пределы,
                    //поэтому следует проверить: не потеряем ли мы
                    //значение при присваивании

    char c;
    if( (n>=-128) && (n<=127)) c=n; //присваивание будет произведено
                                    //без потери значения
    else...                          //иначе значение переменной n не может
                                    //храниться в c без потери значения
}
```

Способ 2 — с использованием средств стандартной библиотеки.

В стандартной библиотеке имеется пара функций, которые возвращают минимальное и максимальное значения типа `char`:

❑ `numeric_limits<char>::min();`

❑ `numeric_limits<char>::max();`

Листинг 3.5 демонстрирует способ их использования.

Листинг 3.5. Использование средств стандартной библиотеки для определения диапазона значений char

```
#include <limits>

{
    int n=значение;
    char c;
    if( (n >= numeric_limits<char>::min() ) &&
        (n <= numeric_limits<char>::max() ) c=n; // без потери
                                                    значения
}
```

Пример 2.

На самом деле стандартная библиотека предоставляет много других полезных средств для получения информации о базовых типах (листинг 3.6).

Листинг 3.6. Примеры использования средств стандартной библиотеки для определения свойств встроенных типов

```
{
    //сколько значащих цифр соответствует каждому из базовых типов:
    n = numeric_limits<char>::digits; //7 значащих двоичных цифр
    n = numeric_limits<unsigned char>::digits; //8 значащих цифр
    //как округляются числа в плавающем формате при приведении
    к целому типу:
    if(numeric_limits<double>::round_style==round_to_nearest)
        //округляется до ближайшего целого
    //...
}
```

3.4.4. Знаковость переменной

Когда мы объявляем знаковую или беззнаковую переменную, то указываем компилятору:

- сколько зарезервировать памяти, при этом знаковость переменной никак не проявляется — это только наше представление (интерпретация про-

граммистом) того значения, которое лежит по ассоциированному месту в памяти. Например, если мы знаем, что диапазон изменения данных не более чем от -128 до $+127$ (всего 256 возможных значений), тогда логично для хранения таких данных завести знаковую переменную типа **char**. А если диапазон изменения данных от 0 до 255 (возможных значений тоже 256), то можно использовать переменную типа **unsigned char**;

- какие команды низкого уровня (разные!) компилятор должен генерировать при операциях с этой переменной, теперь знаковость начинает влиять.

Обратите внимание на то, что двоичные представления для разных значений переменных: знаковой x и беззнаковой y в двоичном виде могут быть абсолютно одинаковыми (см. *разд. П1.5 приложения*). А результат сравнения двух значений получается разный, т. к. компилятор, исходя из типа переменных, генерирует разные низкоуровневые команды (листинги 3.7 и 3.8, в которых соответствующие строки на ассемблере приведены для VC.NET 2005).

Листинг 3.7. Переменные объявлены как знаковые

```
{
    int x=0x1; //Компилятор сгенерирует:
                mov dword ptr[x],000000001h
    int y=0xffffffff; //представление -1. Компилятор сгенерирует:
                mov dword ptr[y],0FFFFFFFFh

    if(x>y){...} //должны получить true, т. к. 1>-1. Компилятор
                сгенерирует:
                mov eax,dword ptr[x]
                cmp eax,dword ptr[y] ; команда сравнения изменяет флаги
                jle <адрес_перехода> ; переход, если меньше или равно
                                с учетом знака
}
```

Листинг 3.8. Переменные объявлены как беззнаковые, а значения в них те же самые

```
{
    unsigned int x=0x1; //Компилятор сгенерирует:
                mov dword ptr [x],000000001h
    unsigned int y=0xffffffff; //представление значения 4294967295.
```

Компилятор сгенерирует:

```
mov dword ptr [y],0FFFFFFFFh
```

```
if(x>y){...} //должны получить false, поскольку 1<4294967295.
```

Компилятор сгенерирует:

```
mov eax,dword ptr[x]
```

```
cmp eax,dword ptr[y] ; команда сравнения изменяет флаги
```

```
jbe <адрес_перехода> ; переход, если меньше или равно  
без учета знака
```

```
}
```

3.4.5. Приведение типов

В приведенных в предыдущем разделе примерах операнды, участвующие в выражении, были одного типа. На практике в любом выражении могут участвовать переменные и литералы разных типов.

Проблемы:

- ☐ процессор не умеет работать с операндами разного типа, поэтому прежде чем производить вычисления, компилятор должен все значения привести к одному типу, например:

```
char a=1;
```

```
double b=2.2;
```

```
double c = a+b; //прежде чем вычислять сумму, компилятор должен  
привести значение, хранящееся в байтовой  
переменной c к типу double
```

- ☐ наиболее эффективно вычисления выполняются, если значения находятся в регистрах;
- ☐ смысл выражения с точки зрения программиста не всегда очевиден, а компилятор всегда должен действовать однозначно, поэтому существуют правила, которым в таких ситуациях следует компилятор.



В примерах, приведенных в табл. 3.9, исходные значения переменных одинаковы, но результат будет разным. Попробуйте предсказать их в каждом случае, а затем сравните ваши предположения с правилами (по которым будет действовать компилятор), изложенными в следующем разделе.

Таблица 3.9. Задания для проверки интуиции

Номер	Попробуйте предсказать результат в каждом случае
1	<pre>char c1=0xff; unsigned char c2 = 0xff; if(c1==c2){...}</pre>
2	<pre>int n=0xff; char c = 0xff; if(c==n){...}</pre>
3	<pre>int n=0xff; unsigned char c = 0xff; if(c==n){...}</pre>

Неявное приведение типов (компилятором)

Для того чтобы компилятор мог вычислить выражение, операнды должны быть одинакового типа, поэтому перед вычислением выражения компилятор модифицирует операнды таким образом, чтобы они были одного и того же типа, но (важно!) сохранили свое значение (операции приведения типов на уровне двоичного кода обсуждаются в *разд. П1.10 приложения*). При этом для того, чтобы не потерять точность, компилятор приводит все значения к старшему типу (о понятии старшинства типов см. *разд. П1.12 приложения*).

Если программист не указывает компилятору явно, как ему делать преобразования при использовании в выражении переменных разных типов, то компилятор в ходе вычисления перед выполнением действия делает внутренние преобразования сам, т. е. осуществляет неявное приведение типов.

Правила:

- сначала вычисляется выражение справа от знака равенства. Важно! При вычислении выражения справа от знака равенства значения преобразуются всегда к старшему типу:
 - компилятор разбивает выражение на подвыражения;
 - каждое подвыражение обрабатывается в соответствии со следующими правилами:
 - ◇ если в выражении используются короткие целые, то компилятор приводит их к размеру регистра с учетом знаковости;
 - ◇ если в подвыражении фигурируют плавающие или **long**, все операнды приводятся к старшему типу;
 - ◇ вычисляется значение подвыражения;

- все, сказанное о вычислении подвыражений, применяется для вычисления выражения в целом;
- полученное в результате вычисления выражения значение приводится к типу слева от знака равенства.

ЗАМЕЧАНИЕ

При выполнении операции присваивания результат приводится к типу выражения слева от знака равенства. В этом случае может возникнуть ситуация преобразования старшего типа к младшему, и, соответственно, потеря точности или даже возникновение грубой ошибки, если значение старшего типа не помещается в разрядную сетку младшего (см. *разд. П1.12 приложения*).

Для того чтобы не получать странные результаты или не потерять точность при вычислении выражений, программист должен знать правила, по которым компилятор осуществляет неявные преобразования. И, если действия компилятора не соответствуют замыслу программиста, необходимо уточнить компилятору смысл посредством явного приведения типов (см. *следующий раздел*).

Особенности приведения целых типов:

- при преобразовании знаковых типов (младшего целого типа к старшему целому типу) старшие биты могут заполняться либо нулями, либо значением знакового (старшего) бита преобразуемого значения (такое преобразование называется распространением знака — *sign extention*);

ПРАВИЛО

Общее правило состоит в следующем: при преобразовании младшего знакового типа к старшему (знаковому или беззнаковому) типу расширенные старшие биты заполняются значением знакового разряда преобразуемого значения; при преобразовании младшего беззнакового типа к старшему (знаковому или беззнаковому) типу расширенные старшие биты всегда заполняются нулями.

- при вычислении выражения целые типы (короче, чем `int`) сначала приводятся к размеру регистра с учетом их знаковости (это сделано для повышения эффективности вычислений, т. к. оптимизирующий компилятор старается все операции выполнять в регистрах процессора);
- преобразование целых типов в типы с плавающей точкой выполняются с помощью команд сопроцессора (или специальными библиотечными процедурами) с заполнением свободных правых битов мантиссы нулями;
- преобразование старших целых типов в младшие выполняется отбрасыванием старших байтов. При этом возможно возникновение грубой ошибки.

ЗАМЕЧАНИЕ

При преобразовании старшего целого типа к младшему типу, результат корректен только тогда, когда целая часть преобразуемого значения помещается в диапазон представления для более короткого целого типа. Компилятор при таких потенциально опасных преобразованиях выдает предупреждение о возможной потере значения.

Особенности приведения плавающих типов:

- ❑ преобразование **double** к **float** выполняется специальными командами сопроцессора с округлением числа до нужного количества знаков в мантиссе;
- ❑ преобразование типа с плавающей точкой в целый тип выполняется командами сопроцессора или специальными библиотечными процедурами и дает ближайшее целое число в соответствии с используемым правилом округления (см. разд. П1.10 и П1.12 приложения).

Пример 1.

```
int x=1;
double res1 = x/2; //так как в выражении участвуют только целые
                    типы (целая переменная x и целый литерал 2), то
                    компилятор сгенерирует низкоуровневую команду
                    целочисленного деления idiv, при выполнении которой
                    дробная часть просто не формируется, поэтому,
                    несмотря на то, что тип res1 – плавающий,
                    значение res1 = 0
double res2 = x/2.; //а в этом случае второй операнд выражения –
                    плавающий литерал, поэтому компилятор сгенерирует
                    совершенно другую низкоуровневую команду плавающего
                    деления fdiv и res2 = 0.5
```



В целях закрепления материала примените правила неявного приведения типа при вычислении выражений (табл. 3.10) и проинтерпретируйте результат.

Пример 2.

Для повышения эффективности вычислений компилятор предпочитает помещать значения операндов в регистры. Следовательно, если в выражении используются короткие целые (и не используются плавающие типы или **long**), компилятор должен преобразовать значение каждого короткого целого к размеру регистра (т. е. представить то же самое значение в другой разрядной сетке).

Таблица 3.10. Задания для закрепления материала

Номер	Проинтерпретируйте результат
1	<code>int x=1;</code> <code>double y=0.1;</code> <code>double dres = y + x/2; //объясните результат</code>
2	<code>int x=1;</code> <code>double y=0.1;</code> <code>int ires = y + x*0.1; //объясните результат</code>
3	<code>int x=10;</code> <code>if(1/x >0)... //объясните результат</code>

Таблица 3.11. Как компилятор поступает с беззнаковыми короткими целыми

Исходные значения	Преобразованные значения (на самом деле будут складываться)	Результат
<code>c1 = 0xFF</code>	<code>0x000000FF</code>	<code>0x00000100</code>
<code>c2 = 0x01</code>	<code>0x00000001</code>	

Если короткие целые беззнаковые, то компилятор преобразует хранящиеся в таких операндах значения к размеру регистра посредством заполнения старших разрядов нулями. Это означает, что число 255, которое было представлено в байтовой разрядной сетке значением 0xff, в четырехбайтовой разрядной сетке должно превратиться в 0x000000ff (табл. 3.11).

```
unsigned char c1=0xff, c2=0x1; // c1=255, c2=1
int sum = c1+c2; //
```

Пример 3.

Пусть имеем выражение такое же, как в предыдущем примере. Внутренние представления значений переменных c1 и c2 тоже такие же. Единственным и существенным отличием является то, что они объявлены как знаковые. Перед вычислением выражения компилятор преобразует значения, хранящиеся в c1 и c2, к размеру регистра (табл. 3.12), а это означает, что число -1, которое было представлено в байтовой разрядной сетке значением 0xff, в четырехбайтовой разрядной сетке должно превратиться в 0xffffffff. Компилятор сохраняет значение знаковой переменной, распространяя значение старшего знакового разряда на всю старшую часть (три старших байта)

посредством специальных низкоуровневых команд. Такая операция носит название *sign extension* (расширение знака).

```
char c1=0xff, c2=0x1; //c1=-1, c2=1
int sum = c1+c2;
```

Таблица 3.12. Как компилятор поступает со знаковыми короткими целыми

Исходные значения	Преобразованные значения (на самом деле будут складываться)	Результат
c1 = 0xFF	0xFFFFFFFF	0x00000000
c2 = 0x01	0x00000001	

Пример 4.

Посмотрим, как поступает компилятор, если в выражении используются как знаковые, так и беззнаковые операнды.

ПРЕДУПРЕЖДЕНИЕ

Не следует смешивать в операторах сравнения знаковые и беззнаковые операнды. Компилятор в таких ситуациях обычно выдает предупреждение, и согласно своим правилам сначала интерпретирует значения как беззнаковые, а потом вычисляет выражение

```
int x=-1; //внутреннее представление - 0xffffffff
unsigned int y=1; //внутреннее представление - 0x00000001
if(x<y){...} //сначала компилятор проинтерпретирует значение
//переменной x как беззнаковое,
//а потом сравнит два
//значения без учета знака.
//Результатом сравнения будет false
```



```
int a=-1;
unsigned int b=4294967295; //внутреннее
//представление - 0xffffffff
if(a==b){...} //какое значение примет выражение
//в данном случае?
```

Явное приведение типа (программистом)

Если программиста не устраивает результат неявных преобразований, то имеется возможность явно указать компилятору, как осуществлять преобразование. При явном приведении типов вся ответственность за результат ложится на программиста, который должен учитывать перечисленные правила для простых типов данных, а также особенности приведения типов для указателей.

Не рекомендуется без оправданной необходимости (особенно начинающим программистам) пользоваться явным преобразованием типа (Б. Страуструп). Необходимость возникает, например, в ситуациях, когда без явного приведения типа теряется точность вычисления выражений.

Явные преобразования типа в стиле Си:

```
int x=1, y=2;
double z = (double)x/y; //явное указание компилятору при
                          вычислении выражения привести значение целой
                          переменной x к типу double
```

ЗАМЕЧАНИЕ

Достаточно явно привести к старшему типу только один из операндов. Второй операнд будет приведен к старшему типу по правилам неявного приведения типов компилятором при выполнении операции деления.

Явные преобразования типа в стиле Си стали нежелательными после введения операторов приведения в новом стиле в C++, поэтому там, где необходимо явное преобразование, рекомендуется пользоваться новыми операторами: **static_cast**, **reinterpret_cast**, **const_cast**, **dynamic_cast** или их сочетаниями (табл. 3.13).

Таблица 3.13. Новые операторы явного приведения типа

static_ cast <type> (выражение)	reinterpret_ cast <type> (выражение)	const_ cast <type> (выражение)	dynamic_ cast <type> (выражение)
механизм времени компиляции (преобразование осуществляется на этапе компиляции)			механизм времени выполнения
преобразование с проверкой корректности преобразования во время компиляции	преобразование без проверки	константное преобразование	преобразование с проверкой во время выполнения
осуществляет преобразование базовых типов, void-указателей и указателей для классов, связанных наследованием	осуществляет преобразование не связанных между собой типов	аннулирует или, наоборот, назначает действие модификаторов const и volatile	осуществляет преобразование указателей (ссылок) для классов, связанных наследованием

ЗАМЕЧАНИЕ 1

Несмотря на то что использование оператора **static_cast** отнимает у программиста больше сил, чем использование старого оператора явного приведения типа (писать приходится больше), все же предпочтительнее использование нового оператора **static_cast**, т. к. при этом компилятор следит не только за возможностью, но и за безопасностью приведения типа. И если с точки зрения компилятора приведение небезопасно, он выдает ошибку. Таким образом, повышается надежность программного обеспечения.

ЗАМЕЧАНИЕ 2

Операторами **const_cast** и **reinterpret_cast** злоупотреблять не стоит. Эти операторы позволяют отменить ограничения, что, в свою очередь, снижает надежность программного обеспечения (т. е. компилятор в такой ситуации просто "умывает руки"). Применять их можно, только отчетливо представляя последствия такого преобразования (см. разд. 6.1.9).

Примеры применения операторов явного преобразования.

Пример 1.

Следующие выражения на первый взгляд выглядят одинаково, но результат получается разный:

```
int x=1;
double res1 = x/2; //т. к. оба операнда в выражении целого типа,
                    компилятор генерирует низкоуровневую команду
                    целочисленного деления idiv, в результате выполнения
                    которой дробная часть просто нигде не сохраняется,
                    а потом целый результат выражения (0) приводится к
                    типу double, поэтому res1=0

double res2 = static_cast<double>(x)/2; //тип переменной x явно
                    приводится к типу double. Так как в выражении
                    появился операнд старшего плавающего типа, второй
                    операнд будет приведен к тому же типу компилятором
                    неявно и деление будет сгенерировано плавающее -
                    fdiv, поэтому res2=0.5
```



Подумайте, почему значение **res3** будет отличаться от **res2**?

```
double res3 = static_cast<double>(x/2);
```

Пример 2.

```
char x=-1; //0xff
int y = x; //значение короткой целой переменной x будет приведено
            с учетом знаковости к размеру операнда y. В этом
            случае y=0xffffffff (представление -1)
```

```
int z = static_cast<unsigned char>(x); //а в этом примере мы  
заставили компилятор проинтерпретировать значение  
переменной x как беззнаковое, поэтому компилятор  
дополнит старшую часть нулями и в результате  
y=0x000000ff (представление 255)
```



Используйте операторы явного приведения типа (а также побитовые операторы и операторы сдвига) для того, чтобы поменять местами младший и старший байты знаковой целочисленной переменной

```
int x = значение;  
x = ... //напишите выражение
```

3.4.6. Тип *wchar_t*

Тип **char** используется для хранения ASCII кодов символов в диапазоне 0—127, а при 8-битовой кодировке коды в диапазоне 128—255 могут использоваться для локализации приложений. С появлением UNICODE-кодировки символов появилась потребность хранить расширенные (двухбайтовые) коды символов, поэтому был введен новый тип: **wchar_t**.

Пример.

```
wchar_t cw = L'ф'; //компилятор зарезервирует для переменной cw  
2 байта и занесет туда значение 0x424  
(из диапазона кириллицы)
```

```
size_t n = sizeof(cw); //n=2
```

```
cw = L'\0'; //какое значение примет переменная cw?
```

```
wchar_t cw1 = 'ф'; //какое значение примет переменная cw1?
```

Подсказка: символьный литерал 'ф' будет интерпретироваться компилятором как **signed char**.



ЗАМЕЧАНИЕ

Не во всех реализациях **wchar_t** по умолчанию является встроенным типом. Это означает, что без подключения заголовочных файлов компилятор выдаст ошибку. В этом случае можно подключить заголовочный файл стандартной библиотеки `<string>`, в котором типу **wchar_t** сопоставлен синоним **unsigned short**.

3.4.7. Тип *bool* и *BOOL*

Переменные логического типа используются для хранения результата вычисления логического выражения и принимают одно из двух значений: **true** (соответствует 1) или **false** (соответствует 0). В соответствии с новым стандар-

том C++ тип **bool** является встроенным типом, размером 1 байт, а для обозначения логических значений используются ключевые слова **true** и **false**.

Примеры.

```
bool b = false; //переменной b будет присвоено нулевое значение
int x=-1, y=1;
b = (x<=y); //т. к. -1<1, b=true (а на самом деле компилятор
           занесет в переменную b 1)
b=5; //т. к. выражение справа от знака равенства !=0 (что
      соответствует true), b=true (а в переменную b
      компилятор все равно занесет 1)
int n=b + 1; //при использовании логических переменных в
            выражениях, компилятор вместо true подставляет 1,
            а вместо false - 0. Поэтому n = 1+1 = 2
```

ЗАМЕЧАНИЕ

В Си и ранних не стандартизованных версиях C++ встроенного логического типа не было, а вместо него в заголовочных файлах стандартной библиотеки был определен тип **BOOL** как псевдоним типа **int** (см. разд. 3.5.1 — ключевое слово **typedef**), а соответствующие значения **TRUE** и **FALSE** там же были определены посредством макроподстановок (см. разд. 5.2.1 — директива препроцессора **#define**).

В C++ рекомендуется пользоваться встроенным типом **bool**, хотя в большинстве случаев эти понятия являются взаимозаменяемыми. Основное отличие, которое может в некоторых редких случаях вызвать проблемы, заключается в размере переменной каждого типа:

```
int n = sizeof(bool); // 1 байт
int m = sizeof(BOOL); // 4 байта
```

3.5. Понятия объявления и определения

Программист дает имена переменным или функциям, а потом использует переменные в выражениях или вызывает функции. Когда компилятор встречается в тексте программы любое имя, он должен знать, что имеется в виду под этим именем (свойства переменной или функции), иначе компилятор не знает, как поступать с таким именем и выдает ошибку. Поэтому в языке C/C++ использованию любого имени должно предшествовать описание его свойств.

Объявление (declaration) — это инструкция компилятору, как использовать указанное имя (описывает свойства переменной или функции). Объявлений

одного и того же имени может быть сколько угодно, главное — чтобы они все были согласованы (т. е. одинаковы)!

Определение (definition) осуществляет привязку имени к сущности (к памяти для данных или к коду для функций), т. е. специфицирует код или данные, которые стоят за этим именем. В языке C/C++ существует правило единственного определения. Это означает, что определение может быть только одно! Если программист это правило нарушает, то получает ошибку компилятора о множественном определении.

ПРАВИЛО

В C/C++ нет умолчаний, поэтому любое имя должно быть объявлено, прежде чем оно может быть использовано!

ЗАМЕЧАНИЕ

В большинстве случаях объявления и определения разнесены, но иногда эти понятия совмещаются — например, для переменных зачастую определение совмещено с объявлением.

3.5.1. Объявление переменной

Переменная — это не что иное, как именованная область памяти, используемая для хранения информации (значения). Свойства переменной (рис. 3.4) определяются:

- ключевыми словами, использованными при объявлении (обязательным является указание типа переменной — **int**, **double**..., но могут быть употреблены и другие ключевые слова, которые предоставляют компилятору дополнительную информацию);
- контекстом определения (некоторые свойства переменных зависят от того места в тексте программы, где находится определение переменной).

Встречая имя переменной в выражении, компилятор должен знать свойства переменной с указанным именем, для того чтобы сгенерировать низкоуровневый код.

Объявление — это инструкция (statement), которая описывает свойства переменной с указанным именем. Следует отметить, что язык C++ (а изначально и Си) — это язык типизированный, т. е. разработанный таким образом, что существует очень тесная связь между типами, переменными и инструкциями для манипулирования этими переменными, а компилятор дотошно проверяет правильность использования программистом переменной в любом выражении.

Встретив объявление переменной, компилятор:

- запоминает соответствие имени переменной и ее типа, что будет в дальнейшем использоваться:
 - для *контроля типов* (можно ли использовать данную переменную в указанном выражении);
 - при *преобразовании операций* с данной переменной на языке высокого уровня в конкретные машинные команды (разные в зависимости от типа переменной);
- выделяет требуемый объем памяти для хранения значения переменной данного типа (но только в том случае, когда объявление является одновременно и определением) и дальше ассоциирует ее имя с адресом выделенного блока памяти.

Структура объявления переменной

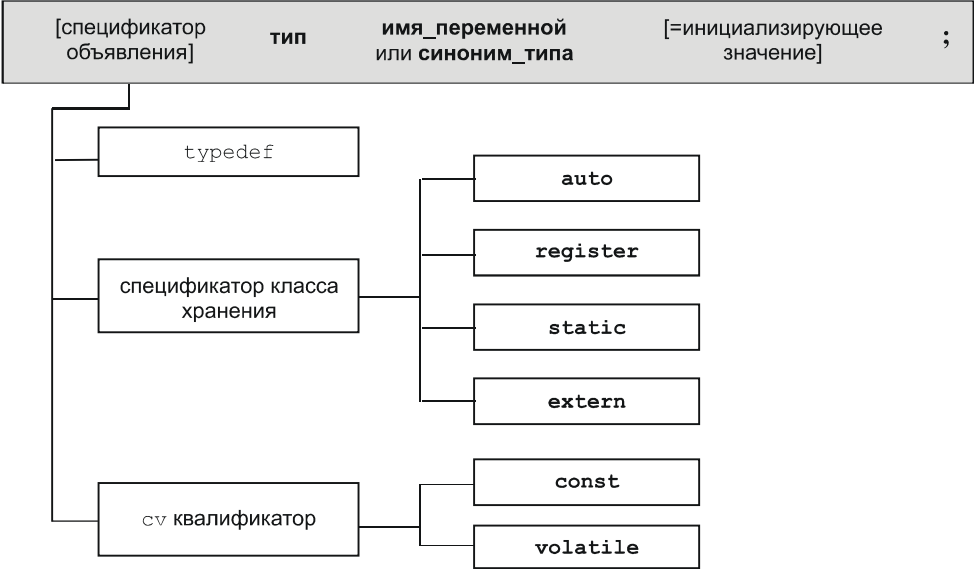


Рис. 3.4

Примеры простых объявлений:

```
extern int iNumber; //это только описание свойств (объявление)
                    внешней переменной iNumber, определенной
```

в другом модуле. Компилятор никакой памяти не резервирует!

```
char cSymbol; //а это объявление, совмещенное с определением.
```

Компилятор резервирует 1 байт

Приведенные объявления являются минимальными, в том смысле, что компилятору сообщается необходимый минимум информации, а он додумывает все остальное. На самом деле в примерах опущены еще две необязательные составляющие:

- ❑ если объявление совмещено с определением, то можно инициализировать переменную требуемым значением;
- ❑ можно ввести в объявление ключевые слова (спецификаторы объявления), которые не связаны с типом переменной, но, тем не менее, предоставляют компилятору некоторую дополнительную информацию о переменной (т. е. указания о том, как переменную следует использовать).

ЗАМЕЧАНИЕ 1

Если несколько переменных имеют один и тот же тип, то можно в объявлении использовать список имен через запятую:

```
int ix1, ix2=1, ix3;
```

РЕКОМЕНДАЦИЯ

Кроме простейших объявлений, не следует использовать список. Не экономьте — пишите объявление для каждой переменной, тогда в случае сложных объявлений избавите себя от поиска непонятных ошибок.

ЗАМЕЧАНИЕ 2

В языке Си обязательным требованием является указание всех объявлений переменных в начале того блока, в котором они используются (непосредственно после открывающей фигурной скобки). В C++ таких ограничений нет, более того, рекомендуется объявлять переменные по мере необходимости.

Ключевое слово *typedef*

Синтаксис:

```
typedef тип синоним_типа;
```

Объявления, начинающиеся с **typedef** — это указание компилятору на то, что задаваемый в объявлении синоним_типа не будет именем новой переменной (не следует отводить память), а будет использован программистом в качестве псевдонима указанного типа.

Ключевое слово **typedef** не создает новых типов данных, оно создает полезные мнемонические синонимы (псевдонимы) для существующих типов.

Используется **typedef** в двух случаях:

- для упрощения сложных объявлений (назначается короткий синоним для часто используемого типа):

```
typedef unsigned char BYTE; //вводит синоним типа, который
                             компилятор будет воспринимать точно так же,
                             как unsigned char
```

```
unsigned char cChar1; //создает переменную типа unsigned char
BYTE cChar2;          //создает переменную типа unsigned char
```

- для устранения зависимостей, например, от платформы или среды (пример приведен в *разд. 5.5*).

ЗАМЕЧАНИЕ 1

Нельзя комбинировать псевдонимы, определенные с помощью **typedef**, с другими типами:

```
typedef int MYINT;
unsigned MYINT my; //компилятор выдаст ошибку, т. к. он считает,
                  что за типом должно следовать имя переменной
```

ЗАМЕЧАНИЕ 2

Так как имена, вводимые **typedef**, являются синонимами, а не новыми типами, то старые типы можно продолжать использовать наравне с их синонимами.

3.6. Способы использования переменных и типы компоновки

Типичными являются следующие способы применения переменных программистом (рис. 3.5):

- первый — переменная используется внутри того блока кода, в котором она объявлена (локально), при этом могут быть два варианта:
 - (1а) — каждый раз при выполнении этого блока кода переменная заново инициализируется;
 - (1б) — переменная должна сохранять значение при следующем выполнении блока;
- второй (2) — переменная используется только внутри одного файла, но несколькими функциями. Поэтому, с одной стороны, это имя должно быть доступно в любом месте данного файла, а с другой стороны — заносить имена таких объектов в таблицу экспорта объектного модуля не нужно (см. *разд. 1.2.1*) — эта переменная в других модулях использоваться не будет;

- третий (3) — переменная используется разными единицами компиляции (функциями, расположенными в разных файлах), поэтому необходимо занести такое имя в таблицы экспорта/импорта соответствующих объектных модулей.

Способы использования переменных

1. cpp

2. cpp

```
extern int z; //(3) объявление внешней
переменной, определенной в другом
модуле

namespace {int y;} //(2) используется
только внутри данного файла
//или static int y; //(2)

void f1()
{
    int x=1; //(1a) используется только
    внутри блока. Инициализируется каждый
    раз при выполнении этого блока
    static int s; //(1б) используется
    только внутри блока. Инициализируется
    один раз и далее сохраняет значение
    при очередном выполнении блока
}

void f2()
{
    использование y
}

void f3()
{
    использование y
    использование z
}
```

```
int z; //(3) объявление,
совмещенное с
определением, переменной,
которая может быть
использована другими
модулями

void f4()
{
    использование z
}
```

Рис. 3.5

Со способами использования связаны понятия внутренней компоновки (internal linkage), внешней компоновки (external linkage) и без компоновки (no linkage):

- *без компоновки* (для первого способа — 1a, 1б). Не подлежат компоновке:
- все переменные, объявленные внутри любого блока без ключевого слова **extern** (такие имена являются уникальными в блоке);
 - параметры функций;

- *внутренняя* (для второго способа — (2)). Справедливо для переменных, объявленных вне фигурных скобок, но с ключевым словом **static** или заключенных в неименованное пространство имен (такие имена уникальны для данной единицы компиляции, поэтому конфликтов при использовании одного и того же имени в разных файлах не возникает);
- *внешняя* (для третьего способа — (3)). Справедливо для переменных, определенных вне фигурных скобок без спецификатора **static**. Понятие внешней компоновки относится к тем именам, которые описывают в разных единицах компиляции одну и ту же переменную, поэтому все объявления одной и той же переменной должны быть согласованы! Задачей программиста является обеспечение корректного объявления в каждом файле всех имен, определенных в других файлах. Для того чтобы сообщить компилятору, что переменная определена в другом модуле, ее нужно описать (объявить) в файле-клиенте с ключевым словом **extern**.

3.6.1. Безопасная сборка проекта (*type-safe linkage*)

В языке C++ поддерживается безопасная сборка исполняемого файла с учетом типов (*type-safe linkage*). Этот механизм не допускает ситуаций, когда, например, подлежащая внешней компоновке переменная в одном модуле объявлена как **int**, а в другом используется как **double**. В языке Си такого механизма защиты нет, поэтому там вполне возможна ситуация, представленная на рис. 3.6.

Пример возникновения ошибки в Си (без декорирования имен)

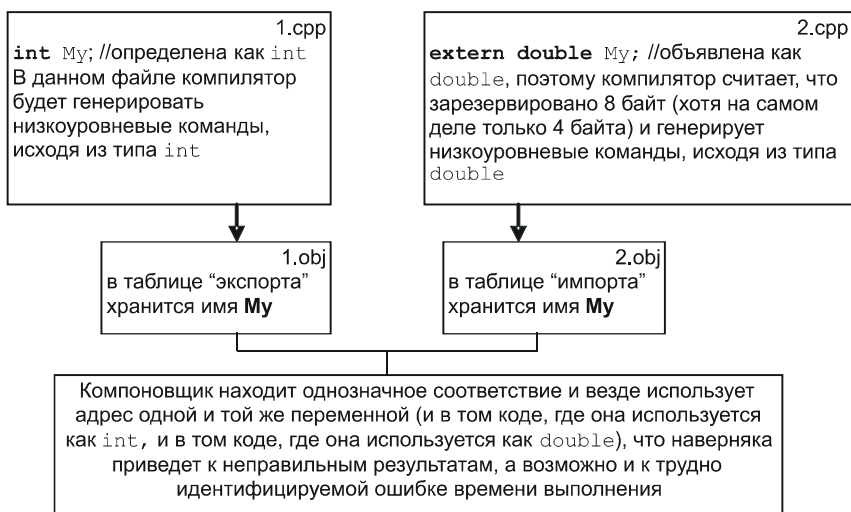


Рис. 3.6

Безопасная сборка проекта в C++

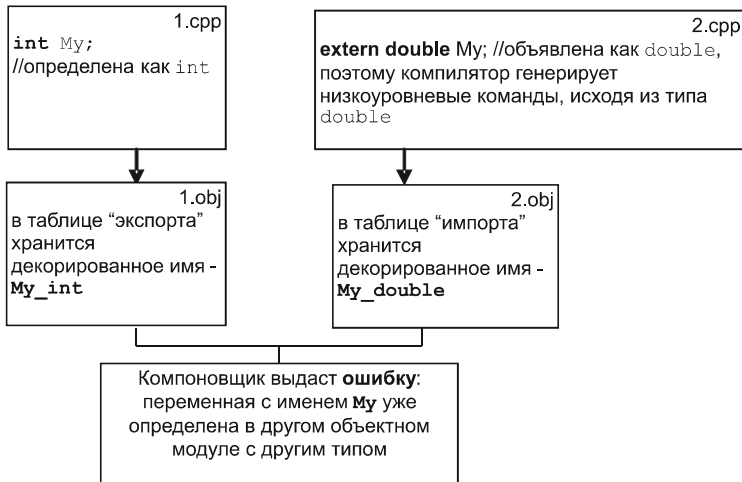


Рис. 3.7

Защита от таких ситуаций основана на понятии декорирования имен.

Декорирование (decorating) — это способность компилятора хранить в объектном модуле информацию об имени переменной вместе с ее типом. Например, если объявлена переменная `int My;` то компилятор помещает в объектный файл не просто имя `My`, а нечто, вроде `My_int` (рис. 3.7). Поэтому `My_int` и `My_double` для компоновщика будут различаться.

Существуют две основные причины для декорирования имен компилятором:

- ☐ обеспечение безопасной сборки исполняемого модуля;
- ☐ поддержка перегрузки имен функций (см. разд. 8.5), что невозможно в языке Си.

ЗАМЕЧАНИЕ

Следует отметить, что способы декорирования имен могут существенно различаться в разных компиляторах C++.

3.7. Размещение и время существования переменных

С видами компоновки связана область памяти, в которой компилятор выделит место для переменной (в C/C++ есть четыре основных способа размещения переменных — табл. 3.14). В свою очередь, от того, где компилятор

выделит память, зависит время существования переменной. По истечении этого интервала компилятор считает данный участок памяти свободным и может занять его под значения других переменных.

Каким образом компилятор решает, в какой же области памяти разместить переменную? В основном, исходя из контекста определения (того места в тексте программы, где определена переменная). Дополнительную информацию программист может предоставить компилятору посредством соответствующих ключевых слов: **static**, **auto**, **register**.

Таблица 3.14. Способы размещения и время существования переменных

Способы размещения	Время существования
<i>Статическая память</i>, в которую компилятор помещает объект на все время выполнения программы. Здесь располагаются глобальные переменные, переменные из пространств имен и статические переменные. Переменная, размещаемая в статической памяти, создается один раз (память выделяется один раз перед началом выполнения), ее значением можно пользоваться на всем протяжении выполнения программы. Такая переменная всегда имеет один и тот же адрес	Статическое Переменные существуют на всем протяжении выполнения программы
<i>Автоматическая память</i>, в которой по умолчанию располагаются переменные, определенные внутри любого блока (программист может явно определить их со спецификатором класса хранения — auto), а также параметры функций. Для таких переменных компилятор выделяет место в стеке. Память выделяется и освобождается автоматически, отсюда и название (ее также называют памятью в стеке). Следует помнить, что после освобождения памяти, выделенной под автоматическую переменную, компилятор может использовать освобожденную память для других целей. Для автоматических переменных при каждом последующем вызове функции может быть использован совершенно другой участок стека	Локальное Можно считать, что переменные создаются при входе в окружающий их определение блок и освобождают память при выходе из блока. Замечание На самом деле для всех локальных переменных функции компилятор выделяет память один раз при вызове функции, а освобождает после завершения функции (см. разд. 8.4)

Таблица 3.14 (окончание)

Способы размещения	Время существования
<i>Регистровая память.</i> Можно порекомендовать компилятору для повышения эффективности вычислений отвести для хранения значения переменной регистр (спецификатор класса хранения — <code>register</code>). Это будет служить подсказкой компилятору — по возможности размещать переменную в регистре, с чем, впрочем, современные компиляторы C++ и сами успешно справляются. Следует отметить, что такой способ оптимизации остается актуальным при программировании микроконтроллеров на языке Си	Локальное
<i>Динамическая память,</i> которую явно запрашивает программист для размещения переменных по мере необходимости и которую сам же освобождает, когда динамический объект больше не нужен (см. разд. 6.4). Такая память также называется <i>кучей</i> (heap). Выделение памяти из кучи осуществляется оператором C++ new (или функцией языка Си типа <code>malloc()</code>), а освобождение — оператором C++ delete (или функцией Си <code>free()</code>). При загрузке программы в ОП система выделяет также память и для кучи, если этой памяти в процессе выполнения не хватает, оператор new (или функция <code>malloc()</code>) запрашивает ее у диспетчера свободной памяти. В результате динамического выделения памяти программист получает для косвенного обращения к объекту адрес объекта	Динамическое Продолжительность существования определяется программистом. Замечание Аккуратный программист не должен забывать освобождать захваченную таким образом память, иначе образуются "утечки памяти" (см. разд. 6.4)

3.7.1. Ключевое слово *static*

Ключевое слово **static** — это указание компилятору выделить память для переменной в постоянной (статической) области памяти. Существуют два способа использования ключевого слова **static**.

Первый способ.

Спецификатор класса хранения **static** можно применить к переменной, определенной внутри любого блока. Смысл введения такой переменной: ком-

пилятор позволит обращаться к ней только внутри блока, в котором она определена, но существовать она будет до завершения программы (т. е. значение переменной будет сохраняться до следующего выполнения этого блока). Спецификатор **static** при объявлении локальной переменной предписывает компилятору:

- выделить память для такой переменной в постоянной (статической) области;
- проинициализировать такую переменную только один раз (на момент начала выполнения программы или при первом выполнении блока).

Например:

```
//выполнить_3_раза (i - номер итерации)
{
    static int n1 = 0; //такое инициализирующее значение будет
                       сформировано компоновщиком и будет храниться в
                       соответствующем разделе исполняемого файла,
                       поэтому никакого низкоуровневого кода этой строке
                       соответствовать не будет! Существует и сохраняет
                       свое значение до завершения программы
    static int n2 = выражение (например, i+4); //т. к.
                       выражение требуется вычислить при первом (только!)
                       выполнении этого блока, компилятор формирует
                       некоторый признак и проверяет этот признак при любом
                       выполнении данного блока. Инициализируется такая
                       переменная по правилам один раз, существует и
                       сохраняет свое значение до окончания программы
    int n3 = 0; //а это обычная автоматическая переменная,
                которая инициализируется при каждом выполнении блока
    n1++;
    n2++;
    n3++;
    //распечатать значения n1, n2, n3
}
//n1++; //ошибка: вне блока нельзя использовать любую
        из локальных переменных
```

В результате выполнения данного фрагмента переменные примут значения, представленные в табл. 3.15.

Таблица 3.15. Значения переменных для соответствующих итераций

Номер итерации (i)	n1	n2	n3
0	1	5	1
1	2	6	1
2	3	7	1

Второй способ.

Спецификатор класса хранения **static** можно применить к переменной, объявленной вне любых фигурных скобок. Если программист объявляет переменную (или функцию) с ключевым словом **static**, он сообщает компилятору, что собирается пользоваться этой переменной в нескольких функциях, но только внутри данного файла (такие переменные не подвержены внешней компоновке, сведения об их экспорте компилятором не формируются). Ключевое слово **static** предотвращает ошибки компоновщика, т. к. каждая переменная используется только внутри соответствующего файла (рис. 3.8).

Спецификатор хранения `static` для внутренней компоновки

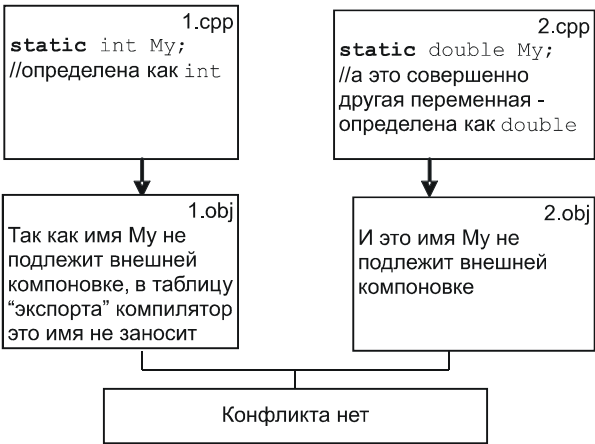


Рис. 3.8

ЗАМЕЧАНИЕ

Использование **static** в языке C++ для внутренней компоновки считается устаревшим. Новые стандарты языка предлагают употреблять для тех же целей неименованные пространства имен (см. разд. 3.8.2).

3.8. Область видимости переменной (scope)

Область видимости переменной контролируется компилятором. Это означает, что имена переменных могут быть использованы только в конкретных областях текста программы. Такой фрагмент текста программы называется областью видимости имени. Компилятор в каждый момент времени должен ассоциировать указанное имя с конкретной областью памяти (если такая ассоциация у него отсутствует, то он выдает ошибку о том, что имя не определено). Существуют правила, по которым компилятор формирует области видимости переменных.

С точки зрения программиста эти правила выглядят следующим образом:

- локальная область видимости. Переменные, определенные внутри любого блока (в частности таким блоком может быть тело функции), называются локальными. Область видимости локальной переменной простирается от места определения до конца блока, содержащего определение:

```
{  
    ...  
    int i;  
    i++;  
}  
i=1; //ошибка: неопределенное имя
```

ЗАМЕЧАНИЕ

Параметры функции тоже имеют локальную область видимости;

- область видимости — файл (File Scope). Любое имя, объявленное вне фигурных скобок, обладает областью видимости файла. Компилятор позволит использовать такое имя (оно видно компилятору) в любом месте данного файла после объявления и до конца файла без каких-либо дополнительных ухищрений.

ЗАМЕЧАНИЕ

Имена, объявленные таким образом без ключевого слова **static**, часто называют глобальными. К таким именам можно обращаться из других файлов. Они подвержены внешней компоновке;

- область видимости — пространства имен (namespaces) — относительно новое средство языка C++ ограничения области видимости глобальных имен (см. разд. 3.8.2);

- область видимости — тело функции (Function scope). Этой областью видимости обладают только метки (см. разд. 4.4). Их можно использовать в любом месте функции, но извне функции они не видны:

```
void f()
{
    goto LABEL;
    ...
    LABEL: //метка
    ...
    goto LABEL;
}
```

- область видимости имен параметров прототипа функции (см. разд. 8.2) распространяется только на объявление функции;
- область видимости членов структуры объединения, класса разрешается посредством *объектов* или *указателей на объекты* (см. разд. 9.3, 9.11).

РЕКОМЕНДАЦИЯ (БЬЕРН СТРАУСТРУП)

В C++ не объявляйте локальную переменную, пока она вам действительно не потребуется (это делает вашу программу более читабельной)! При объявлении по возможности инициализируйте локальные переменные! Неинициализированные переменные провоцируют ошибки!

3.8.1. Скрытие (замещение) имени переменной

Иногда случайно (реже специально) программист называет одним и тем же именем две разные переменные. Объявление имени в блоке скрывает переменную с тем же именем в охватывающем блоке или глобальную переменную. После выхода из блока имя восстанавливает свой прежний смысл. Правила перекрытия областей видимости продемонстрированы в листинге 3.9.

Листинг 3.9. Замещение имени переменной

```
int X; //глобальная переменная

int main()
{
    X = 1; //компилятор ассоциирует имя X с глобальной переменной

    int X; //объявление локальной переменной с таким же именем
           как и у глобальной. С этого момента и до конца функции
           имя локальной переменной скрывает имя глобальной
```

```

X = 2; //локальной переменной присваивается 2
{
    int X; //во вложенном блоке объявляется еще одна локальная
           переменная с тем же именем. С этого момента и до
           конца блока имя внутренней переменной скрывает не
           только имя глобальной, но и имя внешней локальной
           переменной
    X = 3; //внутренней локальной переменной присваивается 3
} //здесь заканчивается область видимости внутренней локальной
  переменной, и имя X восстанавливает тот смысл, который был
  у него перед началом блока
X = 4; //внешней локальной переменной присваивается 4
}

```

Компилятор никогда не позволит определить в одной области видимости несколько переменных с одним и тем же именем (листинг 3.10).

Листинг 3.10. Попытка определения двух переменных с одним и тем же именем в одной области видимости

```

{
    int value;
    ...
    float value; //ошибка: переменная с таким именем в данной
                 области видимости уже существует!
    {
        float value=1.1; //а здесь ошибки не будет, т. к.
                           внутренняя локальная переменная
                           перекрывает область видимости
                           внешней локальной переменной
        value++; //инкремент внутренней переменной
    }
}

```

Оператор разрешения области видимости

В C++ (в языке Си такой возможности не было) к скрытому глобальному имени можно обратиться с помощью оператора разрешения области видимости — "::". Пример приведен в листинге 3.11.

Листинг 3.11. Оператор разрешения области видимости

```
int X;
int main()
{
    X = 1; //компилятор ассоциирует имя X с глобальной переменной
    int X; //внешняя локальная переменная "перекрывает" область
           //видимости глобальной
    ::X = 100; //указание компилятору присвоить значение 100
              //глобальной переменной
    X = 2; //а здесь компилятор под именем X "видит" внешнюю
           //локальную переменную
    {
        int X; //внутренняя локальная переменная перекрывает область
              //видимости как глобальной, так и внешней локальной
        ::X = 200; //поскольку :: разрешает только глобальную
                  //область видимости, то модифицирована будет
                  //глобальная переменная
        X = 3; //под именем X видит внутреннюю локальную
              //переменную
    }
    X = 4; //внешняя локальная переменная восстановила свой контекст
}
```

ЗАМЕЧАНИЕ

Не существует способа обращения к скрытой локальной переменной!

3.8.2. Пространства имен — *namespace*

Если переменная определена вне фигурных скобок без ключевого слова **static**, то компилятор по умолчанию считает, что такая переменная является глобальной (т. е. ее можно использовать в других модулях). При этом возникают следующие проблемы:

- имена всех глобальных переменных и функций (их много!) находятся в единой (глобальной) области видимости:
- имена ваших собственных глобальных переменных и функций;
- имена, предоставляемые стандартной библиотекой (такие простые имена как `copy`, `sort`, `find`, `set`);
- глобальные имена вашего коллеги, работающего над тем же проектом;

- глобальные имена, логически относящиеся к одному понятию, оказываются никаким образом не связанными друг с другом.

Для того чтобы упорядочить этот "глобальный хаос", состоящий в том, что глобальных имен оказывается слишком много и они относятся к объектам, логически не связанным друг с другом, было введено понятие пространств имен (стандарт языка C++ 1998). На рис. 3.9 приведен простой пример возникновения конфликта.

Конфликт глобальных имен

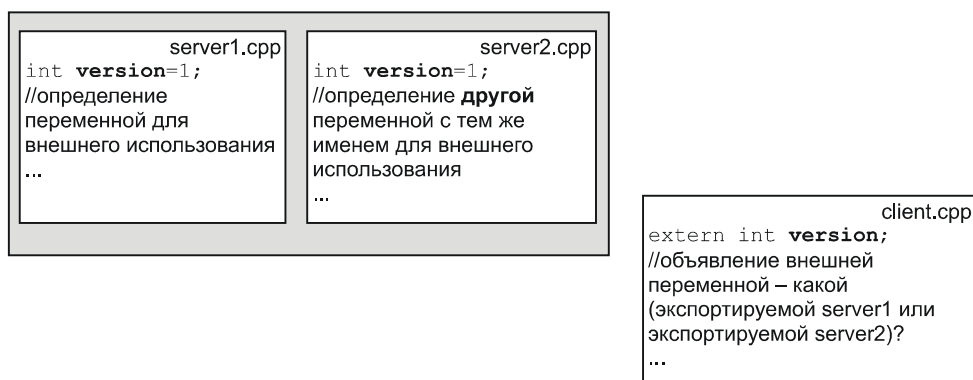


Рис. 3.9

В такой ситуации при сборке проекта компоновщик выдаст ошибку: "множественное определение имени", хотя под этим именем по замыслу программиста имеются в виду две совершенно разных переменных.

Чтобы избежать таких ситуаций, в современных версиях языка C++ введено понятие пространства имен — **namespace**. Пространство имен является средством ограничения области видимости. Для того чтобы поместить несколько имен в одно пространство имен, надо написать:

```
namespace имя_пространства_имен{ объявления_переменных_или_функций; }
```

Каждое имя становится уникальным в своем пространстве имен. Объявив переменную или функцию в пространстве имен, к ним можно (без специальных ухищрений) обратиться только посредством явного указания имени пространства в качестве квалификатора и оператора разрешения области видимости — "::", т. е. все объекты, принадлежащие одному пространству имен, имеют имена, состоящие из двух частей:

```
имя_пространства::имя_переменной_или_функции
```

СЛЕДСТВИЕ

При заключении имени в пространство имен, глобальные переменные (и функции) могут иметь одинаковые имена даже в пределах одного файла, но, естественно, такие преднамеренные дублирования имен вряд ли полезны при программировании. Скорее **namespace** — это средство от случайного дублирования имен.

Конфликтную ситуацию, приведенную на рис. 3.9, можно было бы разрешить способом, который иллюстрируется листингом 3.12.

Листинг 3.12. Использование пространств имен для ограничения области видимости глобальных переменных

```
namespace One{ int version = 1;}
namespace Second{ int version = 2;}
int main()
{
    int n1 = One::version; //n1=1
    int n2 = Second::version; //n2=2
}
```

Особенности использования пространств имен:

- пространство имен позволяет не только ограничить доступ к именам извне, но и сгруппировать имена по некоторому критерию (например, имена переменных и функций, относящихся к одной области применения, или все имена, входящие в разрабатываемые отдельным программистом модули большого программного проекта). Например:

```
namespace Version
{
    int currentVersion;
    int previousVersion;
    void SetVersion(int ver); //причем внутри такой функции можно
                               обращаться к переменным currentVersion и
                               previousVersion, а также вызывать другие функции,
                               заключенные в данное пространство имен без
                               префикса Version
    int GetVersion();
    ...
}
```

- в сущности понятия, заключенные в пространство имен, являются глобальными, а сами пространства просто ограничивают их область видимости.

Поэтому для переменных из пространств имен справедливы те же правила, что и для обычных глобальных переменных — такие переменные должны быть определены только один раз. А для того чтобы можно было пользоваться такими переменными в других модулях, нужно описать компилятору их свойства посредством ключевого слова **extern**. При этом необходимо уточнить, что используемая внешняя переменная заключена в пространство имен (рис. 3.10).

Использование внешних переменных из пространств имен

1.cpp	2.cpp
<pre>namespace One{int version=1;} //определение void f1() { использование One::version }</pre>	<pre>namespace One(extern int version;) //объявление (описание свойств). //обязательным является указание //пространства имен void f2() { использование One::version }</pre>

Рис. 3.10

ЗАМЕЧАНИЕ

Объявление внешних понятий корректнее помещать в специальные, предназначенные для описаний заголовочные файлы (см. разд. 5.6);

- пространства имен обладают свойством открытости. Это означает, что программист может дополнять пространства имен членами по мере необходимости (рис. 3.11). Поэтому возможно разносить понятия, относящиеся к одному пространству имен, по разным файлам (на свойстве открытости основано постепенное включение имен стандартной библиотеки);

Открытость пространств имен

1.cpp	2.cpp	3.cpp
<pre>namespace One{ int version=1;} void f1() { использование version }</pre>	<pre>namespace One{ int prevVersion=2;} void f2() { использование prevVersion }</pre>	<pre>namespace One{ extern int version; extern int prevVersion; int MyVersion=3; } void f3() { использование version использование prevVersion использование MyVersion }</pre>

Рис. 3.11

- использование псевдонимов пространств имен. Короткие имена пространств рано или поздно приведут к конфликту (рис. 3.12). Длинные имена непрактичны и рано или поздно программисту надоест писать такой длинный префикс, поэтому можно сопоставить длинному имени короткий псевдоним и пользоваться им до завершения области видимости псевдонима.

```
{
    namespace ver = Version_beta_12345; //псевдоним
    ver::currentVersion++; //до конца блока можно пользоваться
                           псевдонимом ver для обращения к членам пространства
                           имен Version_beta_12345
}
```

Конфликт коротких названий пространств имен

<pre>namespace A{...} //это пространство имен одного программиста</pre> 1.cpp	<pre>namespace A{...} //а это пространство имен его коллеги, который случайно назвал свое пространство тем же коротким именем</pre> 2.cpp
---	---

В результате все имена оказались в одной области видимости!

Рис. 3.12

Директива *using*

Ограничение доступа было введено для повышения надежности программ. Обеспечение безопасности имеет свою оборотную сторону: для каждого обращения к объекту из пространства имен программисту приходится писать префикс (имя_пространства::). Со временем это начинает раздражать. Если имя, требующее квалификатора, используется часто, довольно утомительно каждый раз писать этот квалификатор. Такое неудобство можно устранить с помощью директивы **using**, которая делает доступными имена из данного пространства имен, как если бы они были объявлены глобально.

Существуют две формы директивы:

- **directive** — делает все члены пространства доступными, т. е. с ними можно работать как с обычными глобальными понятиями (без указания префикса):

```
using namespace имя_пространства;
```

- **declaration** — делает видимым только указанный член пространства имен:

```
using имя_пространства::имя_члена_пространства;
```

Например:

```
{
    //currentVersion++; //ошибка: неопределенный
                          идентификатор currentVersion
    using namespace One;
    //или using One::currentVersion;
    currentVersion++; //корректно
}
//currentVersion++; //ошибка: неопределенный
                          идентификатор currentVersion
```

РЕКОМЕНДАЦИЯ Б. СТРАУСТРУПА

Если вы хотите пользоваться именами из определенного пространства имен во всех функциях файла, конечно, можно вынести директиву **using** до первой открывающей фигурной скобки, но по возможности разрешение таких имен стоит ограничивать блоком использования.

Специфика использования директивы **using**:

- одно пространство имен можно сделать видимым из другого с помощью директивы **using** способом, представленным в листинге 3.13.

Листинг 3.13. Использование директивы using в ситуации, когда одно пространство имен оперирует понятиями из другого пространства

```
namespace A{ int a = 2;}
namespace B{
    int b=1;
    using namespace A; //делает доступными все имена из A, т. е.:
        а) в функциях, заключенных в пространство имен B,
           можно обращаться ко всем объектам пространства A
           без префикса
        б) ко всем именам A можно обращаться посредством B::
    //или using A::a; //если требуется иметь доступ только к какому-то
                   одному имени из A
    void F(){a=5;}    //(a) можно обращаться без префикса
}
```

```
int main()
{
    //разрешили посредством внешнего пространства видеть внутреннее:
    B::b=1;
    B::a = 2;          //(б)можно обращаться посредством B::
    //Как работает директива using:
    using namespace B; //разрешение видеть (как глобальные)
                        все имена как из B, так и из A

    //или

    using B::b;
    using B::a;

    //в обоих случаях теперь можно обращаться к именам пространств A и B
    без префикса
    b=22;
    a=33;
}
```

- директива **using** разрешает компилятору видеть имена из пространства как глобальные. В такой ситуации могут возникнуть конфликты (листинг 3.14).

Листинг 3.14. Пример конфликта при использовании директивы using

```
namespace A{int a=1;} //переменная заключена в пространство имен
int a=2; //обычная глобальная переменная
int main()
{
    //Пока не разрешена область видимости имен A, компилятор
    однозначно видит под именем a глобальную переменную
    a++; //обращение к глобальной переменной
    ::a++; //или то же самое посредством спецификатора
           разрешения области видимости
    A::a++; //обращение к переменной из пространства имен
    using namespace A;
    //a = 5; //ошибка: двусмысленный символ (ambiguous symbol).
           Возникла неоднозначность.
    ::a = 5; //OK
    A::a = 5; //OK
}
```

- использование вложенных пространств имен:
 - без директивы **using** (листинг 3.15);

Листинг 3.15. Использование вложенных пространств без директивы using

```
namespace Out{ //внешнее пространство
    int outer;
    namespace In{ //внутреннее пространство
        int inner;
    }
    void f(){outer++; In::inner++;} //без using компилятору нужно
                                   явно указать префикс In::inner
}
int main()
{
    Out::outer=1;
    Out::In::inner=2;
    ...
}
```

- с использованием директивы **using** (листинг 3.16) (цель — обеспечить видимость членов внутреннего пространства посредством внешнего пространства);

Листинг 3.16. Вложенные пространства и директива using

```
namespace Out{
    int outer;
    namespace In{
        int inner;
    }
    using namespace In;
    void f(){outer++; inner++;} //префикс не нужен!
}
int main()
{
    Out::outer=1;
    Out::inner=2; //теперь можно обращаться к членам внутреннего
```

```

        пространства имен посредством имени внешнего
        пространства

{
    using namespace Out; //а если разрешить видимость
                          внешнего пространства

    outer=1;
    inner=2; //то члены внутреннего пространства тоже
              становятся видимыми глобально
}

```

- если видимость переменной из пространства имен разрешена директивой **using**, то по правилам замещения имен объявление локальной переменной с тем же именем скрывает переменную из пространства имен (листинг 3.17);

Листинг 3.17. Правила замещения имен и директива **using**

```

namespace A{int a=1;}

int main()
{
    using namespace A; //разрешили глобальную видимость
    a++; //можем обращаться без префикса
    int a=3; //локальная переменная с тем же именем перекрывает
             видимость переменной из пространства имен
    a++;    //увеличивается значение локальной переменной
    A::a++; //а переменную из пространства имен теперь можно
            увидеть только посредством префикса
    ::a++;  //или так
}

```

- специфика использования declaration-формы директивы **using** (листинг 3.18).

Листинг 3.18. Пример возникновения конфликта при использовании declaration-формы директивы **using**

```

namespace A{int a=1;}

int main()
{
    using A::a; //разрешает глобальную видимость только a
    a++; //теперь можно обращаться к переменной без префикса
}

```

```

//int a=3; //а теперь при попытке объявления локальной переменной
           с тем же именем компилятор выдаст сообщение
           об ошибке: повторное определение (redefinition)
}

```

Пространство имен стандартной библиотеки

Так как стандартная библиотека изобилует простыми именами, то для предотвращения дублирования таких имен согласно стандарту все сервисы стандартной библиотеки заключены в пространстве имен с именем `std`.

Примеры использования средств ввода/вывода стандартной библиотеки:

□ без директивы **using**:

```

{
    std::cout<<"Input x:"<<std::endl;
    int x;
    std::cin>>x;
}

```

□ с помощью директивы **using**:

- все имена из пространства имен `std` делаем видимыми глобально:


```

{
    using namespace std; //все имена из пространства имен std делаем
                        видимыми глобально
    cout<<"Hello, World!" <<endl; //теперь префикс не нужен
}

```
- делаем видимым глобально только отдельные члены:


```

{
    using std::cout; //только cout делаем видимым глобально
    using std::endl; //endl - аналогично
    cout<<"Hello, World!" <<endl; //префикс не нужен
}

```

Неименованные пространства имен

Неименованные пространства имен применяют для того, чтобы ограничить использование переменной или функции только данным модулем. Таким образом, внутри файла, содержащего безымянное пространство, к членам этого пространства можно обращаться напрямую без префикса, но вне этого файла такие идентификаторы использовать невозможно (рис. 3.13).

Ограничение использования переменной файлом посредством
неименованного пространства имен

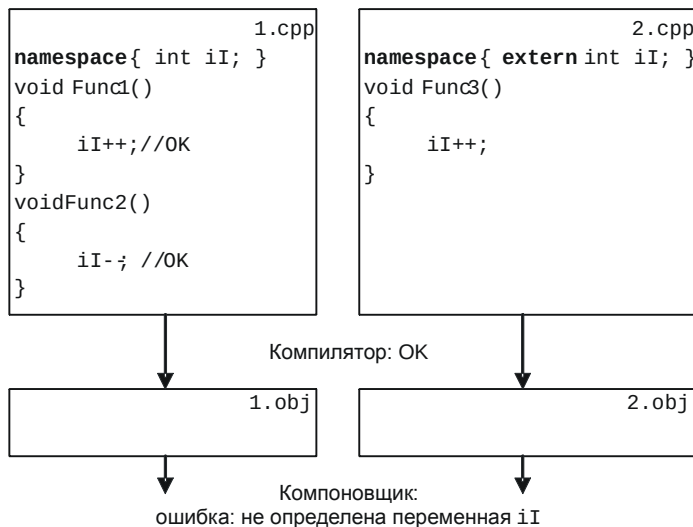


Рис. 3.13

Для переменных, заключенных в неименованные пространства имен справедливы правила замещения глобальных имен локальными (листинг 3.19).

Листинг 3.19. Неименованные пространства имен и правила разрешения области видимости

```

namespace {int a=100;}
int main()
{
    a++; //для того, чтобы обратиться к переменной, заключенной
        в неименованное пространство имен, префикс не нужен
    int a=1; //локальная переменная перекрывает область видимости
            переменной из неименованного пространства
    a++;    //обращение к локальной переменной
    ::a++; //а так можно обратиться к переменной из пространства имен
}

```

РЕКОМЕНДАЦИЯ Б. СТРАУСТРУПА

Для ограничения области видимости переменных (или функций) пределами одного файла предпочтительнее заключать их в неименованное пространство имен, нежели использовать ключевое слово **static**.

3.9. Инициализация переменных

3.9.1. Явная инициализация переменных (программистом)

Большие неприятности могут ожидать программиста, если он небрежно обращается с переменными, в частности — если забывает присваивать им значения перед использованием. Хороший компилятор выдает предупреждение, когда встречает непроинициализированную переменную, а некоторые компиляторы в Debug-версии приложения вставляют проверки на использование таких значений. Инициализация означает присваивание значения при определении, например:

```
int iNumber = 5;  
char cSymbol = 'A';
```

РЕКОМЕНДАЦИЯ

Не объявляйте локальную переменную, пока она вам не потребуется, чтобы можно было сразу же ее проинициализировать (это уменьшит вероятность операций над случайным значением).

3.9.2. Неявная инициализация переменных (компилятором)

Если при определении программист явно не присвоил переменной какого-либо значения, компилятор может проинициализировать ее сам. Правила неявной инициализации:

- ❑ переменные со статическим временем существования (глобальные, в пространствах имен и статические) инициализируются нулем (в прологе приложения, который предоставляется стандартной библиотекой);
- ❑ автоматические переменные не инициализируются (т. е. имеют случайное значение той области стека, которую компилятор выделил под автоматическую переменную);

ЗАМЕЧАНИЕ

В большинстве реализаций в Debug-версии проекта используемая под автоматические переменные область стека заполняется значениями 0хсссссссс, что позволяет при отладке выявлять использование неинициализированных переменных.

- ❑ динамические переменные не инициализируются (т. е. имеют случайное значение той области памяти, которая была выделена под объект в куче).

ЗАМЕЧАНИЕ

В большинстве реализаций в Debug-версии проекта перед выполнением программы область heap заполняется значениями 0xcdcdcdcd, что также позволяет выявить значение неинициализированных переменных в куче.

3.10. Модификаторы *const* и *volatile*

Посредством ключевых слов **const** и **volatile** (или cv-модификаторов) программист предоставляет компилятору дополнительную информацию о том, какие ограничения следует соблюдать при использовании таких объектов.

3.10.1. Ключевое слово *const*

Ключевое слово **const** в C/C++ в разных контекстах имеет разный смысл! Применительно к переменным объявление с ключевым словом **const** означает, что переменную можно использовать только для чтения (она должна быть обязательно проинициализирована при определении). Любая попытка впоследствии изменить ее значение вызовет ошибку компиляции. Например:

```
const int nMax=1000; // корректно
int nX = nMax; // корректно
const int nMin; //ошибка: нет инициализатора
nMax = 2000; //ошибка: нельзя модифицировать константу
```



Подумайте: является ли такое объявление корректным?

```
extern const int nMin;
```

Специфика определения переменных с ключевым словом **const**:

- ❑ инициализирующее значение компилятор знает на этапе компиляции:

```
const int N = 1; //инициализирующее значение константа
```

Оптимизирующий компилятор C++ (в зависимости от дальнейшего использования такой переменной) память для нее отводить не будет, а встречая это имя в выражении, будет просто подставлять ее значение. В языке Си таких оптимизаций компилятор не делает;

- ❑ в качестве инициализирующего значения используется выражение:

```
const int M = выражение;
```

Если выражение невозможно вычислить на этапе компиляции, компилятор отведет для такой переменной память, вычислит значение, присвоит этой переменной результат, но в дальнейшем разрешит использовать ее только для чтения.

3.10.2. Ключевое слово *volatile*

Модификатор **volatile** означает, что переменная может изменяться не только текущей программой, но, возможно, и где-то вне программы (например, обработчиком прерывания). Описание переменной как **volatile** информирует компилятор о том, что в процессе вычисления переменная может подвергнуться изменениям извне, поэтому всякий раз, встречая в выражении это имя, компилятор должен заново считывать ее значение из памяти.

СЛЕДСТВИЕ

Запрещается для такой переменной распределять регистр и проводить некоторые виды агрессивной оптимизации.

Например, пусть наша программа состоит из двух функций, которые могут выполняться параллельно (одновременно).

Использование ключевого слова *volatile*

main.cpp

```
extern volatile int nTicks;
//объявление счетчика тиков таймера

const int INTERVAL = 1000;

int main()
{
    ...
    nTicks = 0;
    while(nTicks < INTERVAL);
    //ожидание истечения интервала.
    Существенным является
    считывание значения из памяти
    на каждой итерации
    //Продолжение выполнения
    функции main()
    ...
}
```

handler.cpp

```
volatile int nTicks;
//определение счетчика тиков таймера

обработчик_прерываний_от_таймера()
{
    nTicks++; //увеличение значения
              счетчика тиков на единицу
}
```

Первая функция `main()`, дойдя до определенного места, должна приостановить свое выполнение на интервал времени, заданный константой `INTERVAL` (для этого она проверяет значение общей переменной `nTicks`).

Вторая функция является специально оформленным обработчиком прерывания от системного таймера, ее задачей является подсчет тиков таймера (обработчик, получая управление, просто инкрементирует общую глобальную переменную `nTicks`).

При таком взаимодействии функций важно, чтобы компилятор для проверки значения переменной `nTicks` на каждой итерации цикла считывал значение из памяти (а не держал его на регистре для оптимизации вычислений), т. к. значение переменной мог изменить обработчик прерываний (рис. 3.14).

ЗАМЕЧАНИЕ

Объект может быть одновременно **const** и **volatile**.

Например:

```
extern const volatile clock; //в этом случае наша программа не  
                               может изменять значение, а извне эта переменная  
                               может быть изменена!
```

Глава 4



Инструкции (*statements*) C/C++

4.1. Общая информация об инструкциях

Программа — это последовательность инструкций. Инструкция C/C++ (statement) — это элемент программы, содержащий имена, операторы и разделители и заканчивающийся точкой с запятой (рис. 4.1).

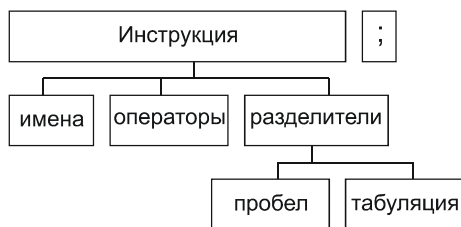


Рис. 4.1

Инструкции выполняются в том порядке, в котором они встречаются в программе, кроме тех случаев, когда порядок нарушается по замыслу программиста опять же с помощью инструкций (инструкций выбора, цикла или безусловного перехода).

Признаком конца инструкций является точка с запятой (;) (ограничитель объявления!).

Разделителями отдельных частей инструкции являются пробелы и табуляции. В большинстве случаев количество пробелов или табуляций не имеет значения — препроцессор лишние проигнорирует.

Перечень инструкций приведен в табл. 4.1.

Таблица 4.1. Инструкции C/C++

Инструкция	Пояснение	Назначение
<i>Составная</i> (compound statement)	Группа инструкций, заключенная в фигурные скобки { } или блок кода	Для выделения логически связанных между собой инструкций. Замечание Группа { } может в частности не содержать вообще никаких инструкций
<i>Объявление</i> (declaration statement)	Описание свойств переменной, функции или пользовательского типа	Объясняет компилятору, каким образом он должен обращаться с вводимым посредством объявления именем
<i>Выражение</i> (expression statement)	Содержит последовательность операндов и операторов для действий над этими операндами	Может содержать как арифметические и логические выражения, так и вызовы функций. Замечание: все действия, предусмотренные в выражении, гарантированно выполняются, прежде чем начинается выполнение следующей инструкции
<i>Пустая инструкция</i> (null statement)	Инструкция, состоящая только из ограничителя — точки с запятой (;)	Применяется в том случае, когда синтаксис C/C++ требует наличия инструкции, но на самом деле никаких действий не предполагается
<i>Выбора</i> (selection statement)	if (условие) инструкция [else инструкция]; switch (условие) инструкция;	Средство для выполнения блока кода в зависимости от выполнения условия
<i>Цикла</i> (iteration statement)	while (условие) инструкция; do инструкция while (условие); for (инициализирующая_инструкция; условие; выражение) инструкция;	Средство для многократного выполнения инструкции. Замечание: в частности, количество итераций может быть равно 0
<i>Безусловного перехода</i> (jump statement)	break ; continue ; goto идентификатор;	Способ прекращения последовательного выполнения инструкций и передачи управления в предопределенное место программы

Таблица 4.1 (окончание)

Инструкция	Пояснение	Назначение
<i>Возврата</i> (return statement)	return выражение;	Формирование возвращаемого функцией значения и возврат управления
<i>Помеченная</i> (labeled statement)	идентификатор: инструкция; case константа : инструкция; default : инструкция;	Для того чтобы передать управление на конкретную инструкцию, она должна быть помечена
<i>Обработки исключений</i>	try {инструкция throw catch	Механизм выявления и обработки исключительных ситуаций в C++

4.2. Инструкции выбора (условия)

Любой язык программирования должен иметь возможность условного ветвления. В некоторых языках для этого есть конструкция **if-then-else**.

Язык C/C++ для формирования условий предоставляет инструкции: **if**, **if...else** и **switch**.

4.2.1. Инструкции **if**, **if...else**

Синтаксис:

```
if(условие){true_инструкция}           //если условие = true
    [else {false_инструкция}] // если условие = false
```

Если условие принимает значение true, выполняется true-инструкция (возможно единственная), в противном случае — false-инструкция, если она указана (рис. 4.2).

Специфика использования:

- ❑ в языке C/C++ нет ключевого слова **then**. Выражение, следующее за условием, является аналогом then-части. Более того, простейшая форма инструкции не требует даже и else-части. Например, пусть требуется вычислить абсолютное значение переменной x:

```
if(x<0) x=-x;
```

- ❑ в качестве условия может быть не только "честное" условное выражение, в котором участвуют логические операторы и операторы отношения (==, !=, <, <=, >, >=, &&, ||, !), а результатом является true или false, но и любое арифметическое выражение, например:

```
if(x!=0){...} эквивалентно if(x){...};
```

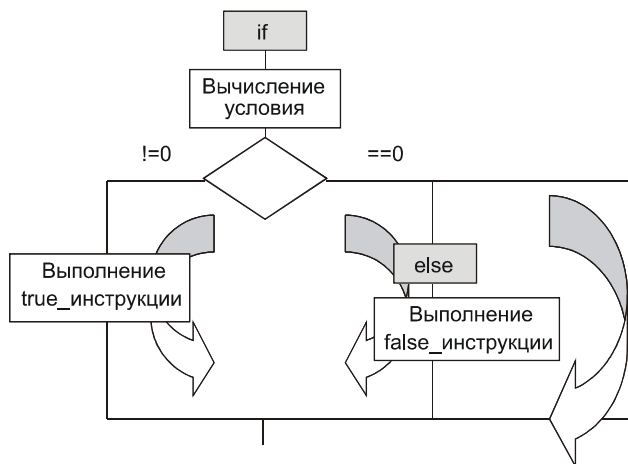


Рис. 4.2

- true-инструкция и false-инструкция, в свою очередь, сами могут быть инструкциями **if**, образуя вложенные условные выражения.

Пример.

Пусть в зависимости от значения x переменная y принимает значения, представленные на рис. 4.3.

Один из возможных способов решения:

```

if(x>0) y=1;
else if(x==0) y=0;
else y=-1;
  
```

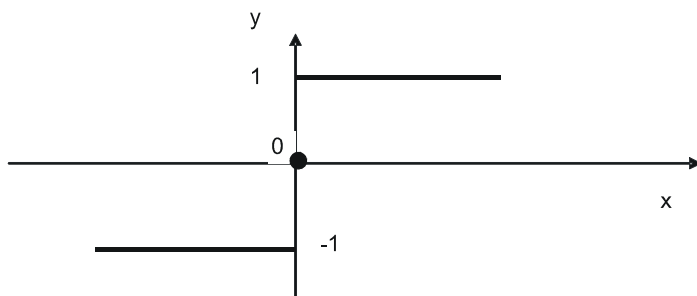


Рис. 4.3

ЗАМЕЧАНИЕ

В таких вложенных конструкциях компилятор сопоставляет каждый **else** с последним встретившимся (ближайшим) **if**, не имеющим своего **else**. Так как инструкция **else** не обязательна, то если хотите, чтобы компилятор правильно понял ваш замысел — ставьте фигурные скобки! В табл. 4.2 продемонстрирован фрагмент кода с фигурными скобками и без них, а также результат его выполнения.

Таблица 4.2. Результаты выполнения инструкции if...else

Некорректно	Корректно
y=1; if(x>=0) if(x==0) y=0; else y=-1;	y=1; if(x>=0) { if(x==0) y=0;} else y=-1;
при x>0 y=-1, (компилятор соотнесет else ближайшему if)	при x>0 y=1

- возможно объявление переменной в условиях. Область видимости такой переменной распространяется на обе ветви **if...else**.

Пример.

```
int NN = 1;  
if(int tmp = func()) //функция может вернуть любое значение,  
                     // в частности 0. Возвращенное значение  
                     // присваивается временной переменной tmp  
                     // и используется компилятором для  
                     // формирования условия  
{ //true-инструкция (tmp!=0)  
  NN /= tmp; //деление безопасно  
}else  
{  
  NN = <что-нибудь другое>; //на ноль делить нельзя!!!  
}
```

- часто возникает необходимость сравнения результата выражения с набором целочисленных констант.

Пример.

```
if(x==1)  
...//выполнить действие 1  
else if(x==2)
```

```
...// выполнить действие 2
else if(x==3)
    ...// выполнить действие 3
else ...//если значение не совпало ни с одним
        из перечисленных
```

ЗАМЕЧАНИЕ

Такую же конструкцию можно получить с помощью инструкции **switch** (гораздо удобнее и структурнее!).

4.2.2. Переключатель — *switch*

На практике часто возникает задача выбора одного из вариантов действий в зависимости от результата вычисления выражения. Это можно сделать с помощью вложенных инструкций **if...else** (см. разд. 4.2.1). Однако более наглядно (структурировано) такие задачи решаются посредством инструкции **switch**. Приведенный в предыдущем разделе пример можно переписать:

```
switch(выражение)
{
    case 1:
        ...//выполнить действие 1
        break;
    case 2:
        ...//выполнить действие 2
        break;
    case 3:
        ...//выполнить действие 3
        break;
    default:
        ...//действие по умолчанию
}
```

Программист посредством ключевого слова **case** предоставляет компилятору набор констант (это точки передачи управления). Компилятор вычисляет выражение и, в случае совпадения результата выражения с одной из констант, передает управление на инструкцию, помеченную ключевым словом **case** (рис. 4.4).

Существенно:

- в скобках программист приводит выражение, которое в конечном итоге сводится к целому типу, например:

```
switch(sizeof(x)){...}
```

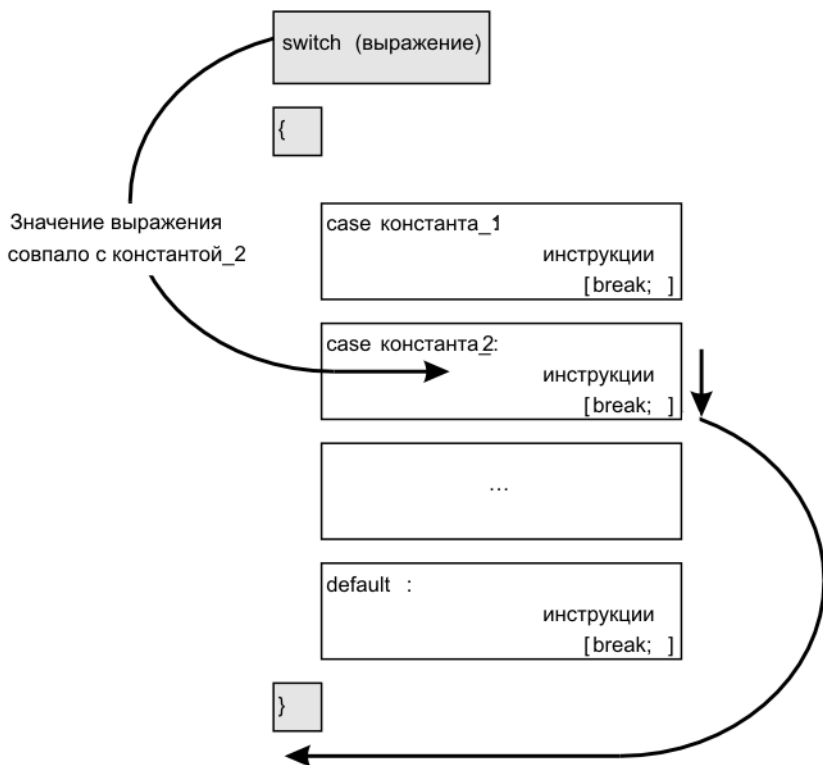


Рис. 4.4

- точки, куда передается управление, помечены ключевым словом **case**, каждой такой метке сопоставлена константа для сравнения со значением, вычисленным в скобках;

ЗАМЕЧАНИЕ

Для задания таких констант удобно использовать **enum**!

- обычное использование инструкции **case** предполагает, что выполняется только данная ветвь программы, поэтому эта ветвь должна каким-то образом завершиться. Для завершения выполняемой ветви используется: или инструкция **break**, которая передает управление на первую инструкцию, следующую за **switch** (см. рис. 4.4), или инструкция **return**, которая возвращает управление вызвавшей функции. Возможен и другой способ прерывания последовательного выполнения переключателя;
- при отсутствии инструкций **break** или **return** последовательно выполняется следующая ветвь переключателя. Иногда бывают случаи, когда про-

граммист намеренно из одной ветви передает управление следующей ветви, помеченной другим ключевым словом **case**:

```
case 'A':
```

```
case 'a':
```

```
    инструкция; //сюда компилятор передаст управление, если
                значение выражения совпало с кодом 'a' или 'A'
```

- если значение выражения не совпало ни с одной из констант, выполняется ветвь, помеченная ключевым словом **default** (таким образом программист может предусмотреть действие по умолчанию);

ЗАМЕЧАНИЕ 1

Присутствие ветви **default** необязательно. Если **default** отсутствует, управление передается на первую инструкцию после закрывающей скобки переключателя.

ЗАМЕЧАНИЕ 2

Ветвь **default** может быть в любом месте переключателя **switch** (не обязательно в конце);

- в переключателе можно объявлять и использовать локальные переменные, при этом лучше локализовать использование таких переменных в соответствующей ветви переключателя посредством ограничения области видимости (фигурных скобок). Например:

```
switch(x)
```

```
{
```

```
case 1:
```

```
    int a; //без инициализации компилятор позволит объявить
           переменную. Ее область видимости распространяется до
           закрывающей фигурной скобки переключателя (т. е. можно
           пользоваться во всех нижележащих ветвях)
```

```
{
```

```
    int b=1; //в этом месте без фигурных скобок
```

```
           компилятор выдаст сообщение об ошибке
```

//Замечание. Если вы используете инициализированные локальные переменные, то фигурные скобки обязательны, т. к. иначе область видимости переменной **b** распространялась бы на все нижерасположенные ветви переключателя и тогда появилась бы возможность использования неинициализированного значения **b** в других ветвях переключателя!

```

//используем b
        } //конец области видимости b
    break;

case 2:
    ...

default:
    ...

} //конец области видимости a

```

- перед закрывающей фигурной скобкой переключателя после метки (**case** или **default**) должна быть хотя бы одна пустая инструкция (;), иначе компилятору некуда передавать управление:

```

switch( x )
{
...
default: //ошибка
case 5:  //ошибка
}

```

```

switch( x )
{
...
default:  ;// OK!
//case 5: ;// OK!
}

```

- инструкция **switch** может быть вложенной:

```

int x,y;
enum {RED, GREEN, BLUE};
enum {ONE=1, TWO};
        //Вычисление значения переменной x
switch( x ) //внешний переключатель
{
case ONE:  //при значении x = ONE требуется выполнить разные действия
            в зависимости от значения переменной y
    switch( y ) //вложенный переключатель
    {
        case RED:
            ...

```

```
        break;
    case GREEN:
        ...
        break;
    ...
} //конец вложенного переключателя
break;
case TWO:
    ...
    break;
default:
    ...
} //конец внешнего переключателя
```

4.3. Инструкции цикла

Макс наблюдает картину: сидящий в клетке попугай говорит "кис-кис", находящаяся рядом кошка подходит к клетке, пытается просунуть морду между прутьев, получает клювом по носу, с криком "мяуууу" отпрыгивает на метр назад, далее попугай говорит "кис-кис"...

Уже четвертая итерация цикла, однако...

Любой цикл — это возможность повторения блока кода (итерация). Без итераций никакой язык программирования особой ценности не представляет.

Циклы в языке C/C++ можно реализовать инструкциями трех видов:

- ❑ **for**;
- ❑ **while**;
- ❑ **do..while**.

Каждая из этих инструкций, в свою очередь, выполняет инструкцию, называемую телом цикла, пока условие продолжения цикла не примет значение **false**, либо пока программист не прервет выполнение цикла другим способом.

ЗАМЕЧАНИЕ 1

Прервать последовательное выполнение инструкций любого цикла можно с помощью: **break** (передает управление на следующую за циклом инструкцию), **continue** (прерывает выполнение текущей итерации и переходит к следующей),

return (возвращает управление вызвавшей функции) и **throw** (генерирует исключение).

ЗАМЕЧАНИЕ 2

Если тело цикла состоит из нескольких инструкций, то такую последовательность следует заключать в фигурные скобки {...}. В некоторых организациях обязательным требованием является заключение тела цикла в фигурные скобки, даже если тело состоит из одной инструкции. Это позволяет избежать ошибок при модификации программы.

4.3.1. Инструкция **while**

Инструкция **while** обычно используется для так называемых нерегулярных циклов, т. е. циклов, для которых количество итераций заранее неизвестно.

Синтаксис:

```
while(условие_продолжения_цикла) {тело_цикла}
```

Условие продолжения цикла вычисляется в начале выполнения инструкции **while** (рис. 4.5). Выражение, которое используется в качестве условия, должно приводиться к целому типу. Если это условие сразу же не выполняется (false), то тело цикла не будет выполнено ни одного раза!

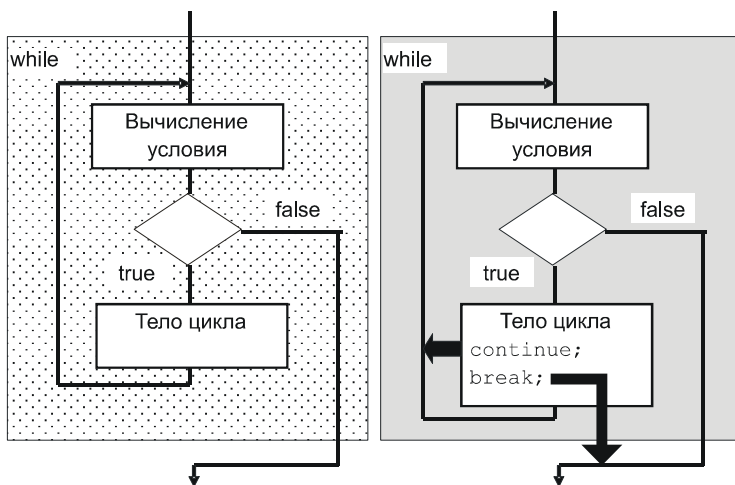


Рис. 4.5

Пример 1.

Требуется подсчитать сумму значений x/n , начиная с $n=1$, пока разность между двумя соседними значениями не станет меньше некоторого порогового значения. Решение представлено в листинге 4.1.

Листинг 4.1. Использование цикла while для вычисления суммы ряда

```

{
    const double delta = ... //задали значение порога
    int n = 1; //начальное значение n
    double x = ...; //сформировали значение x
    double sum = x; //подготовили переменную для суммирования
    while( (x/n - x/(n+1)) > delta) //пока два соседних значения
                                   отличаются больше, чем на delta
    {
        n++;
        sum += x / n; //копим сумму
    }
}

```

Пример 2.

Требуется вводить и подсчитывать количество введенных символов, пока пользователь не введет символ *. Решение представлено в листинге 4.2.

Листинг 4.2. Использование цикла while для подсчета введенных символов

```

{
    int n=0; //инициализация счетчика
    char ch; //переменная для ввода
    while(std::cin>>ch, ch!='*') //использование в условии
                                   оператора "запятая"
    { n++; }
}
std::cout<<"n="<<n;

```

Пример 3.

Модифицируем предыдущее задание — подсчитаем общее количество введенных пользователем символов (в т. ч. пробелов и запятых). Решение представлено в листинге 4.3.

Листинг 4.3. Подсчет общего количества введенных символов (в т. ч. пробелов и запятых)

```

{
    int iTotal=0, iSpaces=0, iCommas=0; //счетчики
    char ch; //переменная для ввода

```



```
while(std::cin>>ch, ch!='*')
{
    switch(ch)
    {
        case ' ': //пробел
            iSpaces++;
            break;
        case ',': //запятая
            iCommas++;
            break;
    }
    iTotal++; //в общем количестве следует учесть любой
                СИМВОЛ
}
std::cout<<"Total "<<iTotal<< std::endl;
std::cout<<"Spaces "<<iSpaces<< std::endl;
std::cout<<"Commas "<<iCommas<< std::endl;
}
```

ПРЕДОСТЕРЕЖЕНИЕ 1

Одна из самых неприятных особенностей цикла заключается в возможности образования бесконечного цикла. Будьте бдительны!

Например:

```
while(i=1) {тело_цикла} //часто встречаемая ошибка использования
оператора присваивания вместо оператора сравнения на
равенство. Такое условие всегда будет true, и, соответственно,
ваша программа заиклится!
```



Это тоже очень распространенная ошибка.

Подумайте, сколько раз выполнится цикл?

```
int i=100;
while(i>0) ;
{ i--; }
```

ПРЕДОСТЕРЕЖЕНИЕ 2

Будьте осторожны с условием, в котором фигурируют плавающие переменные (**float** или **double**), т. к. условие типа **while**(<выражение> != 1.1111) может из-за округления превратить цикл в бесконечный.

ЗАМЕЧАНИЕ

Иногда программист в явном виде задает бесконечный цикл: `while(true)` {тело_цикла}. Такой цикл обязательно должен быть прерван, например, инструкцией **break**.



Подумайте, имеет ли смысл такой цикл?

```
while(false){тело_цикла}
```

Если программист никаким образом не вмешивается в естественное выполнение цикла, то выход происходит только в том случае, когда условие продолжения цикла принимает значение `false`. Бывают ситуации, когда естественное выполнение цикла требуется прервать при возникновении некоторого дополнительного условия в теле цикла. Для этих целей можно воспользоваться инструкциями **break** и **continue** (см. рис. 4.5).

Использование инструкции **continue** демонстрирует листинг 4.4, а инструкции **break** — листинг 4.5.

Листинг 4.4. Подсчет количества символов без учета пробелов

```
{
    int iTotat=0; //счетчик символов без учета пробелов
    char ch; //переменная для ввода
    while(std::cin>>ch, ch!='')
    {
        if(ch==' ') continue; //прерываем текущую итерацию,
                               //передаем управление на следующую итерацию.
                               //При этом остаток тела цикла игнорируется
        iTotat++; //а все остальные символы считаем
    }
    std::cout<<"Without spaces"<<iTotat<< std::endl;
}
```

Листинг 4.5. Подсчет количества символов до первого пробела

```
{
    int iTotat=0;
    char ch;
    while(std::cin>>ch, ch!='')
```

```
{  
    if(ch==' ') break;  
    iTotal++;  
}  
std::cout<<"Up to first space "<<iTotal<< std::endl;  
}
```

4.3.2. Инструкция *do...while*

Инструкция *do...while* аналогична инструкции *while* (рис. 4.6).

Синтаксис:

```
do {тело_цикла} while (условие_продолжения_цикла);
```

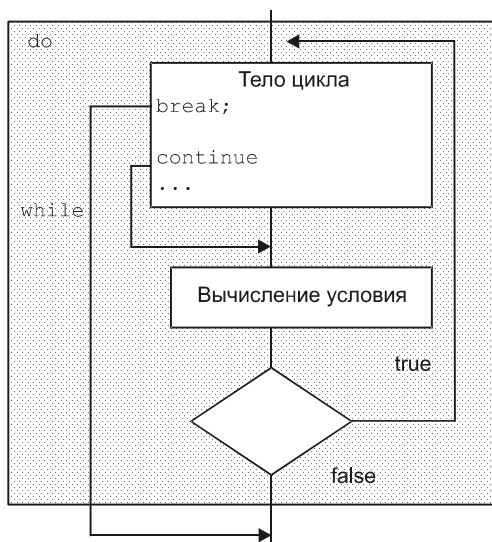


Рис. 4.6

Основное отличие (и причина ошибок) состоит в том, что сначала выполняется тело цикла, а потом проверяется условие, поэтому независимо от значения условия тело цикла всегда выполняется хотя бы один раз. Как следствие, во многих случаях оказывается необходимой дополнительная проверка внутри тела цикла. В листинге 4.6 демонстрируется подобная ситуация.

Листинг 4.6. Вычисление суммы натуральных чисел до указанного пользователем значения без проверки введенного значения

```
{  
    int n; //в этой переменной пользователь сформирует значение  
    std::cout<<"Input integer positive value: "; //предложим ему  
                                                ввести целое положительное значение  
    std::cin>>n; //но он может не послушаться и ввести отрицательное!  
    int sum=0; //здесь будем копить сумму  
    do{  
        sum +=n; //при отрицательном значении n переменная sum  
                все равно будет модифицирована!  
        n--;  
    } while(n>0); //условие продолжения цикла  
}
```

Если пользователь ввел отрицательное значение, то результат получен неправильный, поэтому в теле цикла нужна дополнительная проверка. Модифицируем предыдущий пример с учетом проверки (листинг 4.7).

Листинг 4.7. Вычисление суммы натуральных чисел с проверкой введенного значения

```
{  
    int n;  
    std::cout<<"Input integer positive value: ";  
    std::cin>>n;  
    int sum=0;  
    do{  
        if(n<=0) break; //дополнительная проверка  
        sum +=n;  
        n--;  
    } while(n>0); //условие продолжения цикла  
}
```

Бывают ситуации, когда логично использовать цикл **do..while**. Например, пользователю предлагается ввести два значения: *top* и *bottom*, причем значение *top* должно быть больше, чем *bottom* (но пользователь может пренебречь этим условием и ввести значения с точностью до наоборот). С помощью цикла **do..while** заставим его вводить значения до тех пор, пока он не сформирует их правильно (листинг 4.8).

Листинг 4.8. Использование цикла `do...while` для правильного формирования вводимого пользователем диапазона

```
{
    int top, bottom;
    do
    {
        std::cout<<"Input bottom:";
        cin>>bottom;
        std::cout<<"Input top (it must be larger than bottom):";
        cin>>top;
    }while(top<=bottom); //а если пользователь ввел неправильные
                           значения, отправляем его на новую итерацию..
    ...
}
```

4.3.3. Инструкция **for**

Если инструкции **while** и **do...while** используются в основном (хотя это совершенно необязательно) для нерегулярных циклов (когда количество повторений явно не задано), то инструкция **for** служит (хотя это тоже необязательно) для организации регулярного цикла (с известным числом повторений).

Синтаксис:

```
for([вычисление_инициализаторов] ; [выражение1, обычно условие] ;
    [выражение2]) {тело_цикла}
```

В отличие от **while** и **do...while** в инструкции **for** все части, управляющие циклом (переменная цикла, условие продолжения и выражение, модифицирующее переменную цикла), сосредоточены в одном месте, что улучшает восприятие кода. Порядок выполнения инструкции **for** (рис. 4.7):

1. Сначала один раз вычисляется инициализирующая часть (обычно здесь задаются начальные значения).
2. Затем вычисляется выражение 1 (условие продолжения цикла), и если условие принимает значение **true**, то выполняется текущая итерация:
 - выполняется тело цикла;
 - вычисляется выражение 2 (обычно здесь модифицируется переменная цикла).
3. Итерации повторяются, пока условие не примет значение **false**.

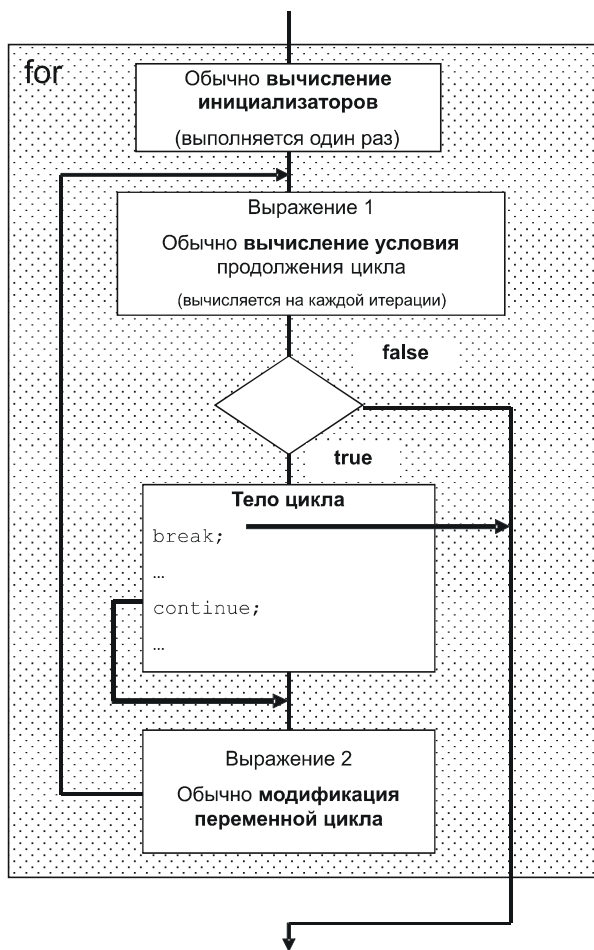


Рис. 4.7

Пример 1 (самое распространенное применение **for**).

Реализуем посредством **for** задачу из *разд. 4.3.2* (см. листинг 4.6).

Решение представлено в листинге 4.9. Если пользователь вводит отрицательное значение, то тело цикла не выполняется.

Листинг 4.9. Вычисление суммы n натуральных чисел

```

{
    int n;
    std::cout<<"Input integer positive value: ";

```

```

std::cin>>n;
int sum=0;
for(int i=1; i<n; i++) //i-переменная цикла, с помощью которой
                        перебираем диапазон значений от 1 до n
{
    sum +=i;
}

```

Пример 2.

Переменная цикла не обязательно должна быть типа **int** (листинг 4.10).

Листинг 4.10. Вывод на экран всех заглавных английских букв по алфавиту

```

{
    for(char c='A'; c<='Z'; c++) //c-переменная цикла
    {
        std::cout<<c;
    }
}

```

Специфика использования инструкции **for**:

- ❑ условие продолжения цикла вычисляется на каждой итерации, поэтому не стоит в этом месте заставлять компилятор всякий раз вычислять сложные выражения (если есть такая возможность, лучше вычислить его один раз до начала цикла):

```
for(int i=0; i< x*y*z; i++){тело_цикла}
```

Если переменные *x*, *y*, *z* в теле цикла не изменяются, то эффективнее такую задачу реализовать следующим образом:

```
int n = x*y*z;
for(int i=0; i< n; i++){тело_цикла}
```

- ❑ три составляющих цикла разделяются обязательными символами (точка с запятой), а сами составляющие (все три) являются необязательными;



Если необязательные составляющие опустить, получится бесконечный цикл.

Подумайте, как прервать выполнение такого цикла?

```
for(;;){...}
```

- ❑ в любой части цикла может быть несколько выражений. Они должны быть разделены запятой, а не точкой с запятой:

```
int i;  
double res;  
for(i=0, res=1 ; i<100 ; i++, res+=res/i){тело_цикла}
```

- ❑ не стоит смешивать в инициализирующей части выражения и объявления (компилятор воспримет все как список объявлений, проигнорирует в приведенном примере **double**, выдаст предупреждение и будет интерпретировать **sum** как **int**):

```
for(int i=0, double sum=0; i<100; i++) { sum+=...; }
```

- ❑ видимость переменных, объявленных в инициализирующей части **for**, распространяется до закрывающей фигурной скобки тела цикла.

ЗАМЕЧАНИЕ

Специфика Microsoft (до версии VC.net 2003 включительно): переменные, объявленные в инициализирующей части **for**, имеют такую же область видимости, как если бы они были объявлены непосредственно перед **for**;

- ❑ тело цикла может быть пустым:

```
for(int i=0, sum=0; i<n; sum+=i, i++) ;
```

- ❑ обычно выражение 2 (см. синтаксис инструкции **for**) используется для модификации переменной цикла, но это не обязательно;
- ❑ переменная цикла в принципе может быть любого типа, но будьте осторожны с переменными цикла плавающего типа, т. к. из-за ограниченности разрядной сетки при большом количестве итераций можно накопить ошибку!

Для прерывания выполнения инструкции **for** можно использовать инструкции **break** и **continue**.

Пример 1.

Пусть требуется определить в заданном диапазоне количество значений, удовлетворяющих некоторому условию (листинг 4.11).

Листинг 4.11. Подсчет количества чисел, делящихся нацело на 3

```
int iUp = 100, iDown = 0;  
for(int nTotal = 0, i = iDown; i<iUp; i++)  
{  
    if( i%3 ) continue; //если
```



```

        остаток от целочисленного деления не равен нулю
        (условие=true), то считать его не нужно, поэтому
        переходим на следующую итерацию
nTotal++;
}

```

Пример 2.

Допустим, необходимо ввести максимально N целых чисел и просуммировать введенные положительные значения, но если очередное введенное число окажется отрицательным, то нужно прекратить суммировать и выйти из цикла. Решение представлено в листинге 4.12.

Листинг 4.12. Ввод и суммирование целых чисел

```
int Sum = 0;
for(int i=0; i<N; i++)
{
    int n;
    cin>>n;
    if(n<0) break; //если очередное
                   //введенное значение оказалось
                   //отрицательным, то выход из цикла
    Sum += n;
}
```

Пример 3.

Модифицируем предыдущий пример. Введем все N чисел, но просуммируем только положительные, а отрицательные проигнорируем. Решение представлено в листинге 4.13.

Листинг 4.13. Суммирование только положительных целых чисел

```
int Sum = 0;
for(int i=0; i<N; i++)
{
    int n;
    cin>>n;
    if(n<0) continue; //если
```

очередное введенное значение оказалось

отрицательным, то переходим к следующей итерации (т. е. игнорируем его)

```
    Sum += n;  
}
```

4.4. Инструкции безусловного перехода: *break, continue, return, goto*

Инструкция **break** прерывает выполнение **switch**, **while** и **for** и передает управление на инструкцию, следующую за ними.

ЗАМЕЧАНИЕ

Если имеет место вложенность, то прерывается самая внутренняя по отношению к **break** инструкция.

Инструкция **continue** прерывает выполнение **while** и **for** и переходит к следующей итерации.

Инструкция **return** прерывает выполнение текущей функции и возвращает управление вызвавшей функции.

Инструкция **goto** осуществляет безусловный переход (только в пределах одной функции!).

Синтаксис:

```
goto <идентификатор>;
```

Например:

```
{  
    goto M;  
    ...  
M: инструкция  
}  
    //или  
{  
N: инструкция  
    ...  
    goto N;  
}
```

ЗАМЕЧАНИЕ

Без инструкции **goto** в подавляющем большинстве случаев можно обойтись (Дейкстра). Не рекомендуется использовать ее, поскольку в этом случае неочевидно, из какой точки программы было передано управление!

Одним из немногих разумных случаев использования инструкции **goto** является выход из вложенного цикла или switch-инструкции (т. к. **break** прекращает выполнение только самого внутреннего цикла или switch-инструкции):

```
for(int i=0; i<n; i++)
{
    for(int j=0; j<m; j++)
    {
        if(условие) goto found;
    }
    ...
}
found:...
```



Глава 5

Препроцессор. Заголовочные файлы

5.1. Директивы препроцессора

Получение исполняемого модуля из исходного текста происходит в несколько этапов (*см. разд. 1.2.1*). На первом этапе с исходным текстом программы работает специальная программа — препроцессор. Основная его цель — закончить формирование исходного текста программы на языке C/C++, в частности, препроцессор должен выполнить предназначенные ему программистом директивы (список директив представлен в табл. 5.1). И только затем окончательно сформированный текст программы поступает на обработку компилятору.

Таблица 5.1. Список директив препроцессора

Директива	Пояснение
#include	Включает в исходный текст обрабатываемого файла содержимое других файлов, указанных в директиве (в частности, содержащих интерфейс стандартной библиотеки)
#define	Задаёт макроподстановки и делает определёнными имена для препроцессора
#undef	Отменяет действие директивы #define
#if	Директивы условной трансляции позволяют на стадии препроцессора формировать из исходного текста файла разный конечный текст, который в дальнейшем поступает на обработку компилятору. Это, в свою очередь, позволяет получать различный выполняемый код, не модифицируя исходный текст программы
#elif	
#else	
#endif	
#ifdef	
#ifndef	

Таблица 5.1 (окончание)

Директива	Пояснение
#pragma	Позволяет настраивать компилятор с учетом специфических особенностей конкретной машины или операционной среды (эти особенности и варианты директивы #pragma индивидуальны для каждого компилятора)
#line	Позволяет включать номера строк исходного кода заимствованных файлов в диагностику компилятора об ошибках и предупреждениях
#error	Обычно включается между директивами #if...#endif для проверки какого-либо условия на этапе препроцессора. При выполнении такого условия компилятор выводит сообщение, указанное в директиве #error и останавливается

ЗАМЕЧАНИЕ

В реализации препроцессоров семейства Microsoft Visual Studio имеются еще две директивы: **#import** и **#using**, которые в других реализациях могут отсутствовать.

Специфика использования директив препроцессора:

- ❑ директивы препроцессора пишутся каждая на отдельной строке;
- ❑ никакие разделители, вроде признака конца C/C++ инструкции (точки с запятой) не нужны! Но если директива слишком длинная, то для переноса на другую строку следует применять символ обратной косой черты \, при этом везде препроцессор будет удалять начальные (левые) пробелы;
- ❑ большинство директив препроцессора можно использовать в любом месте файла, при этом их действие на препроцессор распространяется, начиная от того места, где они использованы, до конца файла;
- ❑ комментарии первого рода в стиле C++ — две косые черты // в директивах препроцессора понимает не каждая реализация (не каждый препроцессор).

5.2. Директива **#define**

Директива **#define** позволяет программисту задавать макроопределения (или макросы).

Термином *макроопределение* обозначают возможность сопоставления заданному программистом идентификатору текстового фрагмента. Препроцессор каждый раз, встречая в обрабатываемом тексте этот идентификатор, будет выполнять макроподстановку (макрорасширение), т. е. заменять имя указанным фрагментом. Таким образом, препроцессор формирует окончательный текст, который поступает на обработку компилятору.

Во времена ранних версий Си считалось, что использование макросов улучшает визуальное восприятие текста программы, т. к. во-первых — позволяет вводить осмысленные обозначения для констант, а во-вторых — укрупняет текстовые фрагменты, заменяя их на короткий идентификатор.

Но в связи с развитием языка и появлением новых стандартов C++ теперь принято считать (Бьерн Страуструп — создатель C++), что злоупотребление макросами чревато возникновением трудно выявляемых ошибок. Я бы перефразировала эту рекомендацию следующим образом: оставим написание макроопределений профессионалам, а сами будем осмысленно использовать созданное профессионалами! Заглядывая в заголовочные файлы стандартной библиотеки C/C++, а также заголовочные файлы Windows и MFC (что бывает очень полезным), вы будете встречать директиву **#define** довольно часто.

5.2.1. Использование директивы **#define**

Простое макроопределение

Синтаксис:

#define идентификатор_макро тело_макро

Препроцессор просматривает исходный текст и заменяет каждое вхождение лексемы "идентификатор_макро" на лексему (совокупность лексем) "тело_макро", осуществляя макроподстановки.

Пример.

```
#define FIRST 1 //эта директива может быть приведена в любом месте
                  до ее первого использования
```

```
int main()
{
    if(n == FIRST) {...} //препроцессор заменит FIRST на 1 и на
                        компиляцию будет отправлен текст if(n == 1)
}
```

ЗАМЕЧАНИЕ

Задать макроподстановку можно двумя способами: с помощью директивы **#define** или посредством соответствующего ключа в опциях командной строки компилятору (в VC таким ключом является \D). Использование в тексте программы **#define** FIRST 1 эквивалентно \DFIRST=1.

Вложенные макросы

Препроцессор просматривает текст несколько раз, осуществляя макроподстановки на каждом проходе. После первой подстановки снова выполняется

просмотр уже расширенного макросами текста, что позволяет организовывать вложенные макросы.

Пример.

```
#define FALSE 0
#define TRUE !FALSE
int main()
{
    if(выражение == FALSE) {...} //препроцессор заменит на
                                if(выражение == 0) {...}
    if(выражение == TRUE) {...} //при первом просмотре препроцессор
                                заменит на if(выражение == !FALSE),
                                при втором просмотре на:
                                if(выражение == !0), что
                                эквивалентно if(выражение ==1)
}
```

Макрос с параметрами

Синтаксис:

```
#define идентификатор_макро(список_параметров) тело_макро
```

Далее, встретив в тексте программы указанный идентификатор, совершая макроподстановку, вместо формальных имен, приведенных в списке параметров макроса, препроцессор будет просто подставлять выражения, указанные программистом.

Пример 1.

Рассмотрим макрос, который из четырех однобайтовых значений формирует четырехбайтовое (рис. 5.1).

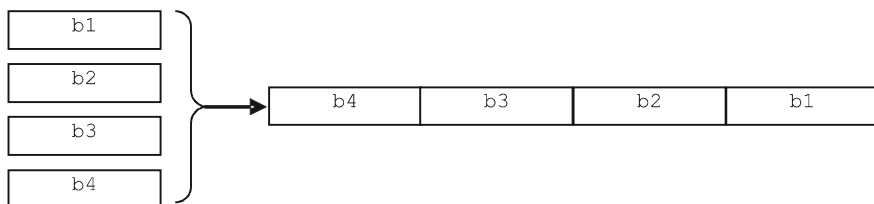


Рис. 5.1

```
#define MAKE(b1, b2 ,b3, b4) static_cast<unsigned char>(b1) | \
    static_cast<unsigned char>(b2) << 8 | \
```

```
static_cast<unsigned char>(b3) << 16 | \
(b4) << 24
int main()
{
    int res = MAKE(1,2,3,255);
}
```



Подумайте: зачем используется явное приведение типа к **unsigned char** первых трех параметров макроса и почему оно необязательно для четвертого.

Подсказка: каким был бы результат без использования явного приведения типа в случае:

```
int res = MAKE(0xff,1,2,3);
```

Пример 2.

Рассмотрим макрос для получения квадрата любого числа:

```
#define SQUARE(x) x*x
int main()
{
    int z = 2;
    int y = SQUARE(z); //препроцессор заменит SQUARE(z) на z*z и
                        //на компиляцию пойдет строка: y = z*z;
}
```

ЗАМЕЧАНИЕ 1

Существенным моментом при написании тела макроса с параметрами является использование скобок, в частности — заключение параметра в теле макроса в скобки.

Например:

```
//без скобок
int z = 2, y;
y = SQUARE(z-1); //препроцессор сформирует для компилятора строку:
                  y = z-1 * z-1; и результат будет совсем
                  не таким, как задумал программист

//используем скобки
#define SQUARE(x) (x)*(x)
int z = 2, y;
y = SQUARE(z-1); //теперь препроцессор сформирует строку:
                  y = (z-1) * (z-1);
```




Подумайте, каковы были бы результаты макрорасширения и каким образом можно исправить ситуацию?

```
#define SUM(x,y) x + y
int a=2,b=3;
int c = SUM(a,b) * 5;
```

ЗАМЕЧАНИЕ 2

Некоторые ошибки не устранить никакими скобками.

Например:

```
#define SQUARE(x) (x)*(x)
int z = 2, y;
int y = SQUARE(z++); //на компиляцию будет отправлена строка:
int y = (z++) * (z++); и значение z будет
инкрементировано дважды, чего программист,
скорее всего, не предполагал
```

ЗАМЕЧАНИЕ 3

Пробел перед открывающей скобкой списка параметров вызовет ошибку, т. к. препроцессор в таком случае не сможет отличить список параметров от тела макроса.

Например:

```
#define SQUARE_(x) x*x //ошибка
```

ЗАМЕЧАНИЕ 4

Как и для обычных макросов, после первой подстановки выполняется повторный просмотр полученного текста, что позволяет создавать сложные вложенные макросы с параметрами.

Пустые макросы

Синтаксис:

```
#define идентификатор_макро
```

Встречая в тексте идентификатор пустого макроса, препроцессор просто исключает его из текста программы. Такие пустые макросы используются не для макрорасширений, а обозначают определенное для препроцессора состояние данного идентификатора. В основном такие макроопределения применяются для условной компиляции кода (см. разд. 5.4 и 5.5).

ЗАМЕЧАНИЕ

Определенность идентификатора для препроцессора можно задать также с помощью опций командной строки компилятору. В VC **#define** идентификатор_макро эквивалентно \Идентификатор_макро.

Конкатенация макросов

С помощью сочетания символов **##** можно слить несколько лексем в одну. Это означает, что препроцессор просто объединит указанные строки в одну значимую единицу для компилятора.

Пример 1.

Можно компоновать идентификаторы из отдельных частей:

```
#define NUM(name, number) name##_##number

int main()
{
    int NUM(n,1) = 1; //препроцессор подставит вместо NUM(n,1) n_1 и
                       //на компиляцию пойдет строка int n_1 = 1;
    double NUM(d,1) = 1.1; //на компиляцию
                           //пойдет строка double d_1 = 1.1;
}
```

Пример 2.

Макрос **_TEXT** подставляет префикс **L**, позволяя формировать расширенные (UNICODE) символы или строки:

```
#define _TEXT(x) L##x

int main()
{
    std::wcout<<_TEXT("ABC"); //препроцессор вместо _TEXT("ABC")
                               //подставит L"ABC"
}
```

5.2.2. Предопределенные макросы

Язык C/C++ предоставляет программисту набор макросов (некоторые из них приведены в табл. 5.2) для получения разнообразной информации во время компиляции.

Пример.

```
int main()
{
```

```
std::cout<<"Current file "<<__FILE__<<
"\nLine "<<__LINE__<<
"\nTime"<<__TIME__;
```

```
}
```

Таблица 5.2. Предопределенные макросы

Идентификатор	Пояснение
<code>__cplusplus</code> (специфика VC)	Определен, если исходный файл должен компилироваться как C++, а не Си
<code>__DATE__</code>	Преобразуется в строку, заключенную в кавычки и содержащую дату компиляции
<code>__FILE__</code>	Преобразуется в строку, заключенную в кавычки и содержащую имя исходного файла с учетом полного пути
<code>__LINE__</code>	Преобразуется в строку, заключенную в кавычки и содержащую номер текущей строки в исходном файле
<code>__TIME__</code>	Преобразуется в строку, заключенную в кавычки и содержащую время последней компиляции
и т. д.	

5.2.3. Диагностический макрос *assert*

Стандартная библиотека предоставляет программисту возможность отладки своей программы посредством использования макроса **assert**.

Макрос имеет вид:

```
assert(выражение);
```

Он позволяет программисту при отладке программы отследить непредусмотренные задачей ситуации. Этот макрос определен в заголовочном файле `<cassert>`.

Например: программист предполагает, что, исходя из задачи, некоторая переменная `x` ни при каких обстоятельствах не должна становиться положительной. Но чего только в реальной жизни не бывает, поэтому на всякий случай в некоторых местах программы дотошный программист включает проверку:

```
assert(x<0);
```

Тогда в `DEBUG`-версии производятся следующие действия:

- 1. Макрос вычисляет свой аргумент, и если результат равен `false` (0), то выводит полную диагностику о том, что произошло:
`assertion failed: x<0, file:...\имя_файла.cpp, line номер_строки,`

2. Вызывает функцию аварийного завершения программы `abort()`. Если вы пользуетесь компилятором Microsoft Visual C++, то при этом еще появляется диалоговое окно диагностики **Debug error**.

ЗАМЕЧАНИЕ

В RELEASE-версии макрос ничего не делает, т. е. никаких проверок не осуществляет.

5.2.4. Рекомендации

Начинающему программисту Б. Страуструп рекомендует воздерживаться от написания собственных макросов, т. к. это может привести к появлению трудновывяемых (а иногда и неустранимых) ошибок. Создание макросов — это прерогатива профессионалов (разработчиков библиотек). Обычному программисту следует стремиться использовать другие (более безопасные) средства языка, предоставляющие практически те же возможности, что и макросы (табл. 5.3).

Таблица 5.3. Средства C++, альтернативные директиве `#define`

Вместо макроса	Пользуйтесь понятиями
Для определения констант: <code>#define YES 0</code>	<code>const</code> (см. разд. 3.10.1) и для целочисленных констант <code>enum</code> (см. разд. 3.3): <code>const int YES=0;</code> <code>enum {YES, NO};</code>
Для того чтобы избежать расходов на вызов функции: <code>#define SQUARE(x) (x)*(x)</code>	<code>inline</code> (см. разд. 8.1.4): <code>inline SQUARE(int x) {return x*x;}</code>
Для определения семейства функций или классов	<code>template</code> (эта тема в данной книге не рассматривается)

ЦИТАТА ИЗ Б. СТРАУСТРУПА

"Предпочитайте компилятор препроцессору!"

5.3. Директива `#undef`

Если программист использовал идентификатор для макроподстановки, или идентификатор является именем пустого макроса, или программист использовал это имя с ключом `\D` в командной строке компилятора, то препроцес-

сор считает такой идентификатор определенным и действующим до конца файла или до тех пор, пока в файле не встретится директива **#undef**.

Поэтому один и тот же идентификатор может означать для компилятора разные лексемы, т. е. одному и тому же имени макроса можно поставить в соответствие разные тела. Можно отменить действие макроопределения с помощью директивы **#undef**.

Пример.

```
#define MESSAGE "Hello"
    std::cout<< MESSAGE; //будет выведено Hello

#undef MESSAGE //отмена действия идентификатора MESSAGE
    std::cout<<MESSAGE; //ошибка: т. к. идентификатор стал
        неопределенным и препроцессор вместо того, чтобы
        подставлять тело макроса, отправил эту строку,
        как есть, компилятору, а тот не понял,
        что такое MESSAGE

#define MESSAGE "Bye"
    std:: cout<< MESSAGE; //будет выведено Bye
```

ЗАМЕЧАНИЕ 1

Директива **#undef** идентификатор будет корректно выполняться независимо от того, был ли до того определен идентификатор или нет.

ЗАМЕЧАНИЕ 2

Если одному и тому же идентификатору (не отменяя его действие с помощью **#undef**) программист сопоставляет новое тело, то препроцессор ANSI Си выдает ошибку, а препроцессор C++ предупреждает о повторном определении макроса и использует последнее значение.

Например:

```
#define MESSAGE "Hello"
    std::cout<< MESSAGE; //будет выведено Hello

#define MESSAGE "Bye" //здесь препроцессор C++ выдаст
        предупреждение о повторном определении
        идентификатора MESSAGE (macro redefinition)
    std:: cout<< MESSAGE; //но использовано будет последнее
        и выведено Bye
```

5.4. Директивы **#ifdef**, **#ifndef**, **#else** и **#endif**

Условные директивы **#ifdef**, **#ifndef** используются для проверки: является ли для препроцессора в данный момент указанный идентификатор определенным (определенность идентификатора задается с помощью директивы **#define** или посредством ключа **\D** компилятору). Этот механизм позволяет препроцессору в зависимости от определенности идентификатора формировать разный текст, который затем отправляется на компиляцию, т. е. манипулируя только определенностью идентификатора, можно получать разный результирующий код.

Синтаксис:

```
#ifdef идентификатор_макро
    инструкции //если идентификатор является определенным для
                препроцессора, этот блок кода будет вставлен
                в текст, который пойдет на компиляцию
#else
    инструкции //иначе на компиляцию препроцессор отправит
                этот текст
#endif //признак конца условного блока для препроцессора обязателен!
```

ЗАМЕЧАНИЕ

Директивы препроцессора **#ifdef** и **#else** не имеют никакого отношения к инструкциям **if...else** языка Си! Это механизм совершенно другого этапа (формирования текста, а не собственно компиляции).

Пример 1.

В **DEBUG**-версии проекта программист для своего удобства может выводить какие-то промежуточные значения на печать, но заказчика эти отладочные распечатки не интересуют, поэтому логично вставлять их только в **DEBUG**-версию проекта:

```
{//в VC в опциях командной строки компилятору в отладочной версии
    определено имя _DEBUG
    int x=выражение;
#ifdef _DEBUG // если имя _DEBUG является определенным (отладочная
                версия исполняемого файла), то следующая строка
                будет отправлена на компиляцию, а иначе
                препроцессор ее проигнорирует
        cout<<x;
#endif
}
```

Пример 2.

Хорошим примером использования директив условной трансляции является способ получения простых (однобайтовых) или расширенных (двухбайтовых) символов без модификации исходного текста в VC (это специфика Microsoft). В файле <tchar.h> все понятия, так или иначе связанные со строками, заключены в директивы условной трансляции:

```
#ifndef _UNICODE //если определено имя _UNICODE
    typedef wchar_t TCHAR; //везде, где программист использует
                           псевдоним TCHAR, компилятор будет
                           иметь в виду wchar_t
    #define _T(x) L##x     //везде, где препроцессор встретит
                           макрос _T, он подставит префикс L
    #define _tmain wmain   //вместо макроса _tmain препроцессор
                           подставит wmain (версия функции main
                           для расширенных символов)
    #define _tcscpy wcscpy //вместо макроса _tcscpy препроцессор
                           подставит wcscpy (версия функции
                           стандартной библиотеки для копирования
                           расширенных символов)
    ...
#else //иначе тем же самым псевдонимам и макроподстановкам будут
    сопоставлены варианты для однобайтовых символов
    typedef char TCHAR; //псевдониму TCHAR будет соответствовать char
    #define _T(x) x     //префикса L не будет
    #define _tmain main //_tmain превратится в main
    #define _tcscpy strcpy //_tcscpy превратится в strcpy (версия
                           копирования однобайтовых строк символов)
    ...
#endif
```

В табл. 5.4 приведен пример исходного текста (столбец 1) и результат работы препроцессора (столбец 2 либо 3) в зависимости от определенности идентификатора `_UNICODE`.

Пример 3.

Директивы условной трансляции могут быть вложенными:

```
#define VERSION
#define BETA
```

```
#ifdef VERSION //в программе используется контроль версий
#ifdef ALFA
std::cout<<"Alfa version";
#else
std::cout<<"Beta version"; //эта строка будет отправлена
                             препроцессором на компиляцию
#endif
#endif
```

Таблица 5.4. Примеры использования средств <uchar.h>

Фрагмент программы, написанный с использованием макросов	Если имя _UNICODE для препроцессора определено, препроцессор сформирует:	Если имя _UNICODE для препроцессора не определено, на компиляцию пойдет:
#include <uchar.h> int _tmain() { TCHAR c=_T('Ф'); TCHAR ar[10]; tcscpy(ar,_T("ABC")); }	int wmain() { wchar_t c=L'Ф'; wchar_t ar[10]; wcscpy(ar, L"ABC"); }	int main() { char c='Ф'; char ar[10]; strcpy(ar, "ABC"); }

ЗАМЕЧАНИЕ

Директивы `#ifdef`, `#else` устарели и продолжают существовать только для совместимости с предыдущими версиями языка. Предпочтительным является использование директив, приведенных в следующем разделе.

5.5. Директивы `#if`, `#elif`, `#else`, `#endif`. Оператор препроцессора *defined*

Условные директивы `#if`, `#elif`, `#else` и `#endif` работают так же, как и обычные условные инструкции C/C++, только действуют они на стадии формирования исходного текста, а не выполнения. По сравнению с `#ifdef` и `#ifndef` директивы `#if`, `#elif` предоставляют альтернативный, более гибкий способ формирования условий на этапе препроцессора.

Оператор препроцессора `defined` используется только с директивами `#if`, `#elif` и служит для выявления определенности идентификатора.

Специфика использования операторов **#if**, **#elif**:

- в простейшем случае директиву **#if** и оператор **defined** можно употреблять для выяснения определенности идентификатора, при этом:

#ifdef идентификатор эквивалентно **#if defined** идентификатор

#ifndef идентификатор эквивалентно **#if !defined** идентификатор

Пример 1.

Перепишем пример из предыдущего раздела, используя директиву **#if** и оператор препроцессора **defined**:

```
#if defined _DEBUG //если DEBUG-версия проекта
    cout<<x; //вывод отладочных значений
#endif
```

Пример 2.

Рассмотрим, как можно средствами препроцессора устранить зависимость от платформы:

```
#if defined (_WIN32) //если 32-разрядная платформа
    typedef int int32; //вводим псевдоним для int
#else
    typedef long int32; //иначе на 16-разрядной платформе
                        //тому же самому псевдониму сопоставляем long
#endif
int32 value; //гарантировано 32 разряда независимо от платформы
```

- одним из преимуществ использования **#elif** является возможность многократного использования этой директивы в сложных конструкциях, следующих за директивой **#if**.

Например, пусть программа ориентирована на нескольких потребителей, при этом для каждого клиента в определенном месте программы нужно реализовать разную функциональность. Чего в таких ситуациях точно не следует делать, так это для каждого клиента создавать отдельный проект и дублировать код. Гораздо удобнее и надежнее в такой ситуации в соответствующем месте использовать директивы условной трансляции и управлять получением результирующего кода, делая один из идентификаторов определенным:

```
#define CLIENT2
int main()
{
```

```
...//действия, не зависящие от конкретного клиента
```

```
//действия, специфичные для каждого клиента
```

```
#if defined CLIENT1 //если создаем версию программы
    для пользователя №1
    ...//в текст программы будет включен текст, специфичный
    для пользователя №1
#elif defined CLIENT2 //а если для пользователя №2
    ...//в текст программы будет включен текст, специфичный
    для пользователя №2
#elif defined CLIENT3
    ...//для пользователя №3
#else
    std::cout<<"Client not supported!";
#endif
}
```

- также, в отличие от директивы **#ifdef**, которая умеет проверять только определенность идентификатора макроса, в директиве **#if** можно использовать сложное выражение, где могут фигурировать как идентификаторы, так и логические операторы и целочисленные литералы. При этом следует помнить, что вычислять это выражение будет препроцессор, поэтому в выражении можно использовать только константы. Например:

```
#if defined CLIENT1 || defined CLIENT2 //если определен любой
    из идентификаторов, то будет компилироваться
    следующая последовательность инструкций
    ...
#endif
```

- часто в качестве идентификаторов в директиве **#if** используются имена, определяющие тип процессора или операционной системы. Это позволяет учитывать особенности архитектуры или операционной системы на этапе компиляции, что, в свою очередь, позволяет программисту без изменений адаптировать исходный текст программы к разным средам:

```
#if defined _WIN32_WINNT && _WIN32_WINNT >= 0x0400 //если версия
    Windows NT4 или более поздняя
    // код, специфичный для NT
#elif defined _WIN32_WINDOWS && WINVER>=0x0410
    // код, специфичный для Windows 98 и более поздних версий
#endif
```

5.6. Директива `#include`. Заголовочные файлы

Зачем нужны заголовочные файлы:

- ❑ для того чтобы структурировать программу, мы разбиваем текст на отдельные файлы, поэтому, естественно, появляются зависимости между этими файлами (см. разд. 1.2.1). Одним из способов разрешения этих зависимостей является отделение интерфейса (объявлений внешних понятий) (см. разд. 3.5) от реализации (от определений) и помещение интерфейсов в заголовочные файлы;
- ❑ для того чтобы воспользоваться средствами стандартной библиотеки, нужно в своем модуле предоставить компилятору их описания (объявления). Эти описания сгруппированы разработчиками стандартной библиотеки в соответствующих заголовочных файлах;
- ❑ ОС Windows позволяет программисту пользоваться системными возможностями (Win32API) посредством подключения системных заголовочных файлов;
- ❑ библиотека MFC предоставляет свои заголовочные файлы
- ❑ и т. д.

Заголовочные файлы предоставляют возможность программисту:

- ❑ реализовать механизм раздельной компиляции;
- ❑ обеспечить согласованность объявлений пользовательских объектов во всех файлах проекта.

В соответствии с потребностями программиста заголовочные файлы подразделяются на следующие группы (рис. 5.2):

- ❑ пользовательские — создает сам программист, описывая экспортируемые понятия;
- ❑ библиотечные — предоставляют программисту интерфейс для использования средств библиотек:
 - стандартной библиотеки C++;
 - Windows;
 - MFC;
 - прочих библиотек.

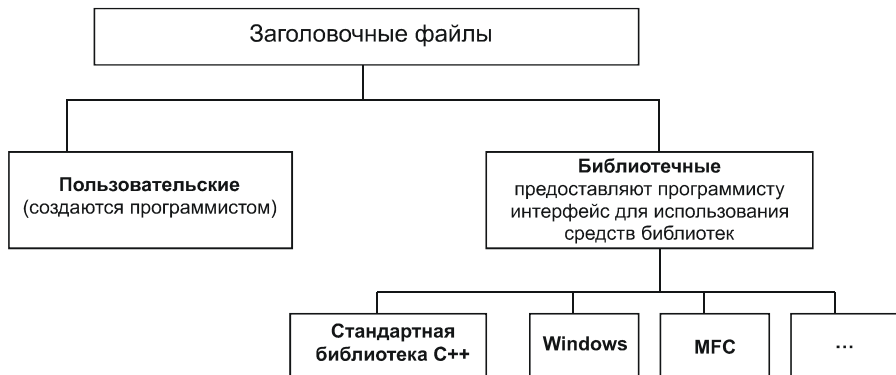


Рис. 5.2

5.6.1. Концепция разделения на интерфейс и реализацию. Механизм подключения заголовочных файлов

Всегда есть кто-то, кто предоставляет сервисы (сервер), и те, кто этими сервисами пользуются (клиенты). А каждый клиент, в свою очередь, может быть сервером для другого клиента (другого уровня) и т. д.

Согласно этой терминологии файл, который экспортирует данные или функции, является сервером, а файл, который их использует, является клиентом. При компиляции каждого файла-клиента компилятор должен знать свойства всех тех переменных или функций, которые определены в других файлах-серверах, и обращение к которым происходит в данном файле-клиенте. Обязанность программиста — предоставить компилятору объявления, т. е. описания импортируемых клиентом понятий:

- ❑ для переменных нужно описать тип и, возможно, некоторую дополнительную информацию, чтобы компилятор смог генерировать низкоуровневые инструкции при действиях с этой переменной (см. разд. 3.5.1);
- ❑ для функций нужно описать способ вызова, количество и типы параметров (см. разд. 8.2), чтобы компилятор смог сгенерировать вызов.

Программисту необходимо описывать все внешние зависимости в исходном тексте программы (иначе компилятор выдаст ошибку), но это можно делать по-разному.

Способ первый.

Если каждый клиент описывает компилятору все используемые им внешние понятия, то текст программы выглядит примерно так, как показано

на рис. 5.3, что является неэффективным описанием внешних зависимостей, поскольку:

- ❑ программист в каждом файле-клиенте пишет одни и те же объявления (**extern int x, y, z;**), непроизводительно расходуя свои силы и увеличивая размер текстов исходных файлов;
- ❑ при этом у него всегда остается возможность сделать при переписывании какую-нибудь ошибку (например, **extern int X, Y, Z;**), а т. к. все объявления одного и того же объекта в разных единицах компиляции должны быть согласованы (одинаковы), то компоновщик выдаст ошибку;
- ❑ если программист, разрабатывающий сервер, по каким-нибудь причинам решит модифицировать экспортируемые понятия, то в файле-клиенте придется изменять все объявления.

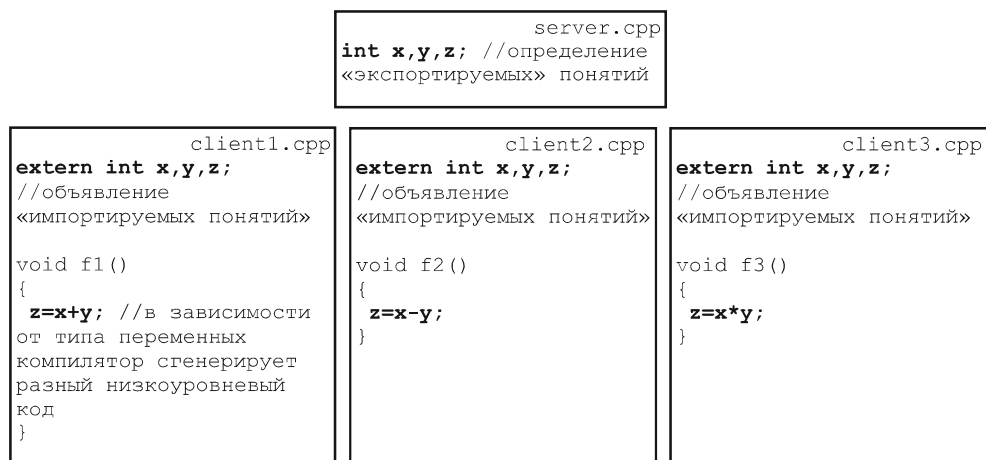


Рис. 5.3

При таком способе описания внешних зависимостей каждый клиент самостоятельно должен заботиться о том, чтобы все внешние для данного файла понятия были описаны до их первого использования клиентом (рис. 5.4).

Способ второй.

Для эффективного описания внешних зависимостей сервер сам предоставляет свой код в виде двух отдельных частей (рис. 5.5):

- ❑ *файл интерфейса* или заголовочный файл (обычно с расширением .h), в который программист помещает описания всех экспортируемых сервером понятий (объявления функций и переменных) — это интерфейс, посредством которого любой файл-клиент может пользоваться экспортируемыми сервером понятиями;

Неэффективное описание внешних зависимостей

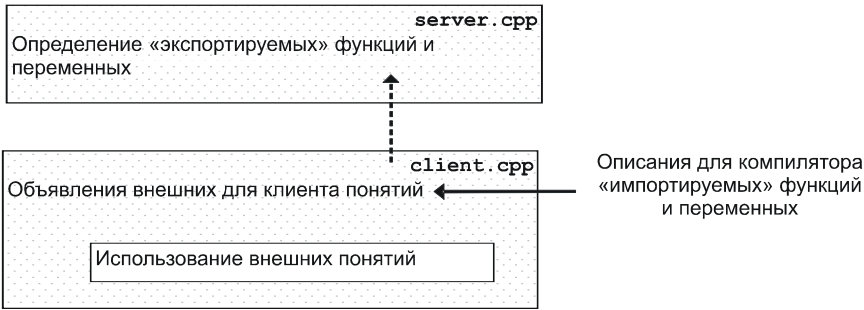


Рис. 5.4

Концепция разделения на интерфейс и реализацию

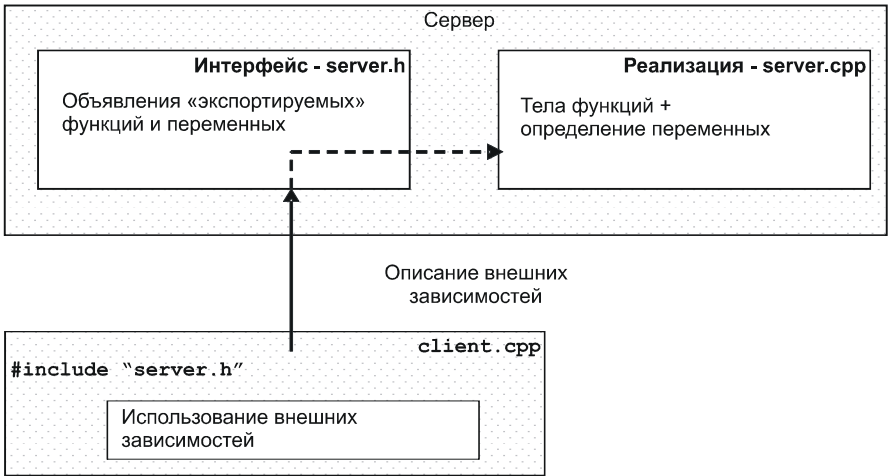


Рис. 5.5

❑ *файл реализации* (обычно файл с расширением .c или .cpp), в котором находятся определения экспортируемых сервером функций и переменных. А начинка такого файла — это подробности реализации, о которых клиенту знать вовсе не обязательно.

Для того чтобы любой клиент мог воспользоваться содержимым заголовочного файла, ему достаточно подключить этот файл посредством директивы препроцессора `#include`.

Встретив директиву:

```
#include "включаемый_файл"
```

препроцессор заменяет строку, содержащую **#include**, на содержимое указанного файла. Таким образом, оказывается, что все клиенты пользуются одним и тем же интерфейсом (одними и теми же объявлениями), поэтому вероятность ошибки у программиста уменьшается. Приведенный пример (см. рис. 5.3) нужно переписать, используя механизм **#include**. Результат представлен на рис. 5.6.

Эффективное описание внешних зависимостей

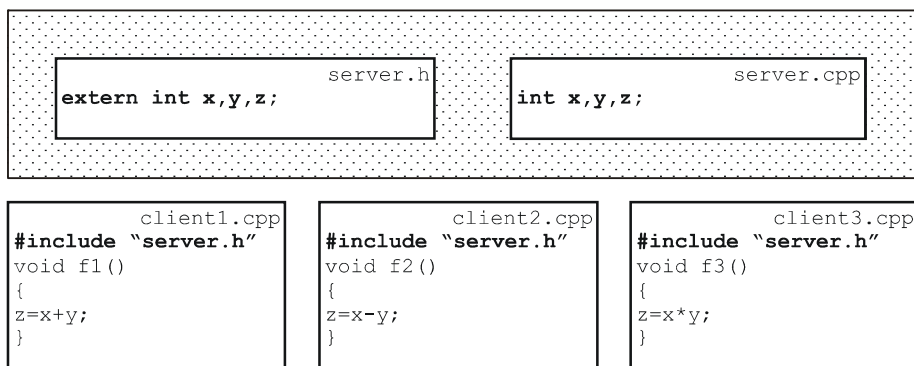


Рис. 5.6

РЕЗЮМЕ

В языке C/C++ механизм **#include** является эффективным средством манипулирования текстом, позволяющим собрать фрагменты исходного кода в одну единицу компиляции. Первейшая задача начинающего программиста — научиться пользоваться этим средством грамотно!

5.6.2. Формы директивы **#include**

Существуют следующие формы директивы:

- **#include** <спецификация_файла>
- **#include** "спецификация_файла"

При использовании любого варианта препроцессор заменяет директиву **#include** текстом указанного файла.

Отличие двух форм:

- первая форма директивы предполагает, что подключается стандартный файл. Поиск файла ведется последовательно в каждом из каталогов, перечисленных в опциях проекта в порядке их определения. В основном,

таким образом подключаются заголовочные файлы стандартной библиотеки, возможно — других библиотек;

- второй вариант специфицирует определенный пользователем включаемый файл. Если путь не указан, то сначала препроцессор ищет файл в текущем каталоге (в том каталоге, в котором находится исходный файл, содержащий **#include**). Если в текущем каталоге данного файла нет, то поиск продолжается дальше в перечисленных в опциях проекта каталогах (так же, как и в первом варианте).

Примеры.

```
#include <iostream> //указание препроцессору искать файл iostream
                     в стандартном включаемом каталоге
```

```
#include "my.h" //указание препроцессору искать файл my.h сначала
                в текущем каталоге, а потом в каталогах проекта
```

```
#include "c:\myhdr\my.h" //указание препроцессору искать файл my.h
                          только в c:\myhdr и нигде больше
```

ЗАМЕЧАНИЕ

Пробелы внутри `< >` или `" "` являются значащими.

Поэтому запись **#include** `<_iostream>` вызовет ошибку — препроцессор не найдет файл с таким именем.

5.6.3. Вложенные включения заголовочных файлов (стратегии включения)

Использовать механизм **#include** тоже можно по-разному. Согласно правилам структурного программирования разработчик группирует логически связанные между собой понятия (функции и данные) в отдельных сpp-файлах. Как быть с интерфейсами?

Для удобства интерфейсы можно группировать (все, что используется большинством файлов-клиентов, можно объединить с помощью вложенных **#include**). Например, пусть большинство (100) файлов проекта используют сервисы ОС Windows (заголовочный файл `<windows.h>`), макросы стандартной библиотеки, определенные в `<tchar.h>` и описания используемых всеми клиентами пользовательских переменных и функций, объявленных в `"my.h"`. Без группировки заголовочных файлов картина выглядела бы так, как представлено на рис. 5.7.

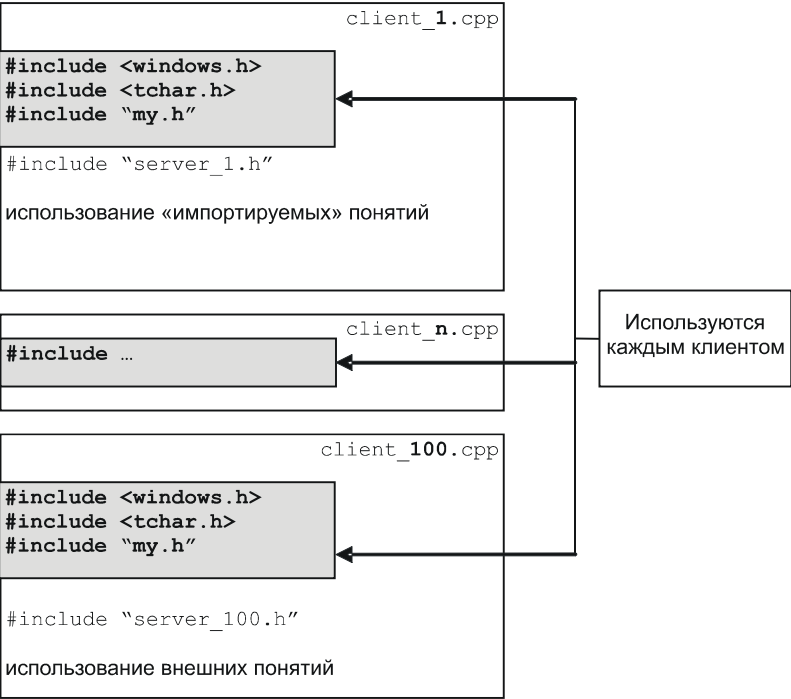


Рис. 5.7



Рис. 5.8

Это означает, что в каждом файле-клиенте (т. е. 100 раз) программист должен написать последовательность из трех совершенно одинаковых директив **#include**. Для того чтобы избежать дублирования текста и избавить программиста от интеллектуального занятия "copy-paste", можно ввести вспомогательный заголовочный файл, в котором сгруппировать подключение таких заголовочных файлов проекта, используемых большинством клиентов (рис. 5.8). Тогда для получения доступа к общеиспользуемым внешним понятиям каждому клиенту достаточно подключить этот единственный вспомогательный файл, в котором сгруппированы все общие интерфейсы. А для получения специфических для данного конкретного клиента интерфейсов, каждый клиент может подключить дополнительные (специфические для данного клиента) заголовочные файлы.

5.6.4. Предкомпиляция заголовочных файлов

В предыдущем примере для удобства мы просто сгруппировали используемые всеми клиентами заголовочные файлы, чем слегка облегчили жизнь программисту, но не компилятору. При компиляции любого клиента препроцессор сначала заменит директиву **#include** "general.h" на содержимое файла general.h, а затем, в свою очередь, каждую директиву **#include** из файла general.h на содержимое соответствующего файла.

Содержимое каждого заголовочного файла — это текст на языке C/C++, который компилятор должен синтаксически разобрать, следовательно в нашем примере он будет сто раз проделывать одну и ту же работу!

Неэффективно заставлять компилятор столько раз анализировать один и тот же текст (хотя, справедливости ради, стоит отметить, что заголовочные файлы, как правило, содержат только объявления, а на анализ объявлений компилятор тратит времени меньше, чем на создание кода).

Чтобы минимизировать затраты компилятора, большинство современных реализаций C++ обеспечивают возможность предварительной обработки компилятором заголовочного файла и сохранения результатов этой обработки в промежуточном файле для дальнейшего использования. Такой механизм называется *предкомпиляцией*. Компилятор один раз осуществляет синтаксический разбор заголовочного файла, предназначенного для предкомпиляции, и сохраняет результат в некотором внутреннем формате (формат может зависеть от конкретного компилятора) в файле с расширением .pch (это расширение принято в VC — от PreCompiled Headers, но расширение файла может зависеть также и от конкретного компилятора). Эту, уже готовую к использованию информацию, компилятор может учитывать при компиляции любого клиента, требующего тот же интерфейс (т. е. не анализировать заново).

Предкомпиляция бывает полезной во время цикла разработки для уменьшения общего времени получения загрузочного модуля, особенно если:

- ❑ используется большое количество кода, который не изменяется (системные заголовочные файлы: `<windows.h>` или библиотечные — `<tchar.h>`);
- ❑ во многих файлах-клиентах подключаются одни и те же пользовательские интерфейсы ("`my.h`"), которые изменяются редко.

Использование предкомпиляции заголовочных файлов

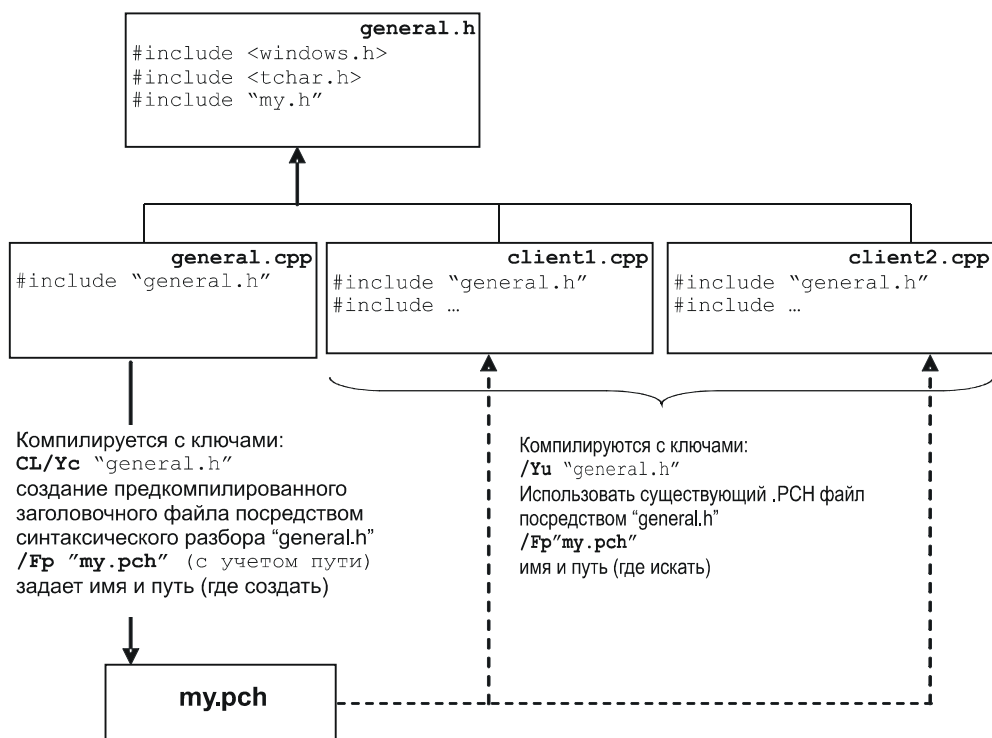


Рис. 5.9

Как использовать предкомпиляцию? Посредством разных опций в командной строке компилятору исходные файлы можно компилировать по-разному. Обычно поступают следующим образом (рис. 5.9): кроме модулей-клиентов в проекте создается вспомогательный файл (пусть в нашем примере он называется `general.cpp`), который содержит одну-единственную строчку: **#include "general.h"**. Этот файл предназначен для единственной цели — создать

предкомпилированный файл, содержащий результаты синтаксического разбора файла `general.h`. Он будет компилироваться с ключом компилятору — `\Yc` (создать файл предкомпиляции) посредством синтаксического разбора файла `general.h`. А все остальные исходные файлы проекта (клиенты) компилируются с ключом `\Yu`— (использовать файл предкомпиляции). В этом случае первая компиляция занимает времени много, т. к. создается `rch`-файл, зато все последующие уже используют этот полуфабрикат, поэтому суммарное время получения исполняемого файла уменьшается.

ЗАМЕЧАНИЕ

Хотя вы можете использовать только один предкомпилированный файл (с расширением `.pch`) для каждого исходного `cpp`-файла, ничто не мешает использовать несколько `rch`-файлов в одном проекте.

Предкомпиляция — это только один из способов ускорения получения исполняемого файла. Для большинства реализаций существуют несколько способов ускорения процесса построения приложения, например для VC:

- ❑ `precompiled headers` (предкомпиляция) — позволяет компилятору не повторять синтаксический разбор общеиспользуемых заголовочных файлов при компиляции каждого `cpp`-файла. Обычно в такой предкомпилированный файл включаются системные и библиотечные заголовочные файлы (они большие!), а включать свои (особенно на стадии разработки) смысла не имеет, т. к. одно-единственное изменение в таком файле повлечет перестройку (`rebuild`) всего проекта. С другой стороны, если в вашем проекте уже имеются отлаженные, не меняющиеся заголовочные файлы, то включение их в предкомпилированный файл ускорит время сборки;
- ❑ `incremental linking` — позволяет делать сравнительно небольшие изменения в программе (ставить "заплаты"), не перестраивая всю программу. То есть компилятор предусматривает для каждой функции небольшой резерв для расширения. Если изменения оказываются достаточно большими, то осуществляется полная перекомпоновка программы;
- ❑ `minimal rebuild` — анализирует изменения, которые вы сделали в `h`-файлах (в основном это касается объявлений классов), и физически перекомпилируются только те `cpp`-файлы, которые реально зависят от изменений (без этой опции `cpp`-файл будет перекомпилирован, даже если вы измените лишь комментарий во включаемом заголовочном файле).

Такие способы ускорения получения исполняемого модуля можно задать посредством опций командной строки компилятору и компоновщику. В интегрированных средах это можно сделать в опциях проекта.

5.6.5. Заголовочные файлы стандартной библиотеки

Практически с момента появления языка Си, в понятие реализации (и в комплект поставки, соответственно) входит, помимо компилятора и компоновщика, стандартная библиотека или (C Run-Time Library) (сокращенно CRT). Библиотека включает в себя часто используемые понятия и действия, к которым программисты на C/C++ уже настолько привыкли, что отождествляют их с языком программирования. Стандартность библиотеки заключается в том, что ее составляющие (имена компонентов, их назначение и принципы применения) должны соответствовать принятым стандартам (т. е. интерфейс использования функций и других средств стандартной библиотеки должен быть одинаков, а реализация (внутренняя начинка) может зависеть от конкретного производителя).

Стандартная библиотека C++ создавалась для предоставления программисту:

- ☐ часто используемых функций (`printf()`, `strcpy()`);
- ☐ макроопределений (`va_arg`, `min`);
- ☐ полезных псевдонимов типов (`size_t`);
- ☐ часто используемых констант (`NULL`, `EOF`);
- ☐ глобальных переменных (`errno`);
- ☐ стандартных типов данных (классов);
- ☐ шаблонов классов, на основе которых программист может строить свои сложные структуры данных и шаблоны функций для часто используемых задач, таких как поиск, сортировка и т. д.

ЗАМЕЧАНИЕ

Все перечисленное готово к употреблению, и любой компилятор C++, отвечающий принятым стандартам, должен все эти понятия поддерживать.

Как использовать возможности стандартной библиотеки

Стандартная библиотека предоставляет программисту свои возможности посредством набора стандартных заголовочных файлов. Сервисы, предоставляемые стандартной библиотекой C++, можно условно разделить на три категории (рис. 5.10).

- ☐ STL (*Standard Template Library*) — библиотека шаблонов (в данной книге не рассматривается):
 - контейнеры (заготовки для сложных типов данных, таких как векторы, списки, деревья и т. д.);

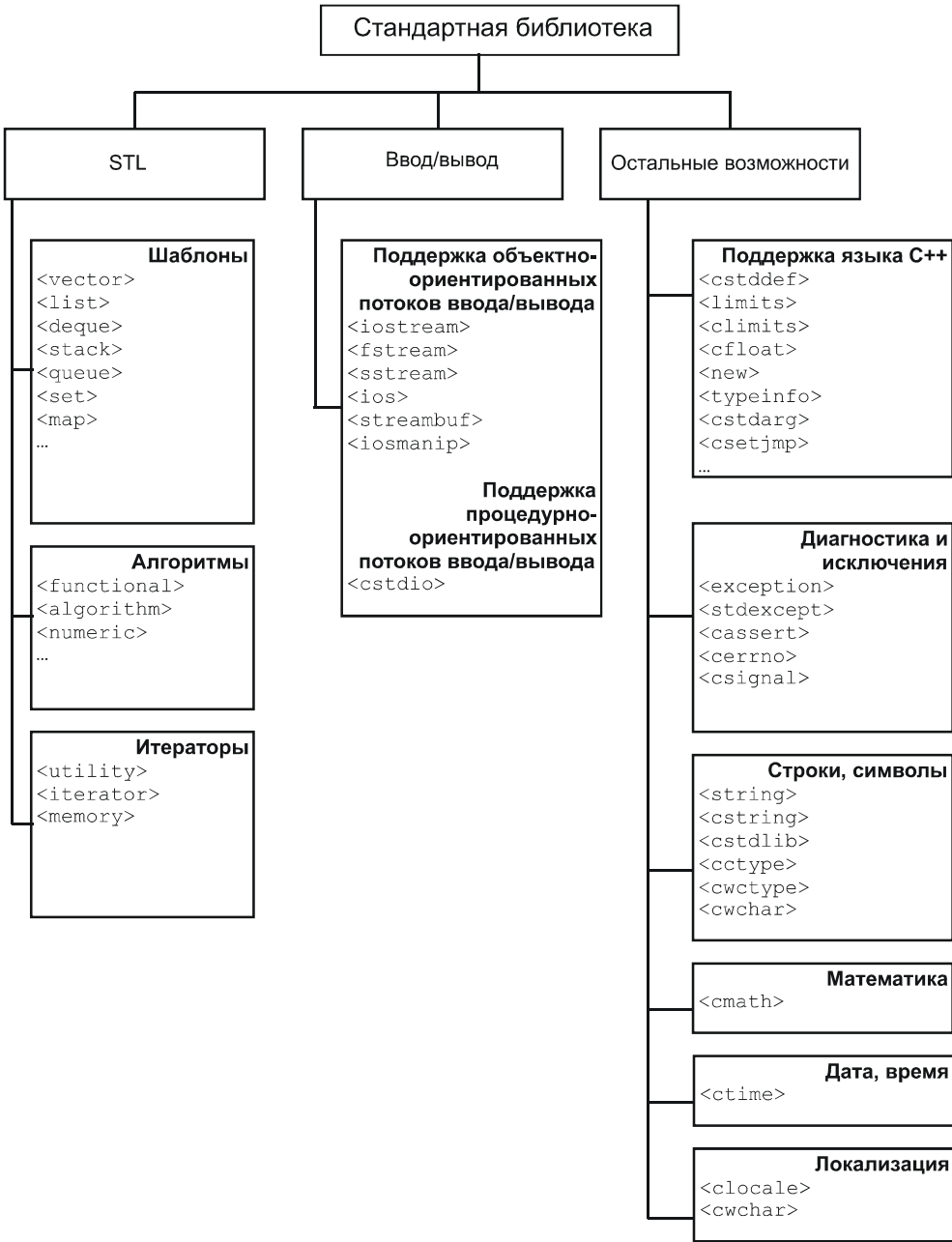


Рис. 5.10

- обобщенные алгоритмы (алгоритмы для работы с контейнерами любого типа);
 - итераторы (средства для навигации по любым контейнерам).
- ❑ Ввод/вывод — средства форматирования и управления операциями ввода/вывода (вывод на экран, ввод с клавиатуры, работа с файлами и т. д.).
- ❑ Остальные возможности: математические функции (`sqrt`, `pow`, `sin`, `atof`, функции Бесселя и т. д.), функции для работы со строками (поскольку таких встроенных типов данных, как строки, в языке C/C++ нет), средства для локализации программ, манипулирования датами, временем и т. д.

Новые заголовочные файлы стандартной библиотеки

По мере развития языка C++, стандартная библиотека тоже развивалась, расширялась и совершенствовалась (рис. 5.11). Сначала была разработана библиотека Си, потом появились классы (библиотека пополнилась полезными классами C++), потом появились шаблоны (в библиотеку добавили контейнеры, обобщенные алгоритмы и итераторы). А потом был утвержден стандарт ISO/IEC 14882 (Standard for the C++ Programming Language...), и как использование, так и состав стандартной библиотеки были унифицированы.

Эволюция стандартной библиотеки

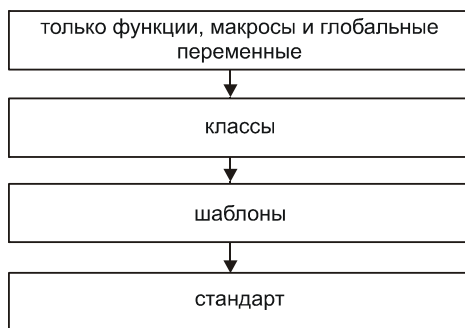


Рис. 5.11

Согласно стандарту, заголовочные файлы были переименованы (поэтому рекомендуется использовать имена заголовочных файлов в новом стиле). Правила, по которым были сформированы названия новых заголовочных файлов, достаточно просты:

- ❑ в заголовочных файлах, которые появились только в C++, просто опущено расширение (`<iostream.h>` переименовали в `<iostream>`);

СУЩЕСТВЕННО

Содержимое двух таких файлов может быть принципиально разным!

- в заголовочных файлах, которые пришли из языка Си, опущено расширение и перед названием добавлен символ `s` (`<math.h>` переименовали в `<smath>`). Такие файлы в основном являются просто обертками для соответствующих нестандартизованных заголовочных файлов с прежним содержанием.

РЕКОМЕНДАЦИЯ

Так как изменились не только названия, но в большинстве случаев и содержимое заголовочных файлов, то старые файлы использовать не рекомендуется, а тем более смешивать новые и старые (хотя в большинстве реализаций параллельно с новыми библиотечными файлами продолжают существовать старые).

ЗАМЕЧАНИЕ

Забавно, что некоторые сервисы пока существуют одновременно в трех реализациях. Например, строковые функции поддерживаются тремя заголовочными файлами: `<string.h>` — процедурно-ориентированные строковые функции, `<cstring>` — новая обертка для `string.h` и `<string>` — классы (вернее, шаблоны классов) поддержки работы со строками.

СУЩЕСТВЕННО

Новые сервисы стандартной библиотеки заключены в пространство имен `std`.

Например:

```
//файл <iostream>
namespace std{
    объявления
}
```

Специфика Microsoft: для заголовочных файлов, пришедших из Си-библиотеки, сделано примерно следующее: согласно стандарту все понятия заключены в пространство имен `std`, но тут же посредством директивы **using** разрешается их видимость, т. е. к ним разрешается обращаться без префикса:

```
//файл <smath>
namespace std{
    #include <math.h>
}
using namespace std;
```


Стандартная библиотека C++ в соответствии с новым стилем вместо имен заголовочных файлов использует стандартные идентификаторы, посредством которых препроцессор находит требуемое содержимое. Новые заголовки C++ являются абстракциями, гарантирующими объявления соответствующих сервисов стандартной библиотеки. Для преемственности новые идентификаторы по-прежнему включаются в исходный файл директивой `#include`. Старый заголовочный файл всегда должен находиться на жестком диске (в него можно заглянуть, как и в любой текстовый файл). Новые заголовки в перспективе могут стать виртуальными. При этом предполагается, что препроцессор по идентификатору всегда сможет найти содержимое заголовочного файла (будет ли этот файл на диске или препроцессор загрузит требуемую информацию каким-либо другим способом).

5.6.6. Защита от повторных включений заголовочных файлов

Иногда (обычно неявно) программист создает такую ситуацию, когда при компиляции исходного файла препроцессор встречает несколько раз включение одного и того же заголовочного файла. В некоторых случаях такое повторное подключение приводит к ошибкам компилятора (рис. 5.12).

Ошибка при повторном подключении заголовочного файла

```
my.cpp
#include "1.h"
#include "2.h"

Ошибка компилятора:
повторное объявление
enum
```

```
1.h
#include "2.h"
```

```
2.h
enum COLOR{RED, GREEN, BLUE};
...
```

Рис. 5.12

В этой ситуации при обработке препроцессором исходного файла `my.cpp` текст интерфейсного файла `2.h` будет включен дважды, и компилятор выдаст ошибку: повторное определение `enum COLOR`.

Для того чтобы избежать повторного подключения, удобно использовать директивы условной трансляции препроцессора так, как это показано на рис. 5.13.

При первом подключении файла `2.h` имя `HEADER_2` для препроцессора не определено, поэтому весь текст файла `2.h` препроцессор подставляет вместо строки `#include "2.h"`. А на момент вторичного подключения того же заго-

ловочного файла имя `HEADER_2` уже является для препроцессора определенным, поэтому весь текст файла `2.h` до директивы `#endif` препроцессор игнорирует.

Защита от вложенных подключений заголовочных файлов

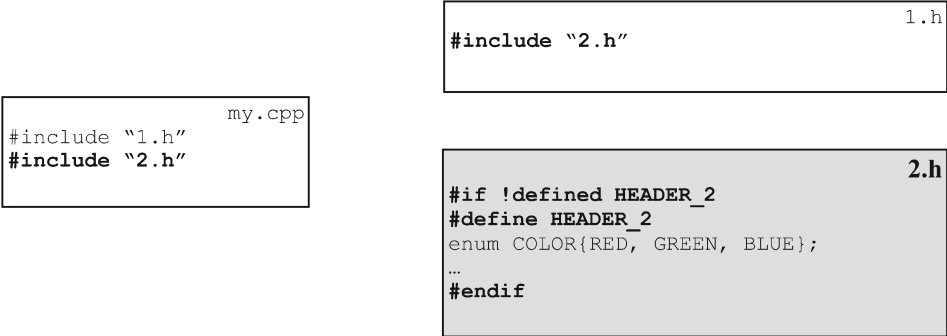


Рис. 5.13

5.6.7. Что может быть в заголовочных файлах и чего там быть не должно

За небольшим количеством исключений, в заголовочных файлах могут быть только объявления (табл. 5.5), они не должны содержать определений (табл. 5.6). Это скорее рекомендация, а не требование языка. Она просто отражает разумный способ использования заголовочных файлов (Б. Страуструп). Компилятор позволит вам не следовать этим рекомендациям, но тогда от использования механизма подключения заголовочных файлов вы получите неприятностей больше, чем выгод (см. следующий раздел).

Таблица 5.5. Что может быть в заголовочном файле

Что может содержать заголовочный файл	Пример
Вложенные директивы <code>#include</code>	<code>#include "general.h"</code>
Объявления функций, определенных в другом модуле	<code>[extern] void MyFunc();</code>
Определение встроенных (inline) функций	<code>inline void MyInlineFunc() { тело_функции }</code>
Объявления внешних переменных, определенных в другом модуле	<code>extern int Global;</code>

Таблица 5.5 (окончание)

Что может содержать заголовочный файл	Пример
Определения простых констант	<code>const double pi = 3.141593;</code>
Перечисления	<code>enum color{RED, GREEN, BLUE};</code>
Директивы условной трансляции	<code>#if !defined HEADER_2</code> ... <code>#endif</code>
Макроопределения	<code>#define VERSION 1</code>
Именованные пространства имен	<code>namespace Version{</code> <code>extern int currentVersion;</code> <code>...}</code>
Объявления структур, объединений, классов	<code>struct Point{int m_x,</code> <code>m_y};</code>
Предварительное неполное объявление	<code>class Point;</code>
Определения шаблонов	<code>template<class T> class</code> <code>MyTempl{...};</code>
Объявления шаблонов	<code>template<class T> class</code> <code>MyTempl;</code>

Таблица 5.6. Чего в заголовочном файле быть не должно

Что не следует помещать в заголовочный файл	Пример
Определения невстроенных функций	<code>void MyFunc(){ тело_функции }</code>
Определения данных	<code>int a;</code> <code>char Ar[] = {'A', 'B', 'C'};</code>
Неименованные пространства имен	<code>namespace{...}</code>

Почему не стоит помещать в заголовочные файлы определения и подключать файлы реализации (.cpp)

Заголовочные файлы могут содержать любой допустимый текст на языке C/C++, в частности — ничто не мешает программисту поместить в заголовочный файл определения переменных или функций (рис. 5.14). Так как `#include` — это просто текстовая подстановка, то в файлах 1.cpp и 2.cpp (в том тексте, который пойдет на компиляцию) будут определены две переменные, подлежащие внешней компоновке, с одним и тем же именем. При сборке компоновщик обнаружит двойное определение и выдаст ошибку.

Не помещайте определения в заголовочные файлы!

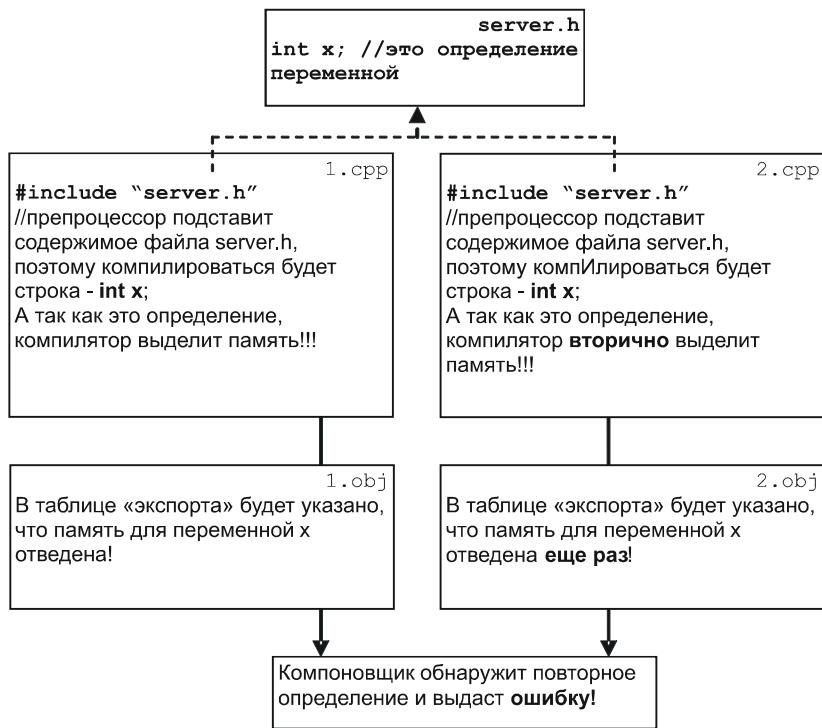


Рис. 5.14

Поэтому структурный подход к оформлению исходных текстов предполагает, что каждый сервер предоставляет свои сервисы в двух частях: интерфейсный файл, содержащий только объявления (описания свойств экспортируемых понятий), и файл реализации, содержащий определения (встречая определение переменной, компилятор выделяет память, встречая определение функции — генерирует код). Тогда для того чтобы воспользоваться внешними понятиями, клиенту достаточно подключить файл интерфейса, а определения будут существовать в единственном экземпляре.

По той же причине не рекомендуется подключать файлы реализации. Несмотря на то что директиву `#include` можно использовать для подключения файла с любым содержимым и любым расширением (это просто текстовая подстановка), подключение несколькими клиентами одного и того же файла реализации снова приведет к повторным определениям.

5.7. Директива **#pragma**

Синтаксис:

#pragma имя_директивы

Посредством директивы **#pragma** программист имеет возможность давать указания препроцессору, компилятору, а опосредованно — и линкеру.

Специфика:

- ❑ каждая реализация C/C++ может поддерживать какие-то особенности, уникальные для используемого процессора или ОС. Директива **#pragma** позволяет настраиваться на реализацию, не модифицируя текст программы;
- ❑ эти директивы зависят от конкретной реализации (от конкретного препроцессора). Если препроцессор встречает **#pragma**-директиву, которую он не распознает, то ничего не делает и генерирует не ошибку, а предупреждение;
- ❑ если некоторые **#pragma**-директивы дублируют опции командной строки компилятора или компоновщика, то первые являются по отношению ко вторым более приоритетными;
- ❑ в отличие от опций командной строки (которые применяются ко всему файлу в целом), **#pragma**-директивы позволяют применять действия к отдельным функциям и переменным.

Пример 1.

С помощью **#pragma check_stack** можно дать указание компилятору вставлять (или наоборот исключать) в генерируемый низкоуровневый код проверку (хватает ли места в стеке для размещения локальных переменных). Если памяти не хватает, то генерируется исключение.

```
#pragma check_stack(on)

void f()
{ // проверка
    ...
}

#pragma check_stack(off) //отключает проверку
#pragma check_stack() //восстанавливает параметры командной строки
```

Пример 2.

Посредством **#pragma optimize** можно предписать компилятору оптимизировать при генерации кода время, память....:

```
#pragma optimize ("t",on) //включает оптимизацию по времени
...
```

#pragma optimize ("",on) //восстанавливает параметры командной строки

Пример 3.

Посредством **#pragma once** можно избежать повторных подключений заголовочных файлов. Модифицируем пример из предыдущего раздела (рис. 5.15). VC предоставляет возможность посредством **#pragma once** сообщать препроцессору, что остаток текста до конца файла нужно игнорировать при повторных подключениях данного заголовочного файла.

Защита от повторных подключений заголовочных файлов

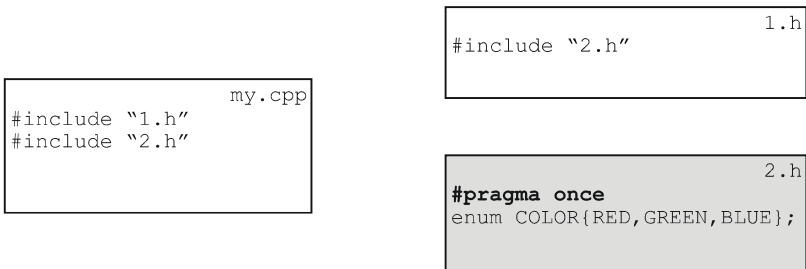


Рис. 5.15

5.8. Директива **#error**

Если препроцессор встречает директиву **#error**, он прекращает формировать текст для компилятора и в качестве диагностики выдает сообщение, предусмотренное программистом.

Пример.

```

#if !defined _WIN32
#error Valid only in Win32 applications
#endif
    
```



Глава 6

Указатели и массивы

В языке C/C++ существует взаимосвязь между указателями и массивами настолько сильная, что их следует рассматривать одновременно. При объяснении одного понятия неизбежны ссылки на другое. Поэтому, начиная тему с рассмотрения указателей, введем очень примитивное понятие массива. Массив можно рассматривать как один (сложный) программный объект. На самом деле массив — это совокупность элементов одного и того же типа (пронумерованных с нуля, последовательно расположенных в памяти), к которым можно обращаться по номеру (индексу).

6.1. Указатели

Указатель — это бумеранг. Если умеешь им пользоваться, возвращаешься с добычей (получаешь эффективный код), а если не умеешь...

При выполнении программы фрагменты программного кода и элементы данных располагаются в различных участках оперативной памяти, каждый из которых имеет свой (уникальный) адрес. Обращение к любому элементу C/C++ программы (имеется в виду переменная базового типа, функция или объект более сложного типа) можно осуществить, зная его адрес.

Такой способ, когда для обращения к объекту программист явно использует адрес объекта, называется *косвенным* обращением. Для представления адреса в C/C++ предназначен тип данных, называемый указателем.

Указатель — это переменная, содержащая адрес другого элемента C/C++ программы. Если переменная содержит адрес некоторого другого элемента, то говорят, что переменная указывает на этот элемент.

Указатель может указывать на любой программный объект (рис. 6.1):

- на обычную переменную;
- на более сложные типы данных:
 - массив (см. разд. 6.3);
 - строку (см. разд. 6.1.2);
 - структуру (см. разд. 9.11);
 - другой указатель (см. разд. 6.1.7);
 - функцию (см. разд. 8.11).

Указатели подразделяются на две основные категории:

- указатели на данные;
- указатели на функции.

Хотя оба типа указателей представляют собой адрес, они имеют различные свойства, назначение и правила работы с ними. В данной главе рассматриваются только указатели на элементы данных. Указатели на функции будут рассмотрены в гл. 8.

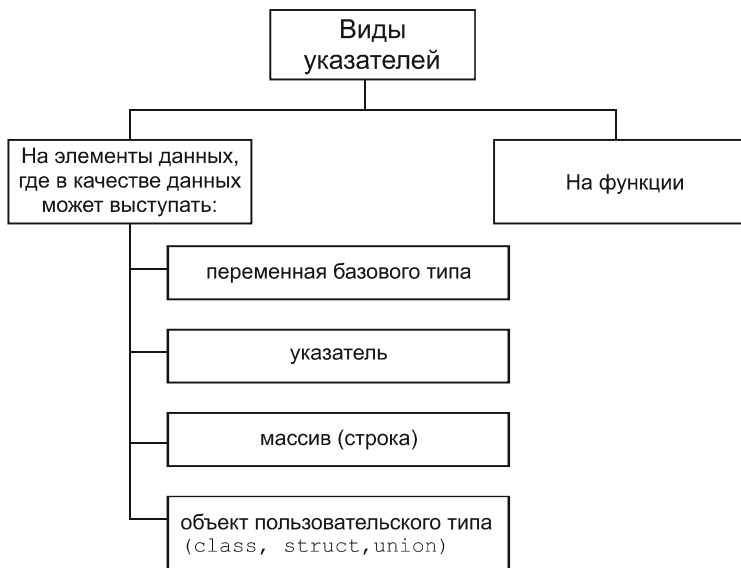


Рис. 6.1

6.1.1. Объявление и определение переменной-указателя

Указатель всегда содержит адрес, а каким образом компилятор должен интерпретировать этот адрес, сообщает ему программист, задавая тип указываемого объекта!

Например:

```
int* p1;           //объявление, совмещенное с определением, где:
                    p1 - имя переменной-указателя,
                    int - тип указываемого объекта,
                    символ * говорит о том, что p1 — это указатель.
                    В этом случае компилятор должен зарезервировать
                    память для p1 (4 байта на 32-разрядной платформе)

double* p2;        //p2 - имя переменной-указателя,
                    double* - тип переменной.
                    Зарезервирована память для указателя p2 - 4 байта

extern char* p3;    //а это только объявление внешней переменной-
                    указателя, которая определена в другом модуле.
```

Сравним, каким образом компилятор выделяет память под объекты и под указатели на объекты (рис. 6.2). Количество байтов, выделяемых компилятором под объект, зависит от типа объекта. Количество байтов, выделяемых для хранения адреса, не зависит от типа объекта, а определяется только аппаратными особенностями используемого процессора — разрядностью регистров (фиксирована для каждой конкретной архитектуры), предназначенных для хранения адресов.

ЗАМЕЧАНИЕ

Существует лишь очень небольшое количество архитектур, в которых может использоваться несколько видов указателей, различающихся размерами. Пример такой архитектуры: семейство Intel MCS-52/251 — в ней указатели могут иметь размер 1, либо 2, либо 3 байта.

Специфические особенности указателей:

- при объявлении указателя компилятору все равно, где располагается символ *. То есть все приведенные ниже способы объявления указателя синтаксически корректны. Каждое из них означает, что объявляется указатель (переменная с именем pn), которая может содержать адрес переменной типа `int`;

```
int* pn;
int *pn; //или так
int * pn; //и так тоже можно
int*pn; //и даже так
```

Выделение памяти под переменные и под указатели

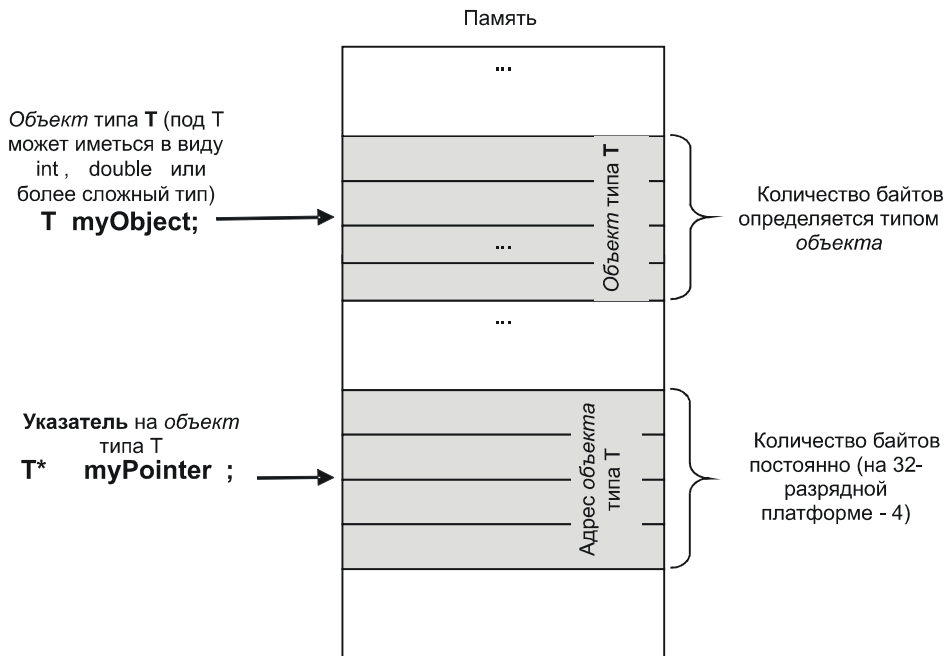


Рис. 6.2

- ❑ указатель может содержать адрес как одиночной переменной некоторого типа **T**, так и адрес одного из элементов массива (см. разд. 6.2). Об этом знает только программист, у компилятора такой информации нет;
- ❑ при определении указателя вида **T* p;** (без инициализации) выделяется память для переменной типа указатель, но сам указатель пока ни на что не указывает, поскольку его значение (т. е. адрес) еще не сформировано. Корректный адрес должен быть присвоен указателю *до его первого использования*. Это рекомендуется (но не обязательно) делать при определении указателя;
- ❑ в зависимости от контекста определения компилятор может выделить память под указатель в разных областях, независимо от того, где находится сам указываемый объект (о выделении памяти при объявлении переменных см. разд. 3.7). Память под указатель может быть выделена:
 - в стеке (локальный указатель);
 - в статической области данных (глобальный или статический);
 - в "куче" (динамический указатель);

Например:

```
int* p; //глобальный
void f()
{
    int* p1; //локальный
    static int* p2; //статический
}
```

❑ после объявления указателя на один тип попытка использования этого указателя для ссылки на другой тип (без явного преобразования типа) приведет к ошибке при компиляции (см. разд. 6.1.9):

```
int* p;
double* p1=p; //ошибка: компилятор не может преобразовать int*
                к типу double*
```

При объявлении указателей можно использовать объявление списком, но начинающий программист легко может допустить следующую ошибку:

```
//программист хочет объявить три указателя, вот так:
int* p1,p2,p3;    //а компилятор считает, что объявлен один указатель
                  p1 и две переменные типа int – p2, p3
int* p4, *p5, *p6 //а теперь компилятор считает, что объявлены три
                  указателя
```



Подумайте, в чем заключается разница приведенных примеров (табл. 6.1). Что объявлено?

Таблица 6.1. Задания на интерпретацию объявлений

Номер	Задание
1	<code>typedef char* PCHAR;</code> <code>PCHAR p1,p2;</code>
2	<code>define PCHAR char*</code> <code>PCHAR p1,p2;</code>

6.1.2. Инициализация указателя и оператор получения адреса объекта — &

Если программист не проинициализировал переменную-указатель при определении, то компилятор будет инициализировать указатели по общим правилам (см. разд. 3.9.2). Глобальные указатели, указатели, заключенные в про-

странства имен, и указатели, объявленные с ключевым словом **static**, *неявно* (по умолчанию) инициализируются нулем. Локальные и динамические указатели неявно компилятором не инициализируются, т. е. значение указателя после объявления не определено!

Для того чтобы воспользоваться указателем, программист должен сформировать в такой переменной адрес объекта. Это можно (и предпочтительнее) сделать при определении указателя, *явно* проинициализировав переменную посредством оператора получения адреса объекта &:

```
int n=1;           //переменная типа int
int* pn=&n;        //определение указателя на объект типа int и инициализация
                    //указателя адресом переменной n с помощью унарного
                    //оператора получения адреса — &
```

Действия со строковыми литералами в языке C/C++ всегда имеют особенности. В частности, рассмотрим специфику инициализации указателя адресом строкового литерала:

```
char* pStr = "Строка"; // компилятор отводит память для хранения
                        //строкового литерала (массива символов), а указателю
                        //присваивается адрес начала строки (адрес первого
                        //символа)
                        //или эквивалентное выражение:
char* pStr = &"Строка"; //синтаксически оба варианта корректны
                        //и смысл их одинаков
```

ЗАМЕЧАНИЕ 1

Оператор & можно применять только к объектам, с которыми компилятор ассоциирует выделенную память, поэтому следующие примеры вызовут сообщение компилятора об ошибке:

```
//int* p = &0xff00; //ошибка: нельзя определить адрес константы
int value=2;
//int* pV = &(value*3); //ошибка: нельзя определить адрес
//результата, получаемого при вычислении арифметического
//выражения справа от знака равенства
```

ЗАМЕЧАНИЕ 2

Нельзя присвоить указателю абсолютное значение без явного приведения типа (см. разд. 6.1.9), исключение составляет нулевое значение:

```
//int* p = 0x100000; //ошибка: компилятор не может преобразовать
                    //int к типу int*
int* p = reinterpret_cast<int*>(0x100000); //корректно
int* p1 = 0; //корректно
```



Выполните задания, приведенные в табл. 6.2.

Таблица 6.2. Задания на использование оператора получения адреса

Номер	Задание	Подсказка
1	Подумайте, каким образом будет проинициализирован указатель? <code>int* p = &((x>y) ? x : y);</code>	См. разд. 2.4.5
2	Являются ли корректными оба выражения? <code>int* p1 = &(++x);</code> <code>int* p2 = &(x++);</code>	См. разд. 2.4.1

6.1.3. Получение значения объекта посредством указателя: оператор разыменования — *

Сам по себе адрес объекта обычно особой ценности не представляет. Он служит лишь средством для доступа к объекту, на который указывает, и во многих случаях использование указателя может повысить эффективность вычислений.

Например:

```
int* p;    //объявление указателя
           //формирование значения p
int n=*p;  //унарный оператор * интерпретирует операнд p как
           адрес и обращается по этому адресу, чтобы извлечь содержимое
```

ПРАВИЛО

Сравните объявление указателя с операцией получения значения по указателю. Это поможет вам сформулировать простое мнемоническое правило, которое будет полезно при использовании любых сложных описаний: *если объявлен указатель p на объект типа T, то выражение *p эквивалентно объекту типа T.*

Пусть даны:

```
x=1;
int* p=&x;
```



Попробуйте при одних и тех же исходных данных описать последовательность действий компилятора и получите значение переменной p в примерах, приведенных в табл. 6.3.

Таблица 6.3. Задания на использование оператора разыменования

Номер	Задание
1	<code>n = *(p++);</code>
2	<code>n = (*p)++;</code>
3	<code>n = ++(*p);</code>
4	<code>n = *(++p);</code>

ЗАМЕЧАНИЕ

Выражение `int n = *(reinterpret_cast<int*> (0x42bb18));` будет верно только в том случае, если константа представляет собой корректный адрес. Такой прием не имеет особого смысла, если вы разрабатываете Windows-приложение, которое выполняется в защищенном режиме (т. е. ошибки при компиляции не будет в любом случае, а ошибка периода выполнения вполне вероятна!). Однако именно так порой приходится поступать при работе с периферийными устройствами, если в вычислительной системе реализован *ввод/вывод, отображаемый на память* (см. разд. 6.1.9 и разд. П2.1 приложения).

Как для переменной-указателя, так и для указываемого значения компилятор выделяет память. Как определить объем зарезервированной памяти? Оператор `sizeof` можно использовать для определения размера любого объекта, в частности с помощью этого оператора можно определить размер указателя:

```
double d = 1.1;
double* pd = &d;
size_t n1 = sizeof(pd); //n1=4 на 32-разрядной платформе
size_t n2 = sizeof(char*); //такой же результат
```



Подумайте, чему будут равны `n3` и `n4`?

```
size_t n3 = sizeof(*pd);
char* pc = "abc";
size_t n4 = sizeof(*pc);
```

6.1.4. Арифметика указателей

Важными особенностями арифметических операций с указателями являются:

- ☐ физическое изменение значения адреса, который хранится в указателе, зависит от типа (т. е. от размера) указываемого объекта;
- ☐ при выполнении действий с указателем компилятор всегда интерпретирует то, на что указывает указатель `T*`, как совокупность (массив) элементов типа `T`. А на что действительно указывает указатель (одиночный объект или массив объектов), знает только программист.

Рассмотрим, что можно делать с указателями:

```
int n = 1;  
int* pn1=&n; //определен и проинициализирован указатель pn1  
int* pn2; //определен pn2
```

Пример 1.

Адрес можно копировать из одного указателя в другой того же типа (рис. 6.3):

```
pn2=pn1; //компилятор копирует значение адреса из переменной pn1  
         в переменную pn2, т. е. теперь оба указателя содержат  
         адрес переменной n  
//double* pd = pn1; //ошибка: указатели разного типа
```

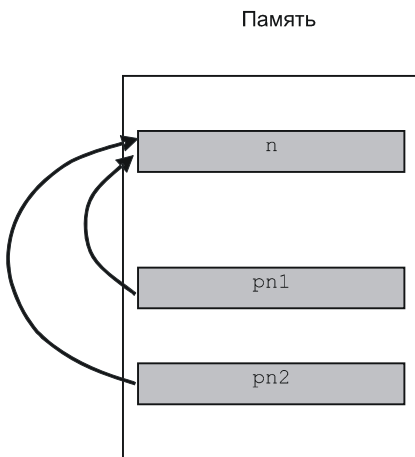


Рис. 6.3

Пример 2.

К указателю можно прибавить (или из него вычесть) значение, но только целое:

```
pn1++; //смысл такой операции заключается в том, что указатель  
       должен переместиться в памяти на следующий объект того же  
       типа, например, на следующий элемент массива, поэтому  
       компилятор прибавляет к адресу, хранящемуся в указателе,  
       размер объекта, в нашем случае sizeof(int)=4  
int n = 5;  
pn2=pn1+n; //а здесь компилятор должен вычислить адрес элемента,  
           отстоящего на n элементов от pn1. То есть значение,
```

которое будет присвоено `pn2`, вычисляется как сумма значения адреса, хранящегося в `pn1`, плюс `n*sizeof(типа)`, в нашем случае плюс `5*sizeof(int)=20`

```
//pn2=pn1+1.1; //ошибка: прибавлять к указателю  
                  можно только целое!
```

Пример 3.

Указатели одного типа можно вычитать (результат вычитания говорит о том, на сколько элементов один адрес отстоит от другого):

```
int number=pn2 - pn1; //результат вычисляется компилятором следующим  
                          образом: это разность значений, поделенная  
                          на sizeof(типа), в данном случае 5
```

Пример 4.

Два указателя одного типа можно всегда сравнить на равенство/неравенство (т. е. выяснить, указывают ли они на один и тот же объект):

```
if(pn2 == pn1){...} //или !=
```

А если указываемые объекты не только одного типа, но еще и последовательно расположены в памяти (например, содержат адреса элементов одного массива), то их также имеет смысл сравнивать с помощью операторов: `<`, `<=`, `>`, `>=`.

```
if(pn2 > pn1) {...}
```

ЗАМЕЧАНИЕ

Сложение указателей бессмысленно — невозможно никаким образом интерпретировать такое действие, поэтому оно запрещено (компилятор выдаст ошибку)!

6.1.5. Указатель типа *void**

Любой указатель содержит адрес, значение которого не зависит от того, какого типа указываемый объект. Информация о типе указываемого значения становится необходимой компилятору только тогда, когда он оперирует содержимым по адресу, хранящемуся в указателе.

В языке C/C++ существует специальный вид указателя — **void***. Такие указатели обладают важным свойством — они могут содержать *адрес объекта любого* типа (листинг 6.1). Но только до тех пор, пока программисту не понадобится оперировать содержимым по этому адресу (т. е. это лишь отсрочка указания типа!). Как только компилятор встретит в программе любые действия над `void*`-указателем, он потребует от программиста указаний, каким

образом интерпретировать адрес, иначе компилятор не сможет сгенерировать низкоуровневый код. Это означает, что программист должен объяснить компилятору, каков на самом деле тип указываемого значения.

ЗАМЕЧАНИЕ

Хотя тип **void** является базовым (см. разд. 3.4.2), он не является самостоятельным типом, а может использоваться только в более сложных объявлениях. Применительно к указателям, ключевое слово **void** говорит об отсутствии информации о свойствах указываемого объекта, в частности — данных о размере объекта.

Листинг 6.1. void*-указатель можно направить на объект любого типа

```
void* pVoid; //определение void*-указателя. Компилятор
              зарезервировал память для переменной,
              но адрес в ней еще не сформирован

int n=1;
char c='A';
int* pn=&n;
pVoid=&n; //присвоили void*-указателю адрес переменной типа int
pVoid=&c; //а теперь занесли в тот же указатель адрес переменной
          типа char
pVoid=pn; //и это корректно
```

Компилятор позволит присвоить void*-указателю адрес любого объекта, при этом он осуществляет неявное преобразование типа (например, **int*** в **void***). Но обратное неверно!

Чтобы манипулировать void*-указателем, необходимо явно объяснить компилятору, каков тип указываемого объекта. Для явного приведения void*-указателя к указателю на требуемый тип используется оператор приведения **static_cast** (листинг 6.2), т. к. в таких ситуациях компилятор доверяет программисту, считая, что последний знает, что делает. При этом следует учесть, что компилятор на самом деле не знает типа объекта, на который указывает void*-указатель, поэтому результат явного преобразования типа лежит полностью на совести программиста.

Листинг 6.2. Манипулирование void*-указателем

```
//Это продолжение листинга 6.1.
//pn = pVoid; //ошибка: несмотря на то, что обе переменные
              содержат адрес, компилятор заставит программиста
```

уточнить, адрес объекта какого типа содержит
void*-указатель

```
pn=static_cast<int*>(pVoid); //явное приведение типа void* к типу
                               int* посредством нового оператора
                               приведения типа

pn=(int*) pVoid; //явное приведение в старом стиле (согласно
                  стандарту данная форма устарела и
                  предпочтительным является использование
                  static_cast)

//int m=*pVoid; //ошибка: для того, чтобы получить значение
                  по адресу, нужно знать, какого типа это значение

int n=(static_cast<int*> (pVoid) ); //получение значения по
                                   адресу, хранящемуся в void*-указателе
```



Подумайте, являются ли корректными оба выражения и чему будет равен результат?

```
size_t n1 = sizeof(pVoid);
size_t n2 = sizeof(*pVoid);
```

ЗАМЕЧАНИЕ

Как правило, небезопасно обманывать компилятор и приводить **void***-указатель к указателю на тип, отличный от того, на который на самом деле **void***-указатель в данный момент указывает.

Например, форматы представления в памяти целых (см. разд. П1.2—П1.5 приложения) и плавающих чисел (см. разд. П1.11 приложения) совершенно разные, поэтому в следующем примере наивный программист может получить странные результаты:

```
int n = 99;
void* pVoid = &n; //void*-указателю присвоили адрес переменной
                  типа int
float d = *(static_cast<float*>(pVoid)); //для тех, кто не знает
                                         форматов хранения целых и плавающих, но надеется получить 99.0,
                                         результат окажется "приятным" сюрпризом
```

6.1.6. Нулевой указатель (**NULL-pointer**)

Нулевое значение указателя в языке C/C++ имеет специальное назначение — это признак того, что указатель никуда не указывает и, следовательно, не может быть использован. В частности, если на момент определения указателя

по каким-то причинам его невозможно инициализировать конкретным адресом, рекомендуется присвоить такому указателю нулевое значение. Это делает вашу программу более устойчивой, т. к. выявить нулевой указатель при выполнении программы гораздо легче, чем неинициализированный.

Например:

```
int* p = 0; //нулевое значение — это признак того, что указатель
           никуда не направлен
... //предполагается, что по ходу выполнения программы указателю
    будет присвоен адрес переменной
if(p) {...} //если указатель ненулевой, то можно его использовать
           для получения значения
else {...} //а если указатель остался нулевым, то нужно
           предпринимать какие-то другие действия
```

Вместо нуля для обозначения такого признака можно использовать специальное макроопределение NULL, которое определено в нескольких заголовочных файлах стандартной библиотеки (в частности, в <stdlib>, <string>, <tchar.h>), следующим образом:

```
#ifndef NULL
#define __cplusplus
#define NULL 0
#else
#define NULL ((void *)0)
#endif
#endif
```

В DEBUG-версии проекта стандартная библиотека предоставляет макрос `assert()` (см. разд. 5.2.3), который можно, в частности, использовать для выявления нулевых указателей.

Например:

```
int* p = 0;
... //программист уверен, что при выполнении программы в указателе
    обязательно будет сформирован корректный адрес
assert(p!=NULL); //а вдруг указатель все-таки остался нулевым?
                Тогда такую ситуацию можно будет отследить
                на этапе отладки
//или assert(p)
```

6.1.7. Указатель на указатель

Сам указатель (как и любая переменная) тоже имеет адрес, значение которого можно получить аналогично получению адреса любой переменной (рис. 6.4, листинг 6.3).

```
T object; //переменная object - это объект типа T, где под T
           имеется в виду любой тип
T* p = &object; //переменная p содержит адрес объекта типа T
T** pp = &p;    //переменная pp содержит адрес указателя
                на объект типа T
//и т.д. как в сказке "Дом, который построил Джек".
```

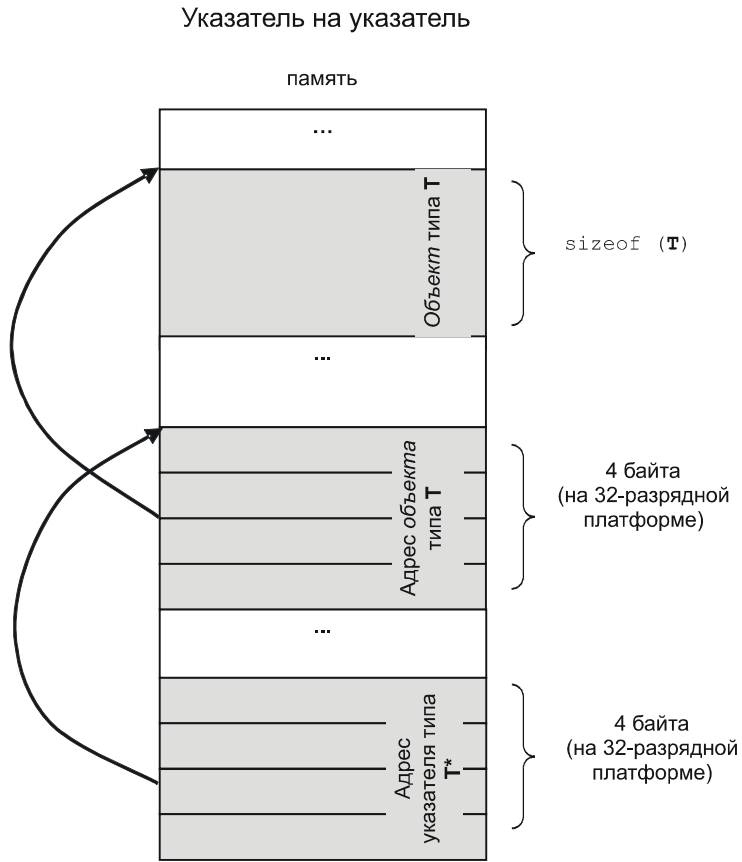


Рис. 6.4

Листинг 6.3. Конструкции типа указатель на указатель

```
double d=1.1;
double* pd=&d;
double** ppd=&pd;
double*** pppd=&ppd;

//а теперь последовательно разыменовывая каждый указатель,
//получим значение переменной d:
double d1 = *pd;
d1 = **ppd;
d1 = ***pppd;

// к каждому указателю применимы правила арифметических действий
// с указателями:
pd++; //значение адреса изменится на sizeof(double)
}
```



На сколько байтов изменится значение каждого указателя после инкремента?

```
ppd++;
pppd++;
```

6.1.8. Указатель и ключевые слова *const* и *volatile*

Ключевые слова **const** и **volatile** (см. разд. 3.10) могут быть использованы при объявлении указателя. Эти слова являются указанием компилятору, каким образом следует в дальнейшем обращаться с таким указателем (в отличие от обычных указателей).

ЗАМЕЧАНИЕ

Далее в основном рассматривается использование ключевого слова **const**. Правила для **volatile** такие же.

Ключевое слово **const** может относиться к значению адреса, или к значению указываемого объекта, или к обоим значениям одновременно. Как объяснить компилятору, что именно является константой? Если ключевое слово **const** находится в объявлении слева от *, то константным является указываемое значение, а если справа от * — то сам указатель.

Рассмотрим специфику использования ключевого слова **const (volatile)**.

Пример 1.

Указываемое значение является константой (листинг 6.4).

Листинг 6.4. Константное указываемое значение

```

{
    const char* pc; //указатель на константное значение. Так как сам
                    указатель не является константой, инициализация
                    при объявлении не обязательна (разд. 3.10.1)
    char const* pc1; //или так (граница константности проходит по *)
    pc="ABC"; //направили указатель на строковый литерал.
                Так как строковый литерал доступен только для чтения
                (во всяком случае, в защищенном режиме процессоров x86
                (разд. 3.2.4), то посредством указателя нельзя
                допускать модификации значения по адресу

    char c = *pc; //читать можно
    pc++; //изменять сам указатель можно
    //*pc = 'W'; //ошибка: изменять содержимое по адресу запрещено
    volatile char* pv; //указываемое значение может быть изменено
                       извне, поэтому компилятор не должен
                       оптимизировать действия с содержимым
                       посредством данного указателя

//Замечание. Компилятор C++ не позволит присвоить обычному указателю
                адрес, хранящийся в указателе на константное значение,
                т. к. иначе появилась бы возможность модифицировать
                константное значение посредством обычного указателя

    //char* p=pc; //ошибка, иначе с помощью p можно было бы изменить
                значение по адресу, которое посредством pc было
                защищено от модификации

    //char* p1=pv; //ошибка, иначе компилятор не смог бы отменить
                оптимизацию при манипулировании содержимым
                посредством p1
}

```

Пример 2.

Указатель является константой — адрес изменять нельзя (листинг 6.5).

Листинг 6.5. Константный указатель

```

{
    char c='A';
    char* const pc = &c; //указатель является константой.

```

```

                                Инициализация обязательна!
*pc = 'B'; //ничто не мешает изменять значение по этому адресу
//pc++; //ошибка: попытка изменения адреса
                (запрещено перенаправлять такой указатель)
extern char* volatile pv; //значение указателя может быть изменено
                        извне, поэтому компилятор не должен
                        оптимизировать действия с указателем
}

```

Пример 3.

И указатель, и указываемое значение являются константами (листинг 6.6).

Листинг 6.6. Константный указатель на константное значение

```

{
    int x=1, z=2;
    const int* const pn =&x; //константный указатель на константное
                            значение. Инициализация обязательна
    //или int const* const pn =&x; //или так
    int y=*pn; //читать можно
    /**pn = 5; //ошибка: модифицировать значение по адресу запрещено
    //pn = &z; //ошибка: модифицировать адрес запрещено
}

```

Пример 4.

Если требуется сформировать указатель на константную переменную, то такой указатель должен быть также объявлен как указатель на константу (листинг 6.7).

Листинг 6.7. Формирование указателя на константную переменную

```

{
    const int n = 1;
    //int* pn1 = &n; //ошибка: иначе появилась бы возможность
                    модифицировать константное значение посредством
                    такого указателя, например (*pn1)++;
    const int* pn1 = &n; //корректно: такой указатель можно
                    использовать только для чтения значения
}

```

//Замечание. Это как раз тот случай, когда даже оптимизирующий компилятор C++ будет не просто подставлять значение 1 везде, где встретит имя *n*, а отведет память под такую переменную (см. разд. 3.10.1), т. к. программисту понадобился адрес этой переменной.

}

Пример 5.

Рассмотрим неявное преобразование *const*-указателей (*volatile*-указателей).

Компилятор считает корректным присвоение значения константного указателя обычному не константному указателю того же типа (листинг 6.8).

Листинг 6.8. Неявное приведение типа *T* const* в *T**

```
{
    int *const cp = &x; //указатель cp модифицировать запрещено
    int *p;    //а это совсем другая переменная, поэтому она имеет
                право изменяться
    p = cp;    //корректно. Неявное приведение типа int* const к int*.
                Обе переменные содержат один и тот же адрес, но
                одну позволено модифицировать, а другую — нет
    p++;       //корректно
    //cp++;    //ошибка: попытка модификации константного указателя
    //cp = p;  //ошибка: попытка присвоить новое значение константному
                указателю
}
```

Возможно преобразование *T** в *const T** (т. е. если речь идет об ужесточении ограничений на действия с указателем, компилятор тоже справляется сам неявно). Обычно такая ситуация возникает при вызове функции (подробную информацию о функциях см. в гл. 8), принимающей в качестве параметра указатель вида *const T** (это означает, что внутри функции нельзя модифицировать значение по переданному адресу, а можно только читать). При этом в вызывающей части программы этот адрес может быть предназначен как для чтения, так и для записи (листинг 6.9). Обратное неверно: смягчить ограничения (*const T** в *T**) компилятор не позволит!

Листинг 6.9. Неявное приведение типа *T** в *const T**

```
void f1(const int* p)
{
    ((*p)++); //ошибка: нельзя модифицировать значение по адресу
```



```

}
void f2(int* p)
{
    (*p)++; //а в этой функции изменять значение посредством указателя
            разрешено
}
int main()
{
    int m=1;
    int* pm = &m;
    (*pm)++; //ОК: модифицировать значения посредством указателя
            разрешено
    f1(pm); //при вызове функции компилятор преобразует тип int* к
            типу const int*
    const int* pc = &m;
    //(*pc)++; //ошибка: посредством этого указателя значение изменять
            нельзя
    //f2(pc); //ошибка: компилятор не может преобразовать const int*
            к типу int*
}

```

Пример 6.

Нельзя void*-указателю присвоить адрес переменной, объявленной как **const** или **volatile** (листинг 6.10).

Листинг 6.10. void*-указатель и ключевые слова const и volatile

```

{
    const int m = 1;
    //void* p1 = &m; //ошибка: если бы компилятор такое допускал,
                    то с помощью void-указателя можно было бы
                    модифицировать значение по адресу
    void const* p1= &m; //ОК: такой указатель может указывать на все,
                    что угодно (при условии, что это const)
    extern volatile int v;
    //void* p2 = &v; //ошибка: если бы компилятор такое допускал,
                    то не смог бы отменить оптимизации при
                    обращении к volatile-объекту посредством
                    void*-указателя
    void volatile* p2 = &v; //ОК
}

```

Пример 7.

Сложные объявления:

```
const int* p1,*p2; //это два указателя типа const int*,
                  инициализация необязательна

int n = 1;

int* const p3=&n,*p4; //p3 - константный указатель,
                     инициализация обязательна,
                     p4 - обычный указатель типа int*
```



Подумайте, что означает каждое объявление, и если по вашему мнению требуется инициализация, то исправьте:

```
int* const p5,* const p6;
int * const * p7;
```

6.1.9. Явное преобразование типа указателя

Оператор `const_cast`

Иногда возникает необходимость снять константность с указателя или с указываемого значения. Применять такие преобразования стоит только в тех редких случаях, когда это действительно необходимо, т. к. при этом защитные возможности компилятора уменьшаются. Приведенные примеры демонстрируют только механизм отмены константности и на содержательность не претендуют.

Пример 1.

Указатель является константой (листинг 6.11).

Листинг 6.11. Как снять константность с указателя

```
{
    char c;
    char* const pc=&c; //указатель является константой
    //pc++; //ошибка: нельзя модифицировать константу
    (const_cast<char*>(pc))++; //но если очень хочется...
                               (и только на время вычисления данного выражения)
}
```

Пример 2.

Указываемое значение является константой (листинг 6.12).

Листинг 6.12. Как снять константность с указываемого значения

```

void f(char*); //функция принимает обычный указатель, но гарантированно
               не модифицирует значение по адресу

int main()
{
    char c = 'A';
    const char* pc = &c; //посредством указателя pc значение
                          //переменной c изменять нельзя
    /*pc = 'B'; //ошибка: указываемое значение модифицировать нельзя
    *(const_cast<char*>(pc)) = 'B'; //ОК, хотя это злостное нарушение
                                     введенных ограничений, и
                                     грамотный программист
                                     так поступать не должен
    //f(pc); //ошибка: компилятор самостоятельно не может смягчить
               ограничения и преобразовать const char* в char*
    f(const_cast<char*>(pc)); //ОК
}

```

Оператор *reinterpret_cast*

Компилятор не может вычислить выражение, если в нем используются операнды разных типов (см. разд. П1.10 — П1.12 приложения).

Поэтому:

- ❑ если компилятор считает приведение типа безопасным (например, **int** в **double**), то осуществляет такие преобразования неявно (см. разд. 3.4.5):

```

int n = 33;
double d = n; //компилятор осуществляет неявное приведение типа
               int в double

```

- ❑ если (хороший) компилятор подозревает о возможной потере точности (**double** в **int**), то выдает предупреждение:

```

double d = 99.99;
int n = d; //компилятор выдает предупреждение и осуществляет
            неявное приведение типа double в int

```

- ❑ если компилятор доверяет приведение типа программисту (**double** в **int**, **void*** в **T***), то позволяет использовать оператор **static_cast** для явного приведения типа (см. разд. 3.4.5 и 6.1.5):

```

double d = 99.99;
int n = static_cast<int>(d); //компилятор осуществляет явное

```

приведение типа `double` в `int`. Предупреждений нет!

```
void* pv = &d;
double* pd = static_cast<double*>(pv);
```

- если компилятор считает преобразование потенциально опасным, то выдает ошибку (при этом `static_cast` не поможет!).

В последнем случае с помощью оператора явного приведения типа `reinterpret_cast` можно заставить компилятор преобразовать целое значение к типу указателя (и наоборот) или указатель на объект одного типа к указателю на объект другого типа. При этом следует учитывать, что объект, адресуемый приведенным указателем, будет интерпретироваться в соответствии с перепределенным типом. Такие преобразования нужно применять с большой осторожностью (и только если вы, с одной стороны — уверены, что такое преобразование необходимо, а с другой стороны — четко представляете последствия такого преобразования).

Пример 1.

В распоряжении программиста есть значение адреса, но для того чтобы можно было по такому адресу обращаться, необходимо объявить указатель и присвоить ему это значение:

```
//int* p = 0x10000000; //ошибка: нельзя преобразовать int в int*
int* p = reinterpret_cast<int*>(0x10000000); //ОК
```

Пример 2.

С помощью указателей разного типа можно заставить компилятор интерпретировать одну и ту же область памяти по-разному (листинг 6.13, рис. 6.5).

Листинг 6.13. Интерпретация одной и той же области памяти по-разному посредством указателей разного типа

```
{
    int n = 0x11223344; //целое можно интерпретировать как
                        //совокупность четырех последовательно
                        //расположенных в памяти байтов

    char c = n; //в этом случае компилятор выдаст предупреждение
                //о возможной потере значения и осуществит неявное
                //преобразование int к типу char
                //(т. е. поместит в переменную c значение младшего
                //байта переменной n — см. разд. 3.4.5).

    int* pn = &n; //посредством такого указателя можно работать
                  //с совокупностью из четырех байтов (int) только
```

как с единым целым. А для того, чтобы работать с каждым байтом независимо, нужен указатель типа `char*`

```
//char* pc = &n; //ошибка: компилятор считает такое преобразование  
//небезопасным
```

```
char* pc = reinterpret_cast<char*>(&n); //указатель pc содержит  
//тот же самый адрес, что и указатель pn, но  
//посредством указателя pc можно работать с каждым  
//байтом отдельно!
```

```
c = *pc; //c=0x44
```

```
//или
```

```
short s=( reinterpret_cast<short*>(pn) ); //s=0x3344
```

```
pc++; //переместили указатель на следующий байт
```

```
c = *pc; //c=0x33
```

```
}
```

Использование явного преобразования типа указателя для интерпретации одной и той же области памяти по-разному

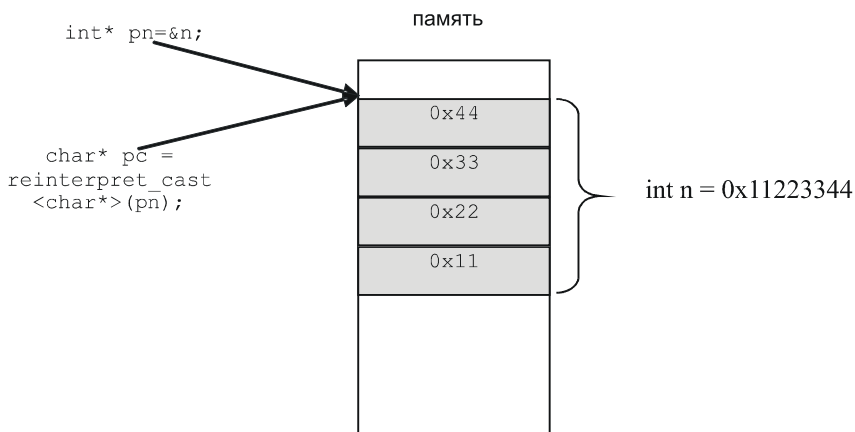


Рис. 6.5

ЗАМЕЧАНИЕ

Используя преобразования такого рода, следует четко представлять себе расположение данных разных типов в памяти (количество занимаемых объектом байтов, смысл и порядок их следования), иначе результаты могут не соответствовать вашим ожиданиям.

Например:

```
double d=99.99; // 0x4058FF5C28F5C28F в 16-чном виде
int* pn = reinterpret_cast<int*> (&d); //компилятор будет
                                     интерпретировать то, что лежит в памяти по адресу,
                                     задаваемому pn, как int
int tmp = *pn; //если вы ожидаете в переменной tmp получить 99
               (целую часть переменной d), то нужно почитать о
               представлении данных в приложении П1.11).
               tmp будет присвоено значение 0x28F5C28F=687194767(10)
```

6.2. Массивы

Массив — это сложный программный элемент (встроенное средство языка), обладающий следующими специфическими особенностями:

- ❑ это совокупность упорядоченных (расположенных *последовательно* в памяти) элементов данных;
- ❑ все элементы массива — одного и того же типа;
- ❑ все элементы массива занимают непрерывную линейную область памяти;
- ❑ каждый массив имеет имя. Имя массива — это база (адрес участка памяти, начиная с которого хранятся элементы массива), относительно которой можно обратиться к любому элементу массива;

ВАЖНОЕ ЗАМЕЧАНИЕ

В языке C/C++ имя массива имеет смысл сложного указателя, тип которого зависит не только от типа элементов, но и от размерностей массива (см. разд. 6.3).

- ❑ все элементы массива пронумерованы, начиная с 0. Доступ к отдельному элементу массива может осуществляться одинаково эффективно по имени массива и порядковому номеру элемента:
 - с помощью оператора индексирования [] (см. разд. 6.2.2);
 - через указатель, посредством оператора разыменования * (см. разд. 6.3);
- ❑ можно задать одномерный, двухмерный, многомерный массив, указав требуемое количество размерностей.

6.2.1. Объявление массива

Перед использованием массива программист должен описать его свойства компилятору, т. е. использованию должно предшествовать объявление.

Тремя обязательными составляющими при описании являются:

- ☐ имя массива;
- ☐ тип элементов;
- ☐ размерность массива.

Размерности задаются в индексных скобках `[]` в следующем порядке:

`T a1[5];` //это одномерный массив с именем `a1` из 5 элементов типа `T`

`T a2[5][10];` //это двумерный массив с именем `a2`, в котором 5 строк.

В каждой строке по 10 элементов типа `T`

`T a3[5][10][20];` //это трехмерный массив с именем `a3`, в котором 5 слоев.

В каждом слое по 10 строк.

В каждой строке по 20 элементов типа `T`

//и т.д.

Как и для обычных переменных (см. разд. 3.5), объявление может быть совмещено с определением, либо речь может идти об объявлении внешнего (`extern`) массива. Отличия приведены в табл. 6.4.

ЗАМЕЧАНИЕ

Размерность массива может быть задана только *константным выражением* (т. е. на момент компиляции компилятор должен знать, сколько выделить памяти!).

Таблица 6.4. Объявление и определение массива

Без ключевого слова <code>extern</code>	С использованием ключевого слова <code>extern</code>
Объявление совмещено с определением (т. е. компилятор должен выделить память), поэтому от программиста требуется предоставить компилятору информацию о том, <i>сколько требуется выделить памяти</i> . Это программист может сделать двумя способами: явно указать все размерности или предоставить компилятору вычислить (только!) старшую размерность по списку инициализаторов (см. разд. 6.2.3)	Это только объявление (описание свойств) внешнего массива, определенного в другом файле, поэтому такие объявления рекомендуется помещать в заголовочный файл. При объявлении программист должен предоставить компилятору информацию о том, <i>как вычислять адрес</i> любого элемента. А т. к. в вычислении адреса участвуют все младшие размерности массива (см. разд. 6.3), то программист обязан при объявлении явно указать все младшие размерности!
Такой массив может быть, исходя из контекста объявления, глобальным, локальным, статическим или заключенным в пространство имен	Такой массив может быть только глобальными или заключенным в именованное пространство имен

Таблица 6.4 (окончание)

Без ключевого слова <code>extern</code>	С использованием ключевого слова <code>extern</code>
Примеры: <pre>char cAr [10]; //компилятор должен зарезервировать память под одномерный массив из 10 элементов типа char int nAr [10][5]; //двухмерный массив из 10*5 элементов типа int float fAr [2][10][5]; //трехмерный массив из 2*10*5 элементов типа float</pre>	Примеры: <pre>extern int ar[10][5]; //такое объявление информирует компилятор о том, что на самом деле глобальный массив из 10*5 элементов типа int объявлен в каком-то другом модуле, поэтому память под данный массив уже отведена</pre> или <pre>extern int ar[][5]; //т. к. старшая размерность не участвует в вычислении адреса, то для внешнего массива ее можно опустить</pre>

Пример задания размерности массива с помощью ключевого слова `const` (только в C++):

```
const int N = 5;
char ar[N]; //ОК
int N1=5;
//char ar[N1]; //ошибка: размерность должна быть задана константой
```

6.2.2. Обращение к элементу массива — оператор []

Доступ к отдельным элементам массива может выполняться с помощью индекса или нескольких индексов (если массив многомерный). Индексы — это средство, которое позволяет компилятору, исходя из размерности массива, вычислить адрес требуемого элемента (как это делает компилятор — см. разд. 6.3). При этом для повышения эффективности вычисления адресов индексы нумеруются с 0, а не с 1. Поэтому корректный диапазон изменения индекса — от 0 до числа, на 1 меньшего размерности массива.

ЗАМЕЧАНИЕ

Индекс может быть задан как числом, так и любым целым выражением.

Пример обращения к элементу одномерного массива приведен в листинге 6.14.

Листинг 6.14. Обращение к элементу одномерного массива

```

{
    char cAr[10]; //определение одномерного массива
    char c = cAr[0]; //обращение к нулевому элементу массива
    c=cAr[9]; //обращение к последнему элементу массива
    int i, j, k;
    //вычисление значений i, j, k
    c=cAr[i]; //обращение к i-тому элементу массива
    c=cAr[i+j+k]; //можно и так
    //c=cAr[5.]; //ошибка: индекс должен быть целым
//Замечание. В языке C/C++ для повышения эффективности вычислений не
    реализован автоматический контроль выхода значения индекса
    за допустимые пределы, поэтому компилятор не выдаст ошибки
    при следующих присваиваниях, а результат будет непредсказуем
    c=cAr [10]; //обращение к несуществующему элементу массива,
                будет чтение из памяти, занятой неизвестно чем
    c=cAr [-1]; //аналогично
    int n = <вычисление_n>; //переменная n в результате вычислений
                            может принять любое значение
    cAr [n]='A'; //а в этом случае корректность операции зависит
                от значения переменной n
}

```

Замечание о синтаксисе.

Если *a* — имя массива, *b* — имя целочисленной переменной, то все три нижеприведенных выражения будут корректны и результат будет одинаковым:

- *a*[*b*];
- *b*[*a*];
- **(a + b)*.

При доступе к элементу многомерного массива должно быть указано столько индексов, сколько размерностей у массива. При этом, например, для трехмерного массива в выражении *ar*[*i*][*j*][*k*] индексы указывают:

- *i* (старший индекс) — номер слоя (т. е. двухмерного массива);
- *j* — номер строки в *i*-том слое (т. е. одномерного массива);
- *k* — номер элемента в *j*-той строке.

Пример приведен в листинге 6.15.

Листинг 6.15. Обращение к элементу многомерного массива

```
int nAr[3][4]; //объявление двумерного массива
int n=nAr[0][0]; //обращение к нулевому элементу массива.
    При этом старший (левый) индекс является номером строки,
    младший (правый) — номером элемента в строке
int n=nAr[2][3]; //обращение к последнему элементу массива
```

6.2.3. Инициализация массива

Инициализация массива может быть:

- неявной (выполняется автоматически компилятором);
- явной (выполняется программистом).

Неявная инициализация

По аналогии с переменными базового типа (см. разд. 3.9.2) все элементы глобальных массивов, заключенных в пространства имен, а также и статических инициализируются компилятором по умолчанию нулями, локальных и динамических массивов — не инициализируются вовсе (листинг 6.16). При этом программист должен указать все размерности массива, для того чтобы компилятор знал, сколько зарезервировать памяти.

Листинг 6.16. Неявная инициализация массивов

```
int ar1[10][5]; //глобальный массив, проинициализирован компилятором
                нулями
namespace {char ar1[10][5];} //массив в неименованном пространстве имен,
                             проинициализирован компилятором нулями
int main()
{
    static char ar3[100]; //статический массив, проинициализирован
                          нулями
    float ar4[3][4][5]; //локальный массив, не проинициализирован
    //double ar5[]; //ошибка: программист не предоставил компилятору
                    информацию о том, сколько зарезервировать памяти
}
```

Явная инициализация

Только при определении программист может указать компилятору, какими значениями инициализировать элементы массива, причем сделать это можно несколькими способами.

Рассмотрим первый способ.

Программист может явно указать все размерности массива. Таким образом он предоставляет компилятору информацию о том, сколько зарезервировать памяти.

Примеры инициализации одномерных массивов:

Пример 1.

```
int Ar1[3] = {1,2,3}; //для каждого элемента массива указан  
                        инициализатор
```

Пример 2.

Инициализирующий список может содержать меньше начальных значений, чем количество элементов в массиве, тогда остальные элементы инициализируются нулем по правилам неполной инициализации массивов (независимо от того, глобальный массив или локальный!):

```
char cAr [10] = {'A', 'B', 'C'}; //первые три элемента  
                                инициализируются кодами символов 'A', 'B', 'C',  
                                значения остальных семи элементов — нулями  
  
char cAr1[10] = {"ABC"}; //можно использовать для инициализации  
                        такого массива строковый литерал. В данном примере  
                        результат будет таким же, как и в предыдущем,  
                        т. к. первые четыре элемента массива будут  
                        заполнены копиями значений строкового литерала,  
                        включая завершающий 0, а остальные — по правилам  
                        неполной инициализации компилятор проинициализирует  
                        нулями  
  
char cAr2[10] = "ABC"; //скобки в случае инициализации одномерного  
                        массива строковым литералом можно опустить
```

//Но! если список инициализаторов содержит больше элементов, чем размерность массива (значение в индексных скобках), выдается сообщение компилятора:

```
char cAr2[3] = "ABC"; //ошибка: слишком много инициализаторов  
                      (см. разд. 3.2.4)
```

Примеры инициализации многомерных массивов.

Пример 1.

При инициализации многомерных массивов инициализирующие значения нужно указывать в том порядке, в котором компилятор выделяет память под

элементы массива (а выделяет он память линейно и непрерывно), поэтому если указаны все инициализаторы, то можно написать:

```
int nAr [2][3]={ 1, 1, 1, //инициализаторы для нулевой строки
                2, 2, 2 //инициализаторы для первой строки
};
```

Пример 2.

Можно с помощью фигурных скобок уточнить компилятору, для какой конкретно строки программист указывает инициализаторы:

```
int nAr [3][2]={ {1, 1}, //инициализаторы для нулевой строки
                 {2, 2}, //для первой строки
                 {3, 3} //для второй строки
};
```

Пример 3.

Для многомерных массивов также справедливы правила неполной инициализации (рис. 6.6):

```
int nAr [3][2]={ {1},
                 {2, 2},
                 };
```

Правила неполной инициализации многомерных массивов

1	0
2	2
0	0

Рис. 6.6

Пример 4.

Инициализация строковыми литералами:

```
char cStringAr [2][80]={ "Первая строка",
                          "Вторая строка"
}; //компилятор зарезервирует 2*80*sizeof(char)
байтов и скопирует в каждую строку массива
значения соответствующего строкового
литерала (оставшиеся в каждой строке байты
будут проинициализированы нулями)
```

Рассмотрим второй способ.

Только при наличии списка инициализаторов при определении массива можно опустить старшую размерность (или оставить ее открытой). При этом компилятор по списку инициализаторов сам вычислит, сколько требуется выделить памяти.

Пример 1.

Можно объявить одномерный массив без указания числа элементов массива, а только со списком начальных значений:

```
char cAr []={'A', 'B', 'C'}; //в результате создается одномерный массив
                             из трех элементов и эти элементы получают
                             начальные значения из списка инициализации
```



Определите, сколько элементов в массиве, сколько зарезервировано байтов и как проинициализированы элементы:

```
char cAr1[]="ABC";
```

Пример 2.

Многомерные массивы могут инициализироваться без указания одной самой старшей (или самой левой) из размерностей массива (при этом компилятор определяет опущенную старшую размерность, исходя из списка инициализации).



Определите, сколько строк в массиве, сколько зарезервировано байтов, как проинициализированы элементы:

```
int nAr [][][3] = {
    {0, 1, 2},
    {10, 11, 12},
    {20, 30, 40}
};
```

Если в многомерном массиве необходимо проинициализировать не все элементы, а только несколько первых в каждой строке (слое), то в списке инициализации можно использовать фигурные скобки, охватывающие значения инициализаторов для строки (слоя...). Все оставшиеся элементы будут проинициализированы компилятором нулями по правилу неполной инициализации:

```
int nAr1[][3] = { {0},           //0-ая строка
                  {10, 11},      //1-ая строка
                  {20, 21, 22} }; //2-ая строка
```

Специфика инициализации массивов:

- массив можно проинициализировать с помощью списка инициализации только при определении. Любая попытка с помощью списка инициализации

торов присвоить элементам уже существующего массива новые значения вызовет сообщение компилятора об ошибке:

```
int ar[5];
//ar = {1,2,3}; //ошибка: список инициализаторов можно использовать
                    только при определении
```

- ❑ в некоторых реализациях при указании пустого списка инициализаторов компилятор выдает ошибку, а в других — инициализирует значения всех элементов нулями:

```
int ar[3] = {}; //зависит от реализации (например,
                в Microsoft Visual Studio 2003 и более поздних
                версиях все элементы будут обнулены)
```

- ❑ в списке инициализаторов могут быть использованы переменные:

```
int n1=1, n2 = 2;
int ar[4] = {7,n1,n2,8}; //элементы массива будут
                        проинициализированы значениями 7,1,2,8
```

- ❑ можно при объявлении сообщить компилятору, что все элементы массива являются константами. При этом так же, как и в случае константных переменных базового типа, массив должен быть проинициализирован при определении:

```
//const int ar[5]; //ошибка: требуется инициализация
//но!
extern const int ar[5]; //ОК, т. к. это просто объявление
                        внешнего массива констант
//const int ar[5] = {1,2,3}; //в некоторых реализациях возможна
                        ошибка, т. к. инициализаторов меньше, чем требуется,
                        в других — компилятор проинициализирует оставшиеся
                        элементы нулями (например,
                        Microsoft Visual Studio 2003 и более поздние версии)
//однако!
const int ar[] = {1,2,3,4,5}; //корректно, т. к. компилятор
                        подсчитает количество элементов
                        по списку инициализаторов
```

Пример работы с одномерным массивом

В качестве примера работы с массивом рассмотрим сортировку выбором (листинг 6.17). Алгоритм этой сортировки основан на поиске самого большого (или самого маленького) из оставшихся не отсортированных элементов массива и обмене значениями найденного и текущего элемента (рис. 6.7).

Алгоритм сортировки выбором

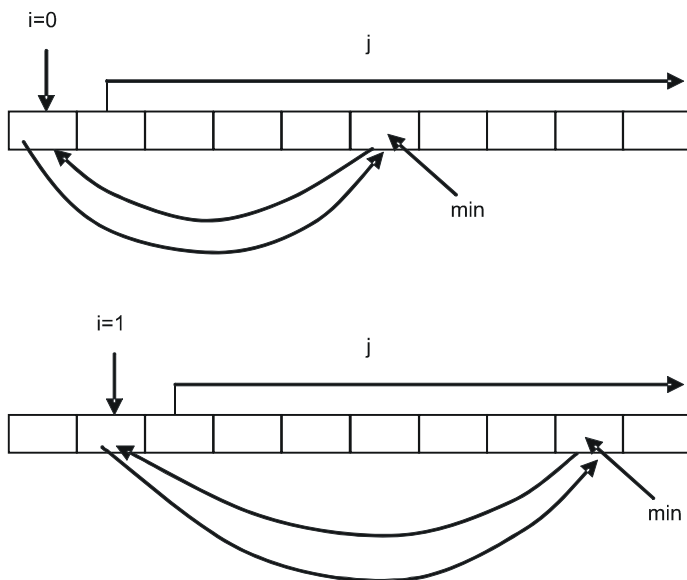


Рис. 6.7

Листинг 6.17. Сортировка выбором по возрастанию значений

```

{
    int ar[]={7,2,-5,6,0,3,2,5,3,-9,99,100};
    int n = sizeof(ar)/sizeof(int); //количество элементов
    for(int i=0; i<n-1; i++) //внешний цикл задает количество итераций
                            (n-1, так как последнее значение
                             автоматически окажется на своем месте)
    {
        int min = i; //здесь будет индекс элемента с минимальным
                     значением
        for(int j=i+1; j<n; j++) //внутренний цикл: поиск
                                наименьшего из оставшихся неупорядоченных значений
        {
            if(ar[j]<ar[min]) min=j;
        }
        //Обмен местами значений текущего и минимального элементов
    }
}

```

```

        int tmp = ar[min];
        ar[min] = ar[i];
        ar[i] = tmp;
    }
}

```

6.2.4. Массивы и оператор *sizeof*

Для *одномерного* массива с помощью оператора **sizeof** можно на этапе компиляции вычислить количество элементов:

```

char ar[] = "abc";
size_t n=sizeof(ar) / sizeof(char); //количество элементов массива
//или
n=sizeof ar / sizeof(ar[0]);          //то же самое

```

Для *двухмерного* массива можно получить информации больше:

```

int ar[][3] = {1, 2, 3, 4, 5, 6, 7};
size_t n=sizeof(ar) / sizeof(int); //количество элементов массива
//или
n=sizeof(ar) / sizeof(ar[0][0]);    //то же самое
n=sizeof(ar[0]) / sizeof(int);       //число элементов в строке
n = sizeof(ar)/ sizeof(ar[0]);       //число строк
//или
n = sizeof(ar)/ sizeof(int[3]);       //число строк

```

Для *трехмерного* массива:

```

size_t n = sizeof(ar)/sizeof(int); //всего элементов
n = sizeof(ar)/ sizeof(ar[0][0]); //количество строк
n = sizeof(ar)/ sizeof(ar[0]);     //количество слоев
n= sizeof(ar[0][0])/sizeof(int);    //элементов в строке

```

Оператор **sizeof** применительно к массивам бывает полезен в следующих случаях:

- если программист решает, что первоначально заданные размерности массива его не устраивают. В таком случае, если он использует абсолютные значения, скорее всего, ему придется еще во многих местах исправлять текст своей программы. Пусть лучше за программиста текущие размерности вычислит компилятор. А т. к. оператор **sizeof** — это механизм этапа компиляции, а не выполнения, то эффективность программы не ухудшается;

- если программист при определении массива не указывает старшую размерность, а предлагает компилятору вычислить ее по списку инициализаторов (листинг 6.18):

Листинг 6.18. Использование оператора sizeof применительно к массивам

```
{
    const int M=2;
    double ar[][M] = { {1.1}, {2.2}, {3.3} };
    double sum=0;
    for(int i=0; i<sizeof(ar)/sizeof(double[M]); i++) //перебираем строки.
                                                С помощью sizeof определяем количество строк
    {
        for(int j=0; j<sizeof(ar[0])/sizeof(double); j++) //перебираем
                                                    элементы в i-той строке
            {sum+=ar[i][j];}
    }
}
```

6.3. Связь массивов и указателей

Теперь рассмотрим связи между массивами и указателями более подробно. Начинаящим рекомендуется перед дальнейшим чтением еще раз просмотреть *важное замечание в разд. 6.2.* и *замечание о синтаксисе в разд. 6.2.2.*

6.3.1. Одномерные массивы

Имя одномерного массива компилятор интерпретирует так же, как интерпретировал бы константный указатель (константный, потому что перенаправить или модифицировать его нельзя) на нулевой элемент массива. А для программиста означает, что этими понятиями можно пользоваться одинаково.

Пусть дан одномерный массив:

```
int ar[5] = {1,2,3,4,5};
```

Элементы такого массива располагаются в памяти линейно и непрерывно (рис. 6.8).

Исходя из интерпретации компилятором имени одномерного массива, можно объявить эквивалентный указатель и направить его на начало массива:

```
int* p = ar; //неявное преобразование имени массива в указатель на
его нулевой элемент (int[] в int*)
```

Одномерный массив из 5 элементов

<code>ar[0]=1</code>
<code>ar[1]=2</code>
<code>...</code>
<code>...</code>
<code>ar[4]=5</code>

Рис. 6.8

Это означает, что к *i*-тому элементу массива теперь можно обращаться как посредством имени массива *ar*, так и посредством указателя *p* четырьмя приведенными способами:

❑ `int tmp = ar[i];`

❑ `tmp = p[i];`

//или согласно правилам арифметики указателей (см. разд. 6.1.4)

то же самое значение можно получить и так:

❑ `tmp = *(p+i);`

❑ `tmp = *(ar+i);`

Несмотря на то, что использование имени массива и эквивалентного указателя выглядит одинаково, существенными отличиями являются:

❑ для переменной-указателя компилятор отводит память, а имя массива это только эквивалент указателя. С этим именем компилятор просто ассоциирует адрес начала массива на этапе компиляции (никакой памяти для хранения такого адреса не отводится);

❑ в большинстве случаев программист может синтаксически одинаково использовать как имя массива, так и соответствующий ему указатель. Исключением является попытка модификации:

`p++;` *//ОК, указатель перемещается на следующий элемент массива*

`//ar++;` *//ошибка: ar не является lvalue*

Для вычисления адреса произвольного *i*-ого элемента одномерного массива компилятору достаточно знать адрес начала массива (адрес нулевого элемента) и размер элемента:

`адрес_элемента_i = адрес_начала_массива + i * sizeof(тип_элемента)`

Важно!

Обратите внимание: размерность массива при вычислении адреса элемента нигде не фигурирует!

В качестве примера перепишем алгоритм сортировки выбором (см. разд. 6.2.3), используя указатели вместо индексов (листинг 6.19).

Листинг 6.19. Сортировка массива методом выбора посредством использования указателей

```
{  
    int ar[] = {4,1,-10, 55, 2,-5};  
    size_t n = sizeof(ar)/sizeof(ar[0]); //количество элементов  
    int* pCur = ar; //на нулевой элемент  
    for(int i=0; i<(n-1); i++) //внешний цикл задает количество  
                               итераций  
    {  
        int* pMin = pCur; //в pMin будет адрес элемента с самым  
                             маленьким из оставшихся значением.  
                             На каждой итерации изначально  
                             предполагаем, что самый маленький  
                             элемент — текущий  
        int* pTmp = pCur + 1; //с помощью pTmp будем перебирать  
                                оставшиеся до конца массива  
                                элементы  
        for(int j=i+1; j<n; j++) //внутренний цикл — поиск самого  
                                 маленького из оставшихся  
                                 неупорядоченных значений  
        {  
            if(*pTmp < *pMin) pMin = pTmp; //если очередное  
                                             значение оказалось меньше,  
                                             перенаправляем на него pMin  
            pTmp++; //сдвигаем указатель на следующий элемент  
        }  
        //Меняем местами значения, на которые указывают pMin и pCur  
        int tmp = *pMin;  
        *pMin = *pCur;  
        *pCur = tmp;  
        //Подготавливаем pCur к следующей итерации внешнего цикла  
        pCur++;  
    }  
}
```

6.3.2. Двухмерные массивы более подробно

Двухмерный массив — это способ организации данных как прямоугольной таблицы, содержащей несколько строк одинаковой длины. Это означает, что двухмерный массив можно представить (как и делает компилятор) как одномерный, каждым элементом которого является строка (т. е. одномерный массив). А имя двухмерного массива компилятор интерпретирует как константный указатель на нулевую строку.

Пусть дан двухмерный массив:

```
const int N=2, M=3;
int ar[N][M] = { {1,2,3},{4,5,6}};
```

Тот факт, что массив двухмерный, означает:

- удобную для программиста интерпретацию данных (прямоугольная таблица);
- правила вычисления компилятором адреса `[i][j]`-го элемента.

А на самом деле элементы такого массива располагаются в памяти линейно и непрерывно (рис. 6.9).

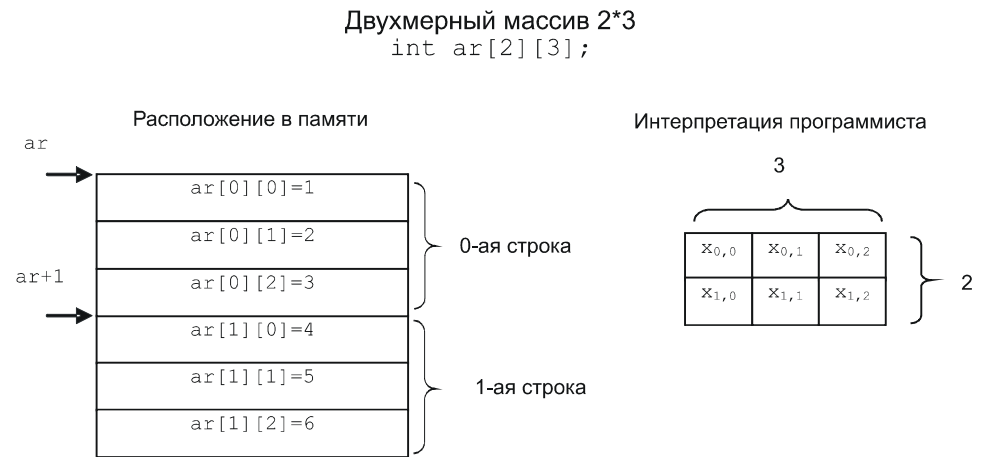


Рис. 6.9

Для вычисления адреса произвольного `[i][j]`-го элемента двухмерного массива элементов типа `T` компилятору нужно знать адрес начала массива, количество элементов в строке и размер элемента:

адрес_элемента_i_j = адрес_начала_массива + i*M*sizeof(T) + j*sizeof(T)

ОБРАТИТЕ ВНИМАНИЕ!

В вычислении адреса участвует только младшая размерность массива, а старшая размерность нигде не фигурирует.

Исходя из интерпретации компилятором имени двухмерного массива, введем эквивалентный указатель:

```
int (*p)[3] = ar; //скобки обязательны, иначе для компилятора  
смысл объявления будет совершенно другим!
```



Подумайте, что означает такое объявление:

```
int *p[3];
```

Теперь к `[i][j]`-тому элементу массива можно обращаться, используя имя массива `ar` или указатель `p`. При этом можно использовать любые комбинации операторов `[]` и `*` (а низкоуровневый код, который сгенерирует компилятор, будет во всех случаях одним и тем же):

```
int tmp = ar[i][j];  
tmp = (*(ar+i)+j);  
tmp = *(ar[i]+j);  
tmp = *(ar+i)[j];  
//и то же самое посредством указателя:  
tmp = p[i][j];  
tmp = (*(p+i)+j);  
tmp = *(p[i]+j);  
tmp = *(p+i)[j];
```

ЗАМЕЧАНИЕ

По правилам арифметики указателей при вычислении выражения `p++`; компилятор переместит указатель на следующую строку массива! Тот же самый адрес он вычислит в выражении `(ar+1)`.

При работе со встроенными массивами программист, зная о том, как они располагаются в памяти (линейно и непрерывно), может оптимизировать вычисления.

Пусть требуется подсчитать сумму элементов двухмерного массива (листинг 6.20).

Листинг 6.20. Вычисление суммы элементов двухмерного массива с помощью вложенных циклов

```
{  
  
    int ar[N][M]; // N и M – константы  
    //формирование значений элементов массива
```

```

int sum=0;
for(int i=0; i<N; i++) //внешний цикл перебирает строки
{
    for(int j=0; j<M; j++) //внутренний цикл перебирает
                           элементы в строке
    {
        sum+=ar[i][j]; //в этом месте компилятор по
                        приведенной выше формуле будет на каждой итерации
                        цикла вычислять адрес, используя операции умножения
                        и сложения (вычислений будет много!)
    }
}
}

```

Имейте в виду, что время выполнения программы может существенно увеличиться за счет того, что на каждой итерации цикла при вычислении адреса `[i][j]`-го элемента компилятор использует достаточно много вычислений. А если `N` и `M` будут достаточно большими?

На самом деле для решения задачи нужно просто последовательно перебрать значения элементов. В таких местах программист, зная о том, что элементы массива расположены в памяти последовательно, может оптимизировать выполнение программы (листинг 6.21), введя вспомогательный указатель, посредством которого можно будет просто перемещаться от элемента к элементу, не думая о том, каким строкам они соответствуют (т. е. рассматривать такой двухмерный массив как одномерный!). Объем вычислений резко сократится.

Листинг 6.21. Оптимизация вычислений посредством использования указателя

```

{
    int* p = &ar[0][0]; //вспомогательный указатель, который содержит
                        адрес начального элемента
    //или эквивалентное выражение: int* p = ar[0];
    for(int i=0; i<sizeof(ar)/sizeof(int); i++)
    {
        sum += *p; //добавляем к сумме значение по текущему адресу
        p++; //перемещаем указатель на следующий элемент
    }
}

```

ЗАМЕЧАНИЕ

Многие современные оптимизирующие компиляторы (например, транслятор C++ среды программирования Microsoft Visual Studio 2005) могут сами делать оптимизации такого рода. При этом в DEBUG-версии программы механизм оптимизации не включается, чтобы предоставить программисту возможность трассировки, зато в RELEASE-версии компилятор умеет применять очень хитрые оптимизации (гораздо хитрее тех, которые мы реализовали вручную).

Для двухмерного массива выражения `ar[0]`, `ar[1]` будут эквивалентны константным указателям на соответствующие строки, т. е. следующие выражения будут тождественны:

□ `ar[0] == &ar[0][0]`

□ `ar[1] == &ar[1][0]`

Пример.

Пусть требуется сформировать двухмерный массив $N \times M$, который будет содержать введенные пользователем строки (листинг 6.22). При этом предполагается, что пользователь не введет строку длиннее, чем M символов (т. к. предусматривать защиту от переполнения строки мы еще не умеем).

Листинг 6.22. Пример работы с двухмерным массивом

```
#include <iostream>

int main()
{
    const int N = 10, M = 10; //размерности массива
    char ar[N][M]; //зарезервировали место в памяти для ввода строк
    // подумайте, почему нельзя объявить массив для ввода таким
    // образом:
    //char* ar[N];

    //Вводим строки
    for( int i=0; i<N; i++)
    {
        //Замечание. Потоки ввода/вывода умеют работать с указателями типа char*
        // (по указанному адресу вводится вся строка, напечатанная
        // пользователем, и формируется завершающий ноль)
        std::cin>>ar[i]; //или std::cin>>&ar[i][0];
    }
    //Вывод сформированных строк
    for(int k = 0; k<N; k++)
    {std::cout<<ar[k]<< std::cout<<endl;}
}
```



6.3.3. Многомерные массивы

Расширим рассмотрение взаимосвязи массивов и указателей на случай многомерных массивов. При этом нам достаточно сделать еще один шаг — рассмотреть случай трехмерного массива, после чего читатель сможет самостоятельно (по индукции) перейти к массивам произвольной размерности.

Трехмерный массив можно представить как одномерный, каждым элементом которого является слой (т. е. двухмерный массив). Поэтому имя трехмерного массива компилятор интерпретирует как константный указатель на нулевой слой (нулевой двухмерный массив), равный начальному адресу области, занимаемой в памяти трехмерным массивом.

Пусть дан трехмерный массив:

```
const int N=2, M=3, K=4;
int ar[N][M][K] = {
    { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12},
      { -1, -2, -3, -4}, {-5, -6, -7, -8}, {-9, -10, -11, -12}
    };
```

Элементы этого массива также располагаются в памяти линейно и непрерывно (рис. 6.10).

Исходя из интерпретации компилятором имени трехмерного массива, введем эквивалентный указатель (указатель на слой):

```
int (*p)[3][4] = ar;
```

Теперь к `[i][j][k]`-му элементу массива будем одинаковым способом обращаться, используя имя массива `ar` или указатель `p`. При этом можно использовать любые комбинации операторов `[]` и `*` (главное, чтобы суммарно их было три):

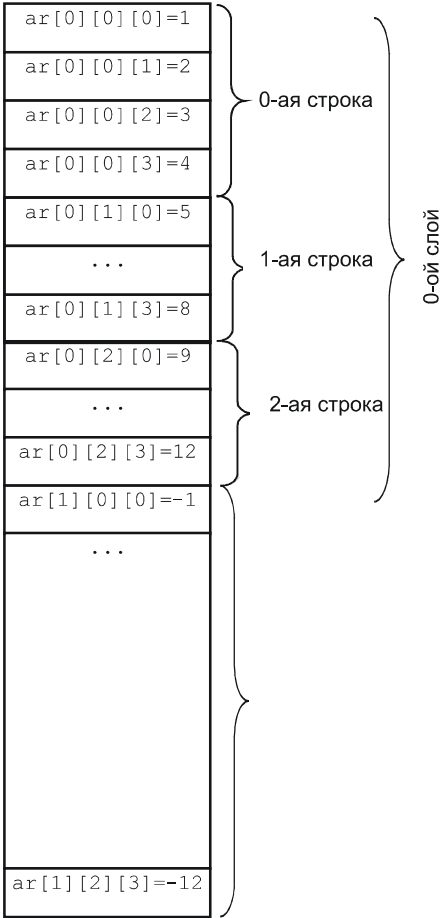
```
int tmp = ar[i][j][k];
tmp = *(*(*ar+i)+j)+k);
...
tmp = p[i][j][k];
...
```

ЗАМЕЧАНИЕ

По правилам арифметики указателей при вычислении выражения `p++`; компилятор переместит указатель на следующий слой массива! Тот же самый адрес он вычислит в выражении: `ar+1` (см. рис. 6.10).

Трехмерный массив 2*3*4
int ar[2][3][4];

Расположение в памяти



Представление программиста

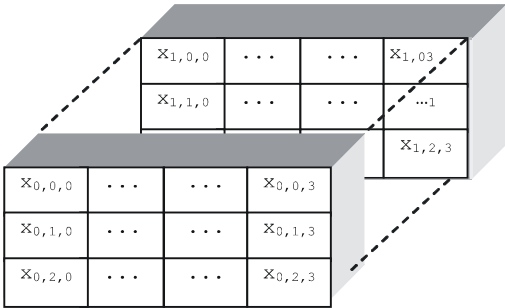


Рис. 6.10

Для вычисления адреса произвольного [i][j][k]-ого элемента трехмерного массива элементов типа Т компилятору нужно знать адрес начала массива, количество элементов в строке, количество строк в слое и размер элемента:

адрес_элемента_i_j_k = адрес_начала_массива + i * М * К * sizeof(T) + j * К * sizeof(T) + k * sizeof(T)

ОБРАТИТЕ ВНИМАНИЕ!

В вычислении адреса участвуют только младшие размерности массива, а старшая размерность (количество слоев) нигде не фигурирует!



Попробуйте самостоятельно оптимизировать задачу о суммировании элементов для трехмерного массива.

6.3.4. Массивы указателей

Элементы массива могут иметь любой тип (в т. ч. они могут быть указателями). Наиболее часто массивы указателей используются для хранения адресов строк текста, динамически создаваемых объектов и т. п. (см. разд. 6.4.4 и 9.10).

Массив указателей, как и любой другой массив, должен быть объявлен и может быть проинициализирован при определении.

Пример 1.

```
extern int* arn[][20]; //объявление внешнего двухмерного массива
                        указателей int*
```

Пример 2.

```
char* arc[5]; //определение одномерного массива из 5 указателей
              (компилятор резервирует место 5*sizeof(char*)
              байтов и, если массив глобальный, неявно
              инициализирует все элементы нулем)
```

Если массив локальный, то гораздо безопаснее (см. разд. 6.1.6) явно проинициализировать его элементы нулевым значением, поскольку нет ничего хуже, чем пользоваться указателем со случайным значением:

```
{
    char* arc[20]={0};
}
```

Сравним два внешне похожих выражения (рис. 6.11):

□ инициализация встроенного двухмерного массива строковыми литералами:

```
char ar1[][6]={ "One", "Two", "Three"};
```

Компилятор производит следующие действия:

- отводит в статической области памяти место для хранения строк;
- выделяет память $3 \cdot 6 \cdot \text{sizeof}(\text{char})$ для двухмерного массива `ar`;

- копирует содержимое строковых литералов в соответствующие строки массива;

❑ инициализация массива указателей строковыми литералами:

```
char* ar[]={ "One", "Two", "Three"};
```

Компилятор производит следующие действия:

- отводит в статической области памяти место для хранения строк;
- выделяет память под три указателя для одномерного массива ar;
- присваивает каждому элементу массива значение, равное адресу начала соответствующего строкового литерала (направляет каждый элемент массива на соответствующий строковый литерал).

Внутреннее представление двумерного массива и массива указателей при инициализации строковыми литералами

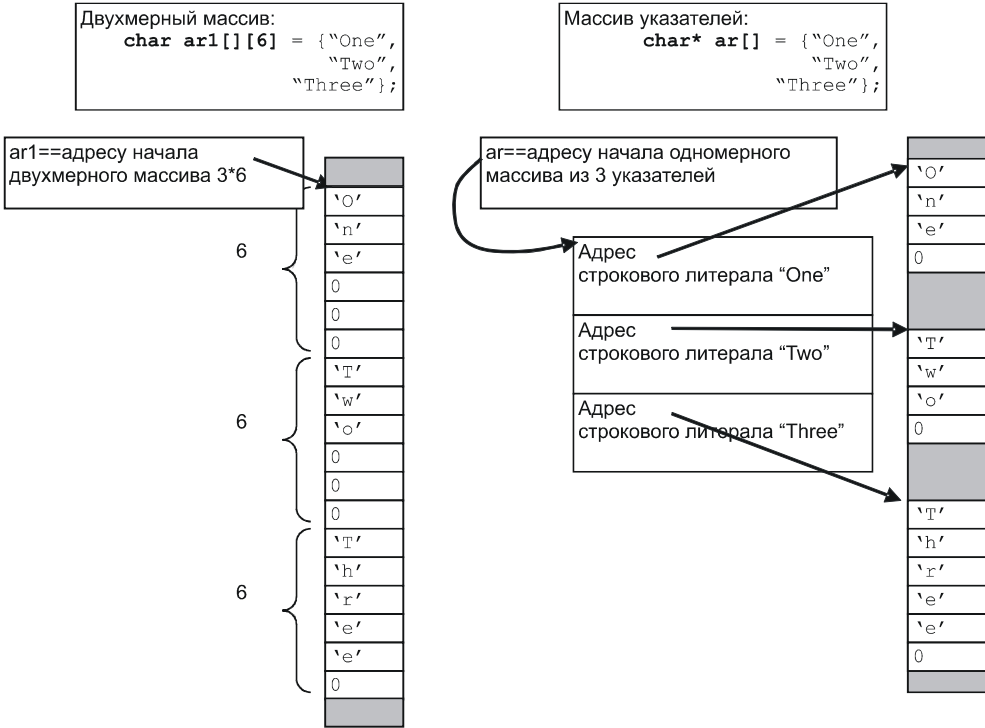


Рис. 6.11

ВАЖНО!

Поскольку строковые литералы компилятор располагает в области памяти (нередко защищенной от записи (см. разд. 3.2.4)), то попытка изменить их вызовет ошибку времени выполнения.

Например:

```
ar1[1][1] = 'D'; //ОК, заменили 'w' на 'D'
```

```
ar[1][1] = 'D'; //компилятор ошибок не выдаст, т. к. синтаксически  
                  выражение корректно, но зато произойдет ошибка  
                  во время выполнения (что гораздо хуже!)
```

РЕКОМЕНДАЦИЯ (Б. СТРАУСТРУП)

Предпочитайте ошибки компилятора ошибкам времени выполнения. Поэтому массив указателей на строковые литералы следует определять таким образом, чтобы компилятор не позволял модифицировать значения посредством указателей, хранящихся в массиве.

Например:

```
const char* ar[]={"One", "Two", "Three"};
```

```
//тогда компилятор не позволит модифицировать значение по адресу
```

```
//ar[1][1] = 'D'; //ошибка: нельзя модифицировать константную  
                  переменную
```

Приведенные примеры демонстрируют принципиальные различия:

- ❑ в расположении в памяти массива указателей и (на первый взгляд) подобного ему двухмерного массива;
- ❑ в использовании массивов (хотя синтаксически обращение к элементам обоих массивов выглядит одинаково!).

6.4. Динамические массивы

Нередко программист на этапе написания программы не может точно сказать, каков размер требуемых массивов — размер вычисляется в процессе выполнения программы. В таком случае возможны два варианта действий:

- ❑ зарезервировать память с запасом на этапе компиляции (при этом всегда существует вероятность того, что запаса может не хватить, или наоборот — неиспользуемый резерв окажется достаточно большим);
- ❑ применить механизм, позволяющий выделить память под массив уже во время выполнения программы в необходимом объеме. Этот механизм носит название — *динамическое управление памятью*.

Напомним, что существуют три вида и, соответственно, три способа использования памяти:

- ❑ статическая память, куда компилятор помещает на все время выполнения программы глобальные данные, заключенные программистом в пространства имен и объявленные с ключевым словом **static**;
- ❑ автоматическая память (в стеке), которую компилятор распределяет под локальные данные и автоматически освобождает ее при завершении функции (см. разд. 8.4);
- ❑ динамическая память (в куче), в которой требуемый блок захватывается только по явному запросу программиста (при вызове функции стандартной библиотеки типа `malloc()` или оператора языка C++ **new**) и освобождается программистом тоже явно (вызовом функции стандартной библиотеки `free()` или оператора **delete**).

ЗАМЕЧАНИЕ

Использование динамической памяти требует от программиста понимания этого механизма, целесообразного его использования и аккуратности. При некорректном манипулировании динамической памятью такая возможность оборачивается существенными неприятностями — ошибками времени выполнения и утечками памяти.

6.4.1. Управление памятью. Низкоуровневые функции языка Си

Понятия, связанные с функциями, подробно рассматриваются в гл. 8, а в данном разделе просто перечислены основные функции стандартной библиотеки, связанные с динамическим выделением памяти, и показан механизм выделения, использования и освобождения такой памяти.

Основные функции стандартной библиотеки для работы с динамической памятью объявлены в заголовочном файле `<stdlib.h>`:

- ❑ функция `malloc()` (или ее аналог `calloc()`, который не только выделяет память, но и обнуляет содержимое) выделяет указанное в качестве параметра количество байтов в динамической памяти (или куче — heap). Если память выделить удалось, функция возвращает ненулевой указатель (адрес начала выделенного блока), в случае же неудачи возвращает нулевое значение. Тип возвращаемого указателя — **void***. Пусть требуется выделить память под переменную `big` элементов типа `int` (листинг 6.23). Значение `big` формируется в процессе выполнения программы, поэтому встроенный массив с переменной размерностью компилятор создать не позволит:

Листинг 6.23. Захват динамической памяти низкоуровневыми средствами Си

```
{
    unsigned int BIG; //здесь будет количество элементов
    //вычисление значения BIG (может быть очень большим!)
    //int ar[BIG]; //ошибка: размерность должна быть известна на этапе
                    //компиляции
    size_t n = BIG * sizeof(int); //число байтов, которые
                                //требуется выделить динамически
    int* p = static_cast<int*>(malloc(n)); //возвращаемый функцией
                    void*-указатель приводим к требуемому типу
    //если память выделить не удалось, возвращается нулевое значение
    if(p)
    { //память выделить удалось, можно пользоваться указателем,
      //например:
      for(int i=0; i< BIG; i++)
      {
          p[i] = 1;
      }
    }else
    {
        //возможно, предпринимаем другие действия
    }
}
```

СУЩЕСТВЕННО!

На самом деле выделяется не столько байтов, сколько запросил программист, а больше, т. к. для каждого запрошенного программой блока памяти формируется дополнительная служебная информация (рис. 6.12). Размер и формат служебного блока могут зависеть от конкретного компилятора, поэтому данную информацию программист пользоваться не может. Такую информацию компилятор формирует при динамическом выделении памяти и использует в первую очередь при освобождении захваченного блока;

- ❑ в некоторых реализациях можно получить размер динамически выделенного блока памяти с помощью функции `_msize()`:
`size_t num = _msize(p);`
- ❑ если объем блока нужно изменить (обычно увеличить), можно использовать функцию `realloc()` (листинг 6.24). При необходимости функция выделяет новый блок памяти, содержимое старого блока переписывает в новый, а старый освобождает. В случае успеха возвращает ненулевой указатель;

Что происходит при динамическом выделении памяти



Рис. 6.12

Листинг 6.24. Перераспределение динамического блока памяти

```
{  
    int* p = static_cast<int*>(malloc(1000));  
    if(p)  
    {  
        //использование p  
        //потребовалось большее количество памяти  
        p = static_cast<int*>(realloc(p,2000));  
        if(p){если удалось захватить новый блок памяти}  
    }  
}
```

❑ функция `free()` освобождает память. При этом блок, на который указывает `p`, исключается из списка занятых и подключается к списку свободных блоков:

```
free(p);
```

ЗАМЕЧАНИЕ

Если программист не аккуратен и забывает освобождать захваченную память, он порождает утечки памяти. Это означает, что при выполнении программы образуются области памяти, которые не используются, но будут считаться занятыми вплоть до завершения программы.

6.4.2. Управление памятью. Операторы C++ *new* и *delete*

Операторы **new** и **delete** являются надстройкой языка C++ над низкоуровневыми функциями `malloc()` и `free()`. Эти операторы являются *встроенными* средствами языка C++, поэтому никаких заголовочных файлов подключать не нужно!

Если вы программируете на C++, то предпочтительнее использовать операторы **new** и **delete** вместо функций `malloc()` и `free()`, потому что:

- ❑ не обязательно явно указывать количество требуемых байтов (компилятор может подсчитать сам по контексту вызова оператора **new**);
- ❑ не нужно явно приводить тип возвращаемого значения (**void***) к требуемому типу (компилятор неявно делает это сам);
- ❑ основное преимущество использования оператора **new** проявляется для объектов пользовательского типа (классов C++), т. к. оператор совмещает выделение памяти с вызовом специальной инициализирующей функции (автоматически вызывает для таких объектов конструктор). Соответственно основным преимуществом использования оператора **delete** является автоматический вызов специальной функции деинициализации — деструктора для объектов классов.

Существуют несколько форм операторов **new** и **delete**. Обычная (самая распространенная) форма использования на самом деле означает вызов функций стандартной библиотеки:

```
void* operator new(size_t); //эта функция, в свою очередь, вызывает
                             низкоуровневую функцию malloc(), которая
                             и резервирует память. Согласно стандарту C++ эта
                             форма оператора new предназначена для выделения
                             памяти под одиночный объект
```

```
void operator delete(void *); //а внутри этой функции вызывается
                               низкоуровневая функция free(). Согласно стандарту C++
                               эта форма оператора delete предназначена для
                               освобождения памяти, занятой одиночным объектом
```


В стандартизованном C++ появились еще несколько форм операторов **new** и **delete**. В частности — в силу специфики использования классов, разработчики ввели специальные версии **new** и **delete** для массивов:

```
void* operator new[](size_t); //выделение памяти для массива
```

```
void operator delete[](void *); //освобождение памяти, занятой массивом
```

ЗАМЕЧАНИЕ

Пока программист не пользуется классами, принципиальной разницы в использовании этих двух вариантов **new/delete** нет. Несмотря на это, стоит выработать полезную привычку использовать **new/delete** для одиночных динамических объектов и **new[]/delete[]** для массивов объектов.

Использование оператора **new** для выделения памяти

Динамически можно выделить память для любого объекта или массива объектов. Например, для того, чтобы зарезервировать память в куче для одиночного значения типа **int**:

```
int* p = new int; //размер выделяемого блока задается неявно.
```

Оператор **new**, исходя из указанного типа, выделяет `sizeof(int)` байтов в отличие от `malloc(4)`, где количество байтов нужно указывать явно

```
int* p = operator new(sizeof(int)); //размер задается явно
```

ЗАМЕЧАНИЕ

В результате успешного выделения памяти возвращается ненулевой указатель, содержащий адрес выделенного блока памяти, а в случае нехватки памяти (согласно стандарту C++) нулевое значение (как в случае с функцией `malloc()`) не формируется, а генерируется исключение, поэтому проверять возвращаемое значение на ноль смысла не имеет. Механизм исключений в рамках данной книги не обсуждается, а вам пока просто не следует задумываться над специальной программной защитой в случае нехватки памяти — она предусмотрена в реализации оператора **new**.

Программисту следует отдавать себе отчет в том, что использование динамической памяти — средство достаточно дорогое, поэтому прибегать к нему следует лишь тогда, когда оно действительно необходимо. Очевидно, что накладные расходы при динамическом выделении памяти достаточно большие, поскольку:

- для небольших объектов дополнительная служебная информация может более чем вдвое увеличить количество памяти для каждого выделяемого блока;
- в процессе выполнения программы на динамическое выделение памяти тратится дополнительное время (на формирование служебных структур данных);

- неочевидным следствием использования динамического выделения памяти является фрагментация кучи. То есть при выполнении программы может возникнуть такая ситуация, когда суммарно запрашиваемый объем памяти имеется, но эта свободная память разрезана на кусочки, которые компилятор склеивать не умеет!

Использование оператора *delete* для освобождения памяти

Выделив динамически память, программист несет ответственность за ее освобождение, поэтому, попользовавшись, не забудьте освободить ее (память) посредством оператора **delete**:

```
delete p; //для освобождения блока памяти, адрес которого содержится
           в указателе p, компилятор использует информацию,
           сформированную в служебном блоке
p=0; //после освобождения памяти в переменной p еще сохраняется
      значение адреса уже недействительного блока, поэтому
      безопаснее такой указатель обнулить!
```

ЗАМЕЧАНИЕ

Попытка выделить 0 байтов выполняется корректно (в большинстве реализаций резервируется один байт).

ПРЕДОСТЕРЕЖЕНИЕ

Будьте бдительны при использовании указателя на динамически захваченный блок памяти! Пример трудно выявляемого дефекта программы приведен в листинге 6.25.

Листинг 6.25. Пример небрежного использования программистом указателя на динамически выделенный блок памяти

```
{
char* p=new char; //динамически выделили один байт
*p='A'; //занесли по этому адресу значение
p="QWERTY"; //перенаправили указатель на строковый литерал,
            т. е. занесли в переменную p адрес совершенно другой
            области памяти. При этом предыдущее значение адреса
            затерли — теперь до завершения программы этот блок
            никто не освободит (память потекла)!

delete p; //а потом вспомнили, что когда-то вызывали оператор new
            и неплохо бы вызвать соответствующий delete, но при
```

выполнении этой строки скорее всего получили ошибку времени выполнения.

}



Попробуйте догадаться, почему во время выполнения возникает ошибка и как следует переписать этот листинг?

6.4.3. Сборщик мусора (*garbage collector*)

Если неаккуратный программист выделил динамически блок памяти, но забыл его освободить, то эта память будет освобождена операционной системой только после завершения приложения. Но если такой программист выделяет память интенсивно и достаточно большими блоками, то даже виртуальная память может закончиться...

В ряде систем существует механизм автоматического освобождения памяти (сборщик мусора). В таких случаях система самостоятельно определяет, какая область памяти больше не используется программой, и освобождает ее. Такой подход удобен для забывчивых программистов, но имеет свои недостатки. Например, он совершенно не годится для создания критичных ко времени программ (деятельность сборщика мусора может требовать немало времени и не синхронизирована с выполнением пользовательской программы). Поэтому в C/C++ сборка мусора не реализована (создатели языка в первую очередь преследовали цель эффективности). А это означает, что за программиста его обязанности по освобождению неиспользуемой динамической памяти никто выполнять не будет!

6.4.4. Операторы *new[]* и *delete[]* и массивы

Создавать массивы динамически стоит в тех ситуациях, когда:

- ☐ все или некоторые размерности массива вычисляются только во время выполнения программы;
- ☐ размерности массива могут изменяться в процессе выполнения программы.

ЗАМЕЧАНИЕ

Если вы на каком-либо этапе знаете размеры массива и можете сообщить их компилятору, не стоит динамически выделять память. Пользуйтесь встроенными массивами — это экономит время и память!

Рассмотрим примеры создания динамических массивов.

Пример 1.

Одномерные динамические массивы (с точностью до того, в какой области выделяется память) очень похожи на встроенные (память выделяется монолитно), но в этом случае программист сам должен заботиться об освобождении памяти (листинг 6.26).

Листинг 6.26. Создание одномерного динамического массива

```
{
    unsigned int n; //здесь будет количество элементов
        //вычислили n
    //int ar[n]; //ошибка: размерность массива должна быть константой
    int* pn=new int[n]; //динамически выделяем требуемый блок памяти
        (по контексту вызова оператора new компилятор догадается,
        что требуется выделить n*sizeof(int) байтов)
    //Использование динамического массива:
    pn[i]=...;
    //Пусть в процессе выполнения программы оказалось, что требуется
        увеличить размерность массива вдвое:
    int* tmp = new int[n*2]; //выделили новую память

    //переписали старое содержимое в новую область:
//а)
    for(int i = 0; i<n; i++){ tmp[i] = pn[i];}
//б) или более эффективный вариант копирования блока байтов из одной
    области памяти в другую посредством функции стандартной
    библиотеки memcopy():
    memcopy(tmp,pn,n*sizeof(int));

    delete[] pn; //старую область памяти освободили
    pn = tmp; // перенаправили указатель pn на новую область памяти
    //Продолжаем пользоваться динамическим массивом посредством
        указателя pn:
    pn[i]=...;

    //Когда массив больше не нужен, память следует освободить
    delete[] pn; //освобождение динамической памяти. После
```

освобождения памяти этим адресом пользоваться
нельзя!

```
pn=0; //безопаснее обнулить указатель, чем оставить в нем значение  
      недействительного адреса
```

```
}
```

Пример 2.

Многомерные динамические массивы.

В зависимости от того, как задаются размерности, с помощью оператора **new** можно создавать массивы по-разному. При этом существенно различаются способы задания и размещение массивов в памяти!

Способ первый.

Все младшие размерности известны на этапе компиляции (константы) (листинг 6.27). Как и в случае одномерного массива, такой динамический массив очень похож на встроенный (отличия: место расположения heap, необходимость явного освобождения памяти).

ЗАМЕЧАНИЕ

Обратите внимание на объявление указателя — при любом другом объявлении указателя компилятор выдаст ошибку.

Листинг 6.27. Создание трехмерного динамического массива с известными на этапе компиляции младшими размерностями

```
{  
    float (*pf)[25][10]; //определение указателя на двухмерный  
                          массив 25*10. Это эквивалент имени  
                          трехмерного массива.  
    unsigned int n = выражение; //вычислили старшую размерность  
    pf = new float[n][25][10]; //динамически выделили блок памяти.  
    Указателем pf можно пользоваться так же, как и  
    именем трехмерного массива  
    //Использование:  
    pf[i][j][k] = ...;  
    delete[] pf; //не забыли освободить память  
}
```

ЗАМЕЧАНИЕ 1

При определении указателя `pf` компилятор выделяет память только под переменную для хранения адреса. Посредством указания младших размерностей мы даем указание компилятору, каким образом интерпретировать этот адрес. Несмотря на то, что память для самого массива отводится динамически, сам указатель (в зависимости от контекста определения) может быть локальным, глобальным, динамическим.

ЗАМЕЧАНИЕ 2

При динамическом выделении памяти все размерности массива, кроме самой левой, должны быть константными выражениями, которые при вычислении дают положительное целое.

Например:

```
int n, m, k;
//вычисляются значения n, m, k
float (*pf)[25][10];
pf = new float[n+m+k][25][10];
```

Способ второй.

Часто возникает потребность работать с многомерными массивами, все размерности которых априори неизвестны. Пусть требуется работать с двумерным массивом, размерности которого N и M вычисляются в процессе выполнения программы. В этом случае у программиста есть две возможности.

Возможность первая — создать одномерный динамический массив элементов требуемого типа T с размерностью $(N \cdot M)$.

```
int N, M;
//вычисление значений N и M
int* p = new int[N*M]; //выделение памяти для одномерного массива
```

В результате программист получает в свое распоряжение указатель, который эквивалентен имени одномерного массива. А ему (программисту) хочется интерпретировать массив как двумерный, но тогда он должен сам каждый раз формировать адрес $[i][j]$ -го элемента таким же образом, как это сделал бы компилятор для встроенного двумерного массива:

```
p[i * M + j]
```

Программист может облегчить себе обращение к $[i][j]$ -му элементу, создав вспомогательный массив указателей на строки (рис. 6.13):

```
int** pp = new int*[N]; //вспомогательный указатель, посредством
которого можно будет обращаться к элементу привычным
```

```
        для двумерного массива способом (посредством двух
        индексов)
for(int i=0; i<N; i++)
{
    pp[i] = p + i*M; //направляем каждый i-й указатель на
                     начало соответствующей i-й строки
}
```

Создание вспомогательного массива указателей на строки для интерпретации одномерного массива как двумерного

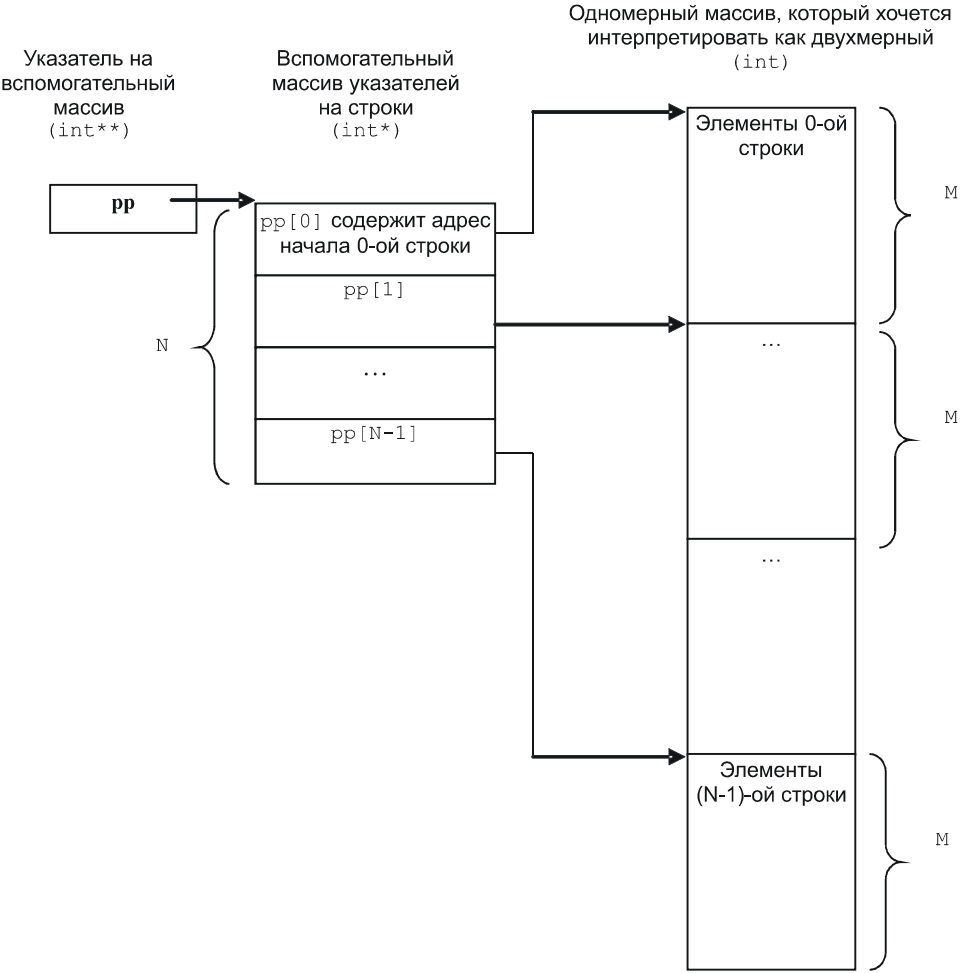


Рис. 6.13

Теперь вспомогательным указателем `pp` можно пользоваться (синтаксически) точно так же, как именем двумерного массива (хотя компилятор будет вычислять адрес по-другому, исходя из типа указателя `pp`):

```
pp[i][j]
```

Этот способ хорош в том случае, когда размерности массива в процессе выполнения программы не изменяются. Очевидным плюсом такого подхода является минимизация накладных расходов при динамическом выделении памяти — два блока служебной информации (для собственно массива значений и для вспомогательного массива указателей).

Но! Подумайте, что придется сделать в том случае, когда количество строк в таком массиве будет уменьшаться или увеличиваться во время выполнения программы. Например, требуется добавить в массив еще одну строку (листинг 6.28).

Листинг 6.28. Действия программиста при увеличении количества строк в динамическом двумерном массиве

```
{
    N+=1; //увеличили количество строк на 1
    int* tmp = new int[N*M]; //выделили новый блок памяти
    for(int i=0; i< (N-1)*M; i++)
    {tmp[i] = p[i];} //переписали в новый блок старое содержимое
    //Теперь старый массив нам не нужен
    delete[] p; //освободили старый массив
    p = tmp; //перенаправили указатель на новый массив
    delete[] pp; //освободили память, занятую вспомогательным массивом
                    указателей
    pp = new int*[N]; //выделить для вспомогательного массива новый
                    блок (он больше, чем предыдущий)
    for(int i=0; i<N; i++) //сформировать во вспомогательном массиве
                    новые адреса
    {
        pp[i] = p + i*M;
    }
    //Продолжаем пользоваться указателем pp
    ...
    //Массив больше не нужен. Память нужно освободить
    delete[] p;
    delete[] pp;
}
```


Возможность вторую можно использовать в таких задачах, где размерности массива могут изменяться во время выполнения. При таком подходе в памяти создается довольно сложная структура (рис. 6.14) со всеми, присущими динамическому выделению памяти накладными расходами, зато минимизируются действия при изменении размерностей массива во время выполнения.

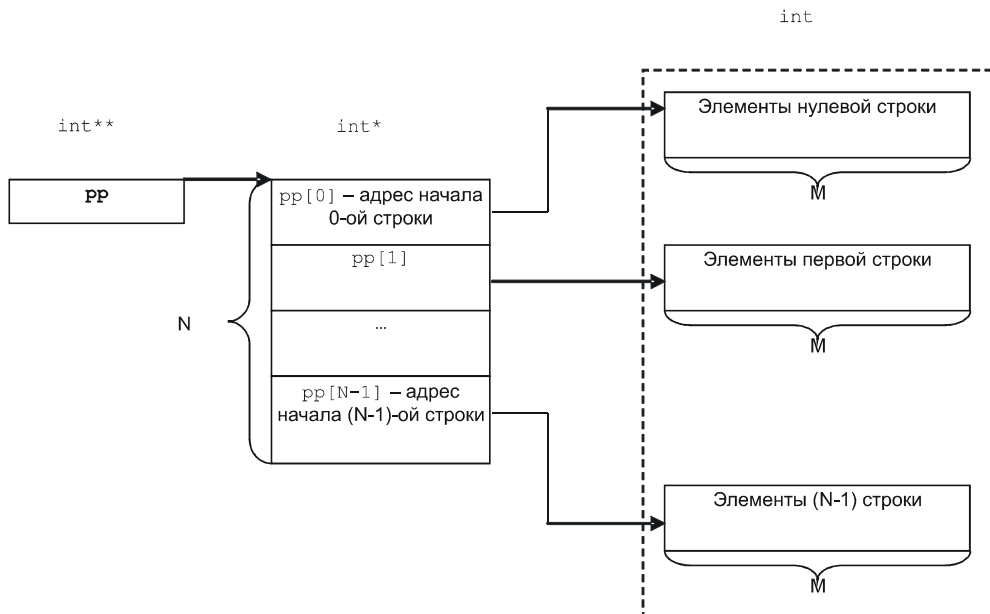


Рис. 6.14

Последовательность действий при создании такой сложной структуры данных в памяти представлена в листинге 6.29.

Листинг 6.29. Создание динамического двухмерного массива

[illegible]

```

                                указатель запоминается в
                                соответствующем элементе
                                массива указателей
}
//Для доступа к элементу пользуемся указателем p и двумя индексами
for(int i = 0; i<N; i++)
    for(int j = 0; i<M; j++)
    {
        p[i][j] = i+j; //например, сформируем значения
                        элементов
    }
}

```

Обращение к элементу такого массива на уровне языка C/C++ синтаксически выглядит так же, как и в случае обычного встроенного массива, но следует иметь в виду, что на низком уровне компилятор (исходя из типа указателя) вычисляет адрес принципиально по-другому.

```

int x = p[i][j]; //компилятор сначала извлекает адрес i-той строки из
                массива указателей, а потом к этой базе добавляет
                смещение j-ого элемента относительно
                начала i-той строки

```

Накладных расходов при этом подходе очень много (служебный блок информации на каждый выделенный блок памяти), зато для такого массива легко увеличивать или уменьшать количество строк. Рассмотрим, как при такой организации массива можно добавить одну строку (листинг 6.30).

Листинг 6.30. Действия программиста при увеличении количества строк в динамическом двумерном массиве

```

{
    N+=1;
    int** tmp = new int*[N]; //выделили новый блок памяти только под
                            массив указателей на строки (сами
                            строки трогать не будем!)
    for(int i=0; i<(N-1); i++)
    {
        tmp[i] = p[i]; //просто переписали в новый массив адреса строк
    }
    delete[] p; //теперь старый массив указателей нам не нужен
                - освобождаем память
}

```

```

p=tmp; //перенаправили наш указатель p на новый массив
p[N-1] = new int [M]; //динамически выделили память для новой
                        добавляемой строки
//продолжаем пользоваться указателем p
...
//Поработав с динамическим массивом, аккуратный программист
    должен корректно освободить всю захваченную память
for(i=0; i<N; i++)
{
    delete[] p[i]; //освобождаем память, занятую i-той строкой
}
delete[] p; //освобождаем память, занятую массивом указателей
            на строки
p = 0; //теперь этим адресом пользоваться нельзя, поэтому для
        безопасности обнуляем
}

```

ЗАМЕЧАНИЕ

При таком создании динамического массива программист должен понимать, что строки массива расположены в разных участках heap, а не занимают непрерывную, линейную область памяти! Поэтому оптимизации, применимые к встроеным массивам (листинг 6.21), невозможны.

Различие операторов *delete* и *delete[]*

Различие в использовании двух форм **delete** и **delete[]** проявляется только тогда, когда речь идет о динамическом создании объектов пользовательского типа (классов). Проблема состоит в том, что для таких объектов перед освобождением памяти компилятор должен автоматически вызвать специальную функцию (деструктор). А т. к. оператор **delete** получает в качестве параметра только указатель, то необходимо предоставить ему возможность различать — является ли данный указатель указателем на одиночный объект (компилятор должен вызвать деструктор для одного объекта) или на массив объектов (деструктор должен быть вызван для каждого элемента массива).

Для этих целей в стандартизованном C++ существуют две формы оператора **delete**, которые не рекомендуется смешивать:

□ **delete** — для одиночного объекта;

□ **delete[]** — для массива.

//Пусть под "Т" имеется в виду любой тип

Т* p = new Т; //выделена память под одиночный объект типа Т

```
T* pAr = new T[100]; //выделена память под массив из 100 элементов
                        типа T
//работаем с обоими указателями
delete p; //корректно
p=0; //рекомендуется
//delete[] p; //результат не определен!
delete[] pAr; //корректно
pAr = 0; //рекомендуется
//delete pAr; //память будет гарантированно освобождена, но
                если под T имеется в виду пользовательский
                тип (class), то деструктор будет вызван
                только для нулевого элемента массива
```

ПРАВИЛО

Если вы применяете **new[]**, то должны использовать **delete[]** и наоборот!

6.4.5. Инициализация динамических массивов

Оператор **new** не позволяет инициализировать динамически выделенные массивы, поэтому программист должен позаботиться о том, чтобы элементы массива не имели случайных значений. Некоторые компиляторы в **DEBUG**-версии программы заполняют свободные и недействительные (освобожденные) области кучи специальными значениями, которые позволяют в **DEBUG**-версии отследить ошибки обращения к таким областям памяти.



Глава 7

Ссылки

7.1. Понятие ссылки

Ссылки появились в языке C++ (в языке Си такого понятия не было).

Ссылки — это еще одно средство, помимо указателей, позволяющее программисту манипулировать адресами программных объектов. Зачем понадобилась новая языковая конструкция, указывающая компилятору, что переменная содержит адрес объекта?

Рискну сделать два предположения:

- ❑ если при процедурном подходе без ссылок можно прекрасно обойтись, пользуясь родным для языка Си указателем, то при объектно-ориентированном подходе (при программировании в классах) некоторые выражения при использовании ссылок выглядят привычнее (тема объектно-ориентированного программирования выходит за рамки данной книги);
- ❑ с целью облегчения адаптации к языку C++ программистов, привыкших к другим высокоуровневым языкам программирования, в которых такого понятия, как указатель, не существует. Несмотря на то, что переменная-ссылка содержит адрес объекта, программист пользуется ссылкой синтаксически так же, как если бы он пользовался самим объектом, а компилятор (зная о том, что в переменной содержится адрес требуемого значения) неявно обращается к содержимому по этому адресу. Таким образом, можно сказать, что программисту посредством ссылки предоставляется псевдоним объекта.

Разницу в реализации указателей и ссылок иллюстрирует рис. 7.1.

Как бы там ни было (ссылки могут нравиться программисту, или он постарается обойтись без них), но ссылки можно воспринимать как удобную альтернативу указателям. Программисту они могут казаться чуждыми языку Си

и искусственными, но, тем не менее, при программировании на C++ пользоваться ими он должен уметь. Более того, в объектно-ориентированной части стандартной библиотеки для различных целей сплошь и рядом используются именно ссылки.

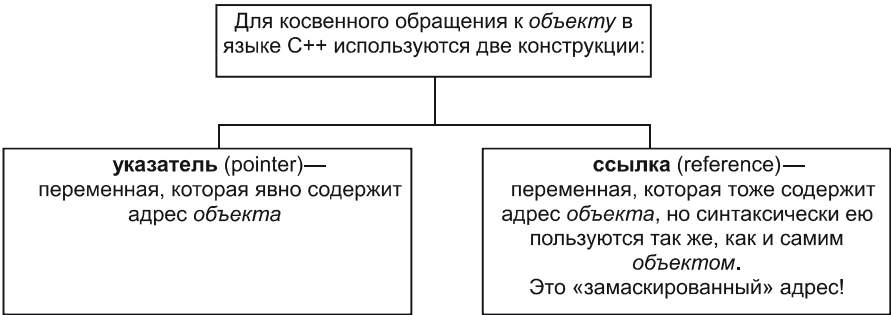


Рис. 7.1

ВАЖНОЕ ЗАМЕЧАНИЕ

Несмотря на то, что и переменная-указатель, и переменная-ссылка содержат адрес объекта, указатели и ссылки подчиняются разным синтаксическим правилам, поэтому эти две конструкции программист использует по-разному!

7.2. Сравнение ссылок и указателей

Специфические особенности синтаксиса и использования ссылок (по сравнению с указателями) представлены в табл. 7.1.

Таблица 7.1. Сравнительная характеристика указателя и ссылки

В чем проявляется отличие	Указатель	Ссылка
Тип переменной	T*	T&
Объявление и определение	int x=1;	
	int* p = &x;	int& r = x; с этого момента ссылка r становится псевдонимом x

Таблица 7.1 (продолжение)

В чем проявляется отличие	Указатель	Ссылка
Инициализация	Инициализация рекомендуется, но не обязательна: <code>int* p; //OK</code>	Инициализация обязательна! <code>int& r;</code> ошибка: нет инициализатора <code>extern int& r1; //OK</code> внешняя ссылка, должна быть проинициализирована в другом файле при определении
Получение значения. Ссылка (как и указатель) содержит адрес объекта, но синтаксически именем ссылки можно (и нужно) пользоваться так же, как именем самого объекта. При этом говорят, что переменная-ссылка предоставляет альтернативное имя для объекта	<code>int tmp = *p;</code> получили значение переменной x <code>(*p)++;</code> значение переменной x изменилось на 1 <code>p++;</code> значение адреса изменилось на <code>sizeof(int)</code>	<code>int tmp = r;</code> получили значение переменной x <code>r++;</code> значение переменной x изменилось на 1 Ссылку можно интерпретировать как константный указатель, при каждом использовании которого автоматически происходит разыменование
Модификация адреса	Если обычный указатель не объявлен как константный, то его значение можно изменять. <code>p++; //изменение адреса</code> <code>int y=2;</code> <code>p = &y;</code> перенаправление указателя (переменной p присваивается адрес y)	Ссылка тоже содержит адрес, но этот адрес формируется один раз при инициализации и изменить его невозможно! <code>r++; //изменение значения по адресу</code> <code>int y=2;</code> <code>r=y;</code> переменной x (адрес которой содержится в ссылке r) будет присвоено значение переменной y
Нулевое значение	Указатель может быть равен нулю (это значение говорит о том, что указатель никуда не указывает)	Ссылка всегда содержит адрес того объекта, которым она была проинициализирована

Таблица 7.1 (продолжение)

В чем проявляется отличие	Указатель	Ссылка
Применение оператора &. Чтобы получить указатель на объект, псевдонимом которого является ссылка r, можно применить к ссылке операцию получения адреса объекта — &	Адрес указателя <pre>int** pp = &p;</pre> применение оператора & к указателю позволяет сформировать адрес этого указателя <pre>int y = **pp;</pre> для того, чтобы получить значение, указатель pp нужно дважды разыменовать	Адрес ссылки <pre>int* pr = &r;</pre> применение оператора & к ссылке позволяет сформировать адрес объекта, псевдонимом которого является ссылка <pre>int y = *pr;</pre> для того, чтобы получить значение, указатель pr нужно просто разыменовать
Конструкция "Адрес адреса"	Конструкция типа "указатель на указатель" (T**) допустима и используется достаточно часто при создании сложных структур данных (см. разд. 6.4.4). <pre>int** pp = &p;</pre>	Понятия "ссылка на ссылку" (T&&) не существует (при попытке такого объявления компилятор выдаст ошибку): <pre>//int&& rr = r;</pre> //ошибка
Ссылка на указатель		Ссылка на указатель выглядит непривычно, но довольно часто именно такая конструкция используется для возвращения адреса переменной-указателя из функции. Поскольку функции рассматриваются в гл. 8, то пока просто покажем механизм использования такой конструкции: <pre>int n;</pre> <pre>int* p = &n;</pre> <pre>int*& refP = p;</pre> теперь можно использовать refP вместо p <pre>*refP = 2;</pre> переменной n будет присвоено значение 2 <pre>//int*& refM =&n;</pre> ошибка: справа не переменная-указатель, а выражение для вычисления адреса (для результата такого выражения псевдоним ввести невозможно)

Таблица 7.1 (продолжение)

В чем проявляется отличие	Указатель	Ссылка
Ключевое слово void	Можно объявить универсальный указатель типа void* void* p; //OK	Ссылку типа void& объявить невозможно, т. к. ссылка — это всегда псевдоним совершенно определенного объекта. //void& r = x; //ошибка
Оператор sizeof	double* p; size_t n1 = sizeof (p); //4 байта size_t n2 = sizeof (*p); //8 бай- тов	double d; double& rd = d; size_t n = sizeof (rd); //8 байтов
Инициализация литералом. Константная ссылка	//int* p = 0x100000000; //ошибка В стиле Си: int* p = (int*) 0x100000000; //OK В стиле C++: int* p = reinterpret_cast<int*> (0x100000000); //OK	//int& r = 1; //ошибка const int& r=1; //OK! При этом компилятор выделяет память (в нашем случае sizeof(int)), заносит туда единицу, а адрес такой фиктивной переменной сохраняет в ссылке r. Такой прием в данном контексте особого смысла не имеет, но весьма актуален при передаче параметров функции (см. разд. 8.6.2)

ЗАМЕЧАНИЕ

Применение ссылок так, как это показано в последнем пункте, весьма экзотично (на мой взгляд), а в основном ссылки используются в качестве параметров функции и возвращаемых функцией значений (см. разд. 8.6.2 и 8.7.2).



Глава 8

Функции

8.1. Понятия, связанные с функциями

Часто встречается ситуация, когда одно и то же действие необходимо выполнить в разных местах программы, возможно, с разными наборами данных. Нецелесообразно в каждом таком месте приводить одну и ту же последовательность инструкций (один и тот же текст). Нередко оказывается, что необходимый фрагмент программы уже реализован другими программистами, в частности — разработчиками библиотек. И в том, и в другом случае нужно уметь один раз оформлять такие фрагменты специальным образом, а пользоваться ими многократно. Для того чтобы использовать один и тот же код, требуется обеспечить следующие действия:

1. Сформировать исходные данные (параметры) для такого фрагмента.
2. Передать управление на начало фрагмента (вызвать его).
3. По окончании выполнения фрагмента получить управление и результат работы назад.

Механизм функций языка C/C++ помогает программисту обеспечить все перечисленные шаги.

Функции — это кирпичики, из которых строится программа (см. *разд. 1.2.2*). Они дают программисту возможность заключить часть кода в черный ящик, который в дальнейшем можно использовать многократно, не интересуясь его внутренним содержанием. Функции принимают параметры, выполняют последовательность инструкций (называемую телом функции), возможно, формируют результат, а затем возвращают управление (рис. 8.1). Результат, формируемый функцией, представляет собой единственную (в простейшем случае скалярную) величину.

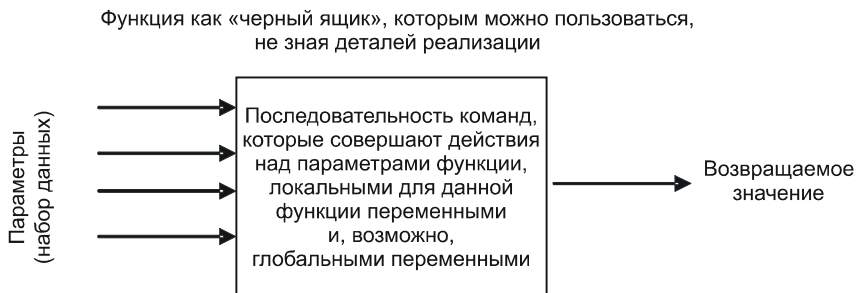


Рис. 8.1

Использование функций позволяет:

- ❑ не дублировать многократно код, который выполняет одни и те же действия с разными наборами данных;
- ❑ использовать посредством функций чужой код (в частности, возможности стандартной библиотеки);
- ❑ улучшать структуру программы (путем выделения в отдельную сущность некоторого законченного действия);
- ❑ уменьшать сложность восприятия больших программ, а при разработке больших проектов или библиотек функции позволяют группе разработчиков относительно независимо трудиться над своими частями разработки.

Для грамотного использования функций программисту нужно вспомнить про такие понятия, как объявление и определение (см. разд. 3.5). Как и в случае переменных, на момент использования (вызова) функции компилятор должен знать ее свойства. Обычно с функциями связаны три понятия (рис. 8.2): объявление, определение и вызов. Пользуясь клиент-серверной терминологией, можно рассматривать функцию как сервер, предоставляющий свои сервисы любому клиенту. Согласно правилам структурного программирования сервер предоставляет свои услуги клиентам в двух частях: файл реализации (.cpp), где находится тело функции, и файл интерфейса (.h), в который сервер помещает описание свойств экспортируемой функции. При таком подходе любому клиенту достаточно подключить заголовочный файл с объявлением функции, чтобы на момент вызова этой функции компилятор знал ее свойства и мог сгенерировать низкоуровневый код, обеспечивающий вызов.

ЗАМЕЧАНИЕ

Только в единственном случае, когда определение функции и вызов этой функции находятся в одном файле, а определение предшествует вызову (рис. 8.3), объявление функции можно опустить.

Понятия, связанные с обычными (не встроенными) функциями

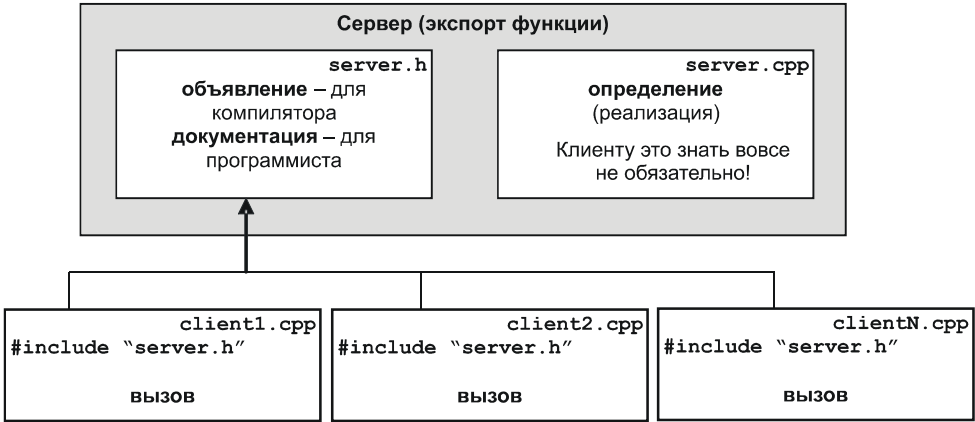


Рис. 8.2

```
1.cpp

//Определение функции f1
void f1()
{
    ...
}

void f2()
{
    вызов функции f1
}
```

Рис. 8.3

Обычно сервер, экспортирующий функцию, разделяет понятия на интерфейс и реализацию (см. разд. 5.6.1) и помещает объявление функции в заголовочный файл, а собственно программный код функции — в исходный .cpp-файл. Исключением является оформление встроенной или inline-функции (см. разд. 8.1.4). Для встроенной функции и объявление, и определение принято помещать в заголовочный файл. Специфика понятий, связанных с inline-функциями, представлена на рис. 8.4.

Специфика использования встроенных (inline) функций

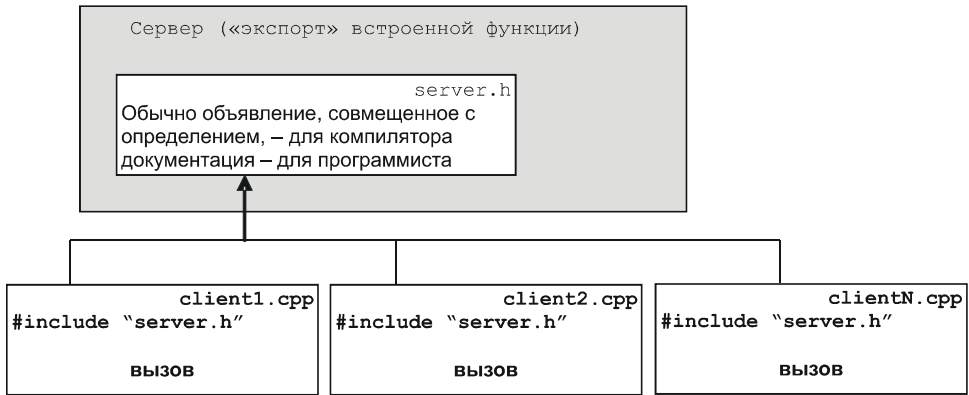


Рис. 8.4

8.1.1. Объявление (прототип) функции

Объявление функции — это предварительное описание, которое извещает компилятор о типе возвращаемого значения, количестве и типах передаваемых аргументов и других свойствах функции. Используя прототип, компилятор может выполнить контроль числа аргументов и проверить соответствие их типов при вызове функции (а при необходимости сделать неявное преобразование типа) и сгенерировать низкоуровневую последовательность команд для вызова.

Синтаксис:

```
[спецификатор] [тип] [соглашение по вызову] имя_функции(  
    [список_аргументов] || [void]);
```

Здесь:

- ❑ **тип** — задает тип возвращаемого функцией значения. Если поле отсутствует, то по умолчанию компилятор считает, что функция должна возвращать тип **int**. Рекомендация: не экономьте на умолчании — хорошим стилем программирования считается указание типа возвращаемого значения всегда! Некоторые компиляторы при отсутствии типа возвращаемого значения просто выдают ошибку. Если поле содержит ключевое слово **void**, функция не возвращает никакого значения. Так как на формирование возвращаемого значения тратится несколько машинных инструкций, то использование **void** в тех случаях, когда в возвращаемом значении нет необходимости, позволяет получить более короткий и производительный код.

Примеры:

```
char MyFunc(); //функция возвращает значение типа char
char* MyFunc(); //функция возвращает значение типа указатель на char
void MyFunc(); //функция не возвращает никакого значения
MyFunc();      //возвращаемый тип - int по умолчанию.
               //Замечание: некоторые компиляторы (VC 2005) такое умолчание
               не поддерживают!
int MyFunc();  //то же самое, что и в предыдущем объявлении. Явное
               указание типа возвращаемого значения является
               предпочтительным!
```

- *имя_функции* — его придумывает программист, используя правила и рекомендации, перечисленные в *разд. 2.2*, а компилятор ассоциирует это имя с адресом начала тела функции. Программист может давать функциям любые легальные имена, но для того чтобы имена легче воспринимались, можно все значащие составляющие имени начинать с заглавной буквы (например, `OnMouseMove`);
- *список_аргументов* — определяет количество и типы аргументов (параметров), передаваемых в функцию.

Синтаксис:

```
список_аргументов == тип_аргумента1 [имя_аргумента1] ,
                    тип_аргумента2 [имя_аргумента2]...
```

- если функция имеет несколько аргументов, то каждая пара *тип_аргумента* [*имя_аргумента*] разделяются запятыми, например:

```
(int n, double d);
```

- поле *имя_аргумента* при объявлении является необязательным и может быть опущено, т. к. компилятору важен только тип параметра (имя параметра в объявлении функции он просто игнорирует). Но хорошим тоном считается присутствие в объявлении функции формальных имен, которые могут предоставить программисту-клиенту дополнительную информацию о том, каким образом данный параметр будет использоваться функцией. Это упрощает использование и документацию функций.

Например, прототип стандартной функции `Си` для копирования строк выглядит следующим образом:

```
char* strcpy(char* dest, const char* source);
```

Из имен параметров `dest` и `source` сразу становится ясно, что первый параметр — это адрес строки-приемника, а второй — адрес строки-источника (причем ключевое слово **const** говорит о том, что содержимое

С этими формальными именами компилятор будет ассоциировать те фактические значения, которые переданы в стеке при вызове

```
{//тело функции содержит последовательность инструкций, которые будут  
    осуществлять действия над фактическими значениями, переданными в стеке  
    const double A=5.5, B=6.6, C=7.7;  
    double result = A*x*x + B*x + C; //вычисление результата  
    return result; //формирование возвращаемого значения  
}
```

Определение функции в некотором смысле аналогично записи формулы в алгебре:

$y = A * x * x + B * y + C$; //это задание правила, по которому будут производиться реальные вычисления

В формуле мы указываем, сколько переменных будет участвовать в вычислениях, какие действия с этими переменными следует производить. Само же вычисление можно произвести только тогда, когда для каждой из переменных будет задано числовое значение. Формирование конкретных значений для аргументов функции происходит при вызове функции.

При каждом вызове функции с аргументами компилятор добавляет в точку вызова машинные инструкции, которые записывают копии значений аргументов в стек. Функция в процессе исполнения пользуется копиями аргументов в стеке как локальными переменными, обращаясь к ним по именам, приведенным в списке формальных аргументов. Копии аргументов в стеке, с которыми функция будет оперировать, называют *фактическими аргументами*.

ЗАМЕЧАНИЕ 1

Компилятор ассоциирует имена формальных аргументов со значениями фактических в стеке. А программист пользуется этими именами как локальными переменными, область видимости и время жизни которых ограничивается фигурными скобками тела функции.

ЗАМЕЧАНИЕ 2

Имена формальных аргументов при определении функции вовсе не обязательно должны быть такими же, как соответствующие имена в списке аргументов при вызове и тем более при объявлении (где они вообще не обязательны).

ЗАМЕЧАНИЕ 3

Недопустимо в теле одной функции определять другую.

8.1.3. Вызов функции

Для эффективного и безопасного вызова функции, компилятор должен обеспечить:

- *связь по управлению*:
 - передачу управления на начало функции и возврат управления на следующую за вызовом функции инструкцию;
 - сохранение и восстановление контекста вызывающей части. Так как локальные переменные и вызывающей, и вызываемой функции компилятор располагает в одном и том же стеке, то он должен гарантировать монопольное использование каждой функцией своих локальных переменных. Поэтому компилятор выделяет для локальных переменных каждой функции область стека, которая называется стековым кадром (stack frame). Обращение к локальным переменным компилятор формирует относительно базового значения, которое хранится в регистре (в англоязычной литературе обычно называемом термином frame pointer);

Механизм вызова функции в стиле Си

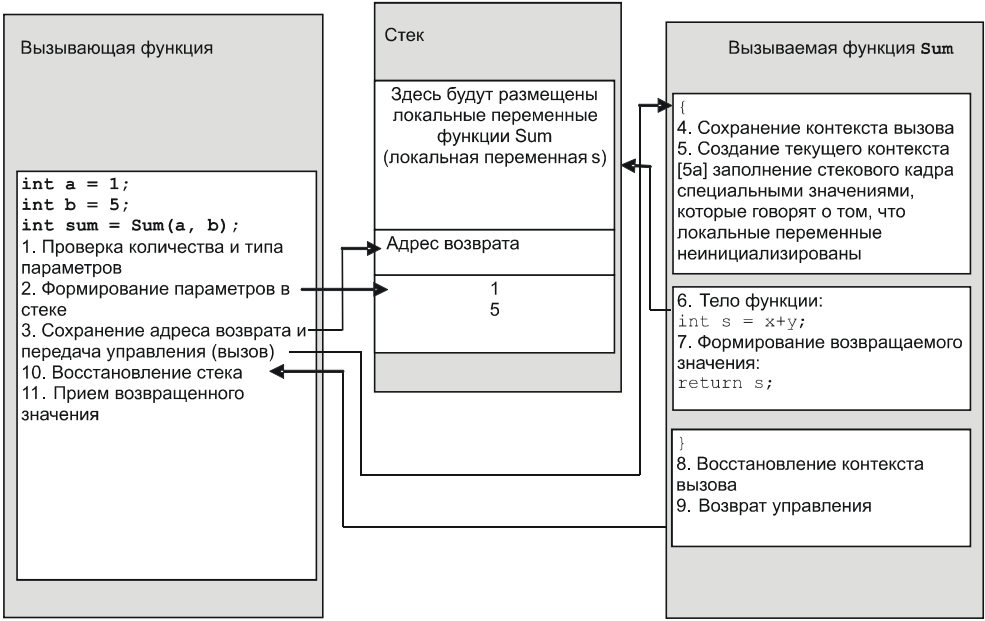


Рис. 8.5

- *связь по данным* — механизм, который позволяет передавать данные из вызывающей функции в вызываемую (pass parameters) и возвращать результат работы функции (return value) в вызывающую функцию.

Рассмотрим "анатомию" вызова функции (листинг 8.1), объявленной без спецификатора **inline** и вызываемой в стиле Си (с соглашением по вызову `__cdecl` — см. разд. 8.1.5). Соответствующие ассемблерные команды приведены для процессора x86.

Последовательность действий компилятора помечена цифрами 1—11 (рис. 8.5).

Листинг 8.1. "Анатомия" вызова функции

```
// Файл sum.h должен содержать интерфейс, предоставляемый сервером  
клиенту:
```

```
int Sum(int x, int y); //прототип функции
```

```
// Файл main.cpp (клиент, в котором осуществляется вызов функции)
```

```
#include "sum.h"
```

```
int main()
```

```
{
```

```
    int a = 1, b = 5;
```

```
    int sum = Sum(a, b); //1.Встречая в исходном тексте программы вызов
```

функции, компилятор использует ее прототип

для проверки количества и типов аргументов

и при необходимости делает преобразование

типов (например, вызов `Sum(1, 5.1)` будет

преобразован в `Sum(1, 5)`. Если преобразование

с точки зрения компилятора некорректно, то

будет выдана соответствующая диагностика

2.Для формирования параметров компилятор

вставляет низкоуровневую последовательность

команд, в нашем случае: `push b` и `push a`.

В стеке оказываются значения `5` и `1`

3.Для вызова функции компилятор генерирует

низкоуровневую команду `call Sum`.

При выполнении этой команды аппаратно в стеке

сохраняется адрес возврата, а затем управление

передается на первую команду функции

```

10. Восстановление стека
    (в нашем случае add esp, 8)
11. Прием возвращаемого значения (в нашем случае
    результат будет сформирован функцией
    в регистре: mov [sum], eax
...
}

// Файл sum.cpp (в котором находится реализация функции)
int Sum(int x, int y)
{ // Открывающая скобка функции превращается компилятором в:
    4. Сохранение контекста вызвавшей функции
      (запоминание адреса ее стекового кадра),
      в нашем случае – push ebp
    5. Формирование контекста текущей функции
      (стекового кадра для хранения локальных переменных
      текущей функции), в нашем случае:
      mov ebp, esp
      sub esp, n
      где n – количество байтов, необходимых для
      хранения локальных переменных текущей функции
      (в нашем случае 4)
    [5a]. В Debug-версии в некоторых реализациях
      трансляторов фирмы Microsoft для процессоров x86
      весь стековый кадр заполняется значениями 0хСС...С
int s=x+y;      // 6. Тело функции
return s;      // 7. Формирование возвращаемого значения (в нашем
                случае mov eax, s) и передача управления на
                закрывающую фигурную скобку функции
}              // 8. Восстановление контекста вызвавшей
                функции:
                mov esp, ebp
                pop ebp
                // 9. Возврат управления – ret

```

Специфика реализации вызова функции компилятором:

- при формировании параметров компилятор вставляет в код вызывающей функции последовательность машинных команд, которые, в свою очередь, помещают в стек копии переменных, перечисленных в списке аргументов

справа налево. Вместо имен переменных в нашем случае можно было бы указать непосредственно константы — `Sum(1, 5)`. В стеке были бы сформированы такие же значения.

Низкоуровневая команда **push**, с помощью которой компилятор формирует значения параметров в стеке, умеет работать только с операндами, размер которых равен размеру регистра, поэтому в следующем примере копии байтовых переменных `c1` и `c2` в стеке будут занимать каждая по 4 байта:

```
char c1=1, c2=2;
int n = sum(c1,c2);
```

- если программист в объявлении (и определении) описал функцию как возвращающую значение, то в теле этой функции должна быть инструкция:

```
return <выражение>;
```

Если такой инструкции нет, компилятор C++ в большинстве случаев выдаст ошибку, а компилятор Си — предупреждение.

При выполнении инструкции **return** выполняются следующие действия:

- вычисляется выражение;
 - формируется возвращаемое значение, при необходимости компилятор приводит его к требуемому типу (способы формирования возвращаемых значений *см. разд. 8.3.1*);
 - управление передается на закрывающую фигурную скобку функции, по которой восстанавливается контекст вызывающей функции и возвращается управление вызывающей функции;
- если функция ничего не возвращает (тип возвращаемого значения **void**), то можно использовать инструкцию:

```
return; //без <выражения>
```

В этом месте компилятор просто передаст управление на закрывающую фигурную скобку функции. В таком случае излишне завершать тело функции инструкцией **return**;

- инструкция **return** в соответствующей форме (**return**; или **return** <выражение>) может встречаться внутри функции много раз и в любом месте — как только возникает необходимость возврата. При этом прерывается выполнение функции, а управление передается на закрывающую фигурную скобку, например:

```
void f()
{
```

...

```
if(условие) return;
```

```
...
```

```
}
```

- обычно один из регистров, называемый *base frame pointer* (в процессорах архитектуры x86 для этого используется регистр *ebp*), используется компилятором для обращения к локальным переменным функции, поэтому его значение всегда обязательно сохраняется при сохранении контекста и, соответственно, восстанавливается при возврате из функции;
- рассмотренный механизм предполагает вызов функции в стиле Си. Если предписать компилятору ту же самую функцию вызывать с другим соглашением по вызову, то последовательность действий тоже будет отличаться.

8.1.4. Вызов *inline*-функции

Функцию можно определить со спецификатором **inline**. Такие функции называются *встроенными*. Ключевое слова **inline** указывает компилятору, что он должен пытаться каждый раз, встречая вызов функции, вместо последовательности действий 1—11, описанной в предыдущем разделе, подставлять тело функции (в отличие от обычных функций, для которых код существует в единственном экземпляре, а в месте вызова туда передается управление).

При использовании встроенных функций имеются:

- положительные моменты:
 - исключаются накладные расходы на вызов функции;
 - сохраняется структурированность исходного текста программы;
- отрицательный момент — обычно увеличивается объем кода программы (.exe-файла).

Специфика использования встроенных функций:

- реализация (тело) *inline*-функции должна быть в заголовочном файле, потому что при вызове (чтобы подставить тело) компилятор должен его видеть;
- ключевое слово **inline** является только вашим пожеланием компилятору (**inline** делает функцию кандидатом на встраивание). Реально такую вставку компилятор осуществляет только в том случае, если по его (компилятора) соображениям соотношение затраты/выигрыш (*cost/benefit*) является выгодным. Если вы пользуетесь компилятором Microsoft — VC, то можете заставить компилятор пренебречь такой оценкой и, независимо от соотношения, заставить его встраивать функции, используя ключевое слово **__forceinline** вместо **inline**;

- компилятор не может встроить следующие функции (даже если вы укажете ему `__forceinline`):
- рекурсивные функции (хотя посредством директивы препроцессора `#pragma inline_recursion on` все же можно заставить компилятор такую функцию встроить с точностью до ограничений на количество рекурсий);
 - функции, вызов которых осуществляется посредством указателя;
 - функции с переменным числом параметров.

Пример встроенной функции:

//файл `sum.h` (в нем находится объявление встроенной функции, совмещенное с определением)

```
inline int Sum(int x, int y) //встроенная функция
{
```

```
    return (x+y);
```

```
}
```

//файл `main.cpp` (в нем находится вызов встроенной функции)

```
int main()
```

```
{
```

```
    ...
```

```
    int z = Sum(x,y); //вместо вызова функции в этом месте компилятор
                        подставит ее тело
```

```
}
```

ЗАМЕЧАНИЕ 1

Не злоупотребляйте `inline`-функциями! Встроенными рекомендуется делать только очень маленькие функции. При этом текст программы остается структурированным, а накладных расходов на вызов функции удастся избежать.

ЗАМЕЧАНИЕ 2

Используйте `inline`-функции вместо макросов (`#define`), тогда артефактов, которые не устранить даже скобками (см. разд. 5.2.1), удастся избежать.

Сравните два варианта реализации одного и того же действия: посредством макроса с параметрами и посредством `inline`-функции (листинг 8.2).

Листинг 8.2. Использование встроенных функций вместо макроопределений

```
//1.
```

```
#define SQUARE(x) (x)*(x) //макроопределение
```

```
inline double Square(double x) { return x*x;} //встроенная функция
```

```

int main()
{
    int x = 2.2;
    double res1 = SQUARE(x++); //параметр макроса будет
                               инкрементирован дважды!
    double res2 = Square (x++); //параметр функции компилятор
                               гарантированно модифицирует один раз!
}
//2.

#define F(x) (x)/2 //макроопределение
inline double f(double x) { return x/2;} //встроенная функция
int main()
{
    double res1 = F(5); // res1=2, т. к. препроцессор сделает
                        макроподстановку, а затем компилятор
                        по своим правилам вычислит выражение
    double res2 = f(5); // res2=2.5, т. к. прежде, чем вызывать
                        функцию, компилятор приведет тип
                        параметра к требуемому, а затем
                        подставит тело функции
}

```

8.1.5. Соглашения о вызове функции

Существуют разные способы кодирования вызова функций на низком уровне (т. е. тех последовательностей низкоуровневых инструкций, в которые компилятор превращает вызов функции, написанный на языке высокого уровня). Это важно знать при использовании библиотечных функций (например, при использовании системных функций Windows, которые обычно используют соглашение о вызове **__stdcall**) или при сопряжении вашего кода и кода, возможно, написанного на другом языке программирования. Для указания компилятору C/C++, во что превращать вызов, существуют следующие ключевые слова, которые программист может использовать при объявлении (и определении) функции:

__cdecl, **__stdcall**, **__fastcall**

ЗАМЕЧАНИЕ

Такие ключевые слова, как **__pascal**, **__fortran**, для 32-разрядных компиляторов считаются устаревшими.

Соглашения определяют следующие моменты:

- ❑ как передаются параметры — помещаются ли они в регистры процессора (и в какие именно) либо в стек (если да, то в каком порядке);
- ❑ кто восстанавливает стек (вызывающая или вызываемая функция);
- ❑ как формирует компилятор внутреннее имя, т. е. в каком виде имена функций хранятся в объектном модуле (для использования компоновщиком).

Рассмотрим различия соглашений о вызове на примере (предполагая, что программа оттранслирована для процессоров x86). Вызовем одну и ту же функцию с разными соглашениями по вызову:

```
void calltype f(char c, short s, int i, double d); //где calltype может
                                                    иметь значение __cdecl, __fastcall или __stdcall

int main()
{
    f('A', 10, 9999, 1.23); //вызов
}
```

Рассмотрим вызов этой функции с разными соглашениями о вызове.

Соглашение о вызове в стиле Си — `__cdecl` (табл. 8.1). Это соглашение о вызове по умолчанию для C/C++-программ, поэтому явно его можно не указывать (но программист может изменить это умолчание в опциях командной строки компилятору или задать опции в интегрированных средах с помощью визуальных средств).

Таблица 8.1. Вызов функции с соглашением `__cdecl`

Характеристики вызова	Реализация
Как осуществляется возврат	Низкоуровневой командой <code>ret</code> . При этом управление аппаратно передается на адрес возврата, а регистр ESP перемещается на первый параметр
Кто корректирует стек — вызывающая или вызываемая функция	Стек корректирует вызывающая функция. В нашем примере — <code>add ESP, 20</code> , (где 20 — это количество байтов, занимаемых в стеке параметрами)
Как формируется компилятором Си внутреннее имя (которое хранится в объектном и исполняемом модуле и используется компоновщиком)	<code>_f</code>

Таблица 8.1 (окончание)

Характеристики вызова	Реализация										
Как передаются параметры	Параметры формируются справа налево в стеке низкоуровневой командой push: Стек ESP → <table><tr><td>Адрес возврата</td><td></td></tr><tr><td>'А'</td><td>4 байта</td></tr><tr><td>10</td><td>4 байта</td></tr><tr><td>9999</td><td>4 байта</td></tr><tr><td>1.23</td><td>8 байтов</td></tr></table> Регистры для передачи параметров не используются Замечание Так как команда push умеет заносить в стек только значения, размер которых совпадает с размером регистра, то короткие целые (char, short) при передаче в качестве параметра функции в стеке будут занимать на 32-разрядной платформе 4 байта. А для того чтобы передать функции параметр типа double или long long, компилятор просто два раза использует команду push	Адрес возврата		'А'	4 байта	10	4 байта	9999	4 байта	1.23	8 байтов
Адрес возврата											
'А'	4 байта										
10	4 байта										
9999	4 байта										
1.23	8 байтов										
Как формируется внутреннее декорированное компилятором C++ (VC) имя	?f@@YAXDFHN@Z Здесь А означает соглашение о вызове __cdecl, а все остальные символы означают количество и типы параметров и т. д.										

Соглашение о вызове **__stdcall** (табл. 8.2). Может быть использовано в Си-программе для уменьшения объема исполняемого файла, для вызова функций из динамически загружаемых библиотек, написанных на других языках, а также при программировании под Windows при вызове системных Win32API-функций.

Таблица 8.2. Вызов функции с соглашением `__stdcall`

Характеристики вызова	Реализация										
Как передаются параметры	Так же, как и при соглашении по вызову <code>__cdecl</code>. Параметры формируются справа налево в стеке низкоуровневой командой push: Стек ESP → <table><tr><td>Адрес возврата</td><td></td></tr><tr><td>'А'</td><td>4 байта</td></tr><tr><td>10</td><td>4 байта</td></tr><tr><td>9999</td><td>4 байта</td></tr><tr><td>1.23</td><td>8 байтов</td></tr></table> Регистры для передачи параметров не используются	Адрес возврата		'А'	4 байта	10	4 байта	9999	4 байта	1.23	8 байтов
Адрес возврата											
'А'	4 байта										
10	4 байта										
9999	4 байта										
1.23	8 байтов										
Как осуществляется возврат	Низкоуровневой командой ret n . При этом управление аппаратно передается на адрес возврата, а регистр ESP изменяет свое значение на n байт										
Кто корректирует стек — вызывающая или вызываемая функция	Стек корректирует вызываемая функция при возврате управления (для чего используется команда процессора x86 ret n , которая не только выполняет возврат из функции, но и увеличивает содержимое указателя стека на n байтов, освобождая тем самым место, занятое параметрами)										
Как формируется внутреннее имя компилятором Си	<code>_f@20</code> Здесь 20 — это количество байтов, занимаемых параметрами в стеке										
Как формируется внутреннее декорированное имя компилятором C++ (VC)	<code>?f@@YGXDFHN@Z</code> Здесь G означает соглашение о вызове <code>__stdcall</code>										

Соглашение о вызове `__fastcall` (табл. 8.3). Компилятор оптимизирует передачу параметров и, соответственно, время выполнения за счет того, что часть параметров передается в регистрах.

Таблица 8.3. Вызов функции с соглашением `__fastcall`

Характеристики вызова	Реализация										
Как передаются параметры	Часть параметров (первые два слева, размер которых не превышает размера регистра) передаются в регистрах ECX, EDX, остальные — в стеке справа налево низкоуровневой командой push: Стек ESP → <table><tr><td>Адрес возврата</td><td></td></tr><tr><td>9999</td><td>4 байта</td></tr><tr><td>1.23</td><td>8 байтов</td></tr></table> Регистры <table><tr><td>ECX</td><td>'A'</td></tr><tr><td>EDX</td><td>10</td></tr></table>	Адрес возврата		9999	4 байта	1.23	8 байтов	ECX	'A'	EDX	10
Адрес возврата											
9999	4 байта										
1.23	8 байтов										
ECX	'A'										
EDX	10										
Как осуществляется возврат	Низкоуровневой командой ret n										
Кто корректирует стек — вызывающая или вызываемая функция	Стек корректирует вызываемая функция										
Как формируется внутреннее имя компилятором Си	@f@20										
Как формируется внутреннее декорированное имя компилятором C++ (VC)	?f@@YIXDFHN@Z Здесь I означает соглашение о вызове __fastcall										

Декорированные имена — это внутренние (служебные) имена, которые формируются компилятором, хранятся в объектных файлах и используются компоновщиком для безопасной сборки исполняемого файла (см. разд. 3.6.1). Схема декорирования зависит от конкретного компилятора. При формирова-

нии декорированного имени компилятор добавляет к имени, данному пользователем, дополнительные символы, несущие информацию:

- ☐ о типах параметров, принимаемых функцией;
- ☐ о соглашении о вызове;
- ☐ об имени пространства имен;
- ☐ об имени класса, если функция является членом класса,
- ☐ и т. д.

8.2. Способы передачи параметров в функцию

Существуют два способа передачи параметров функции (рис. 8.6): по значению, когда функции передается копия значения, и по адресу, когда в распоряжение функции предоставляется адрес памяти, где находится оригинал параметра. В свою очередь, адрес в C++ может быть представлен двумя конструкциями — указателем и ссылкой.

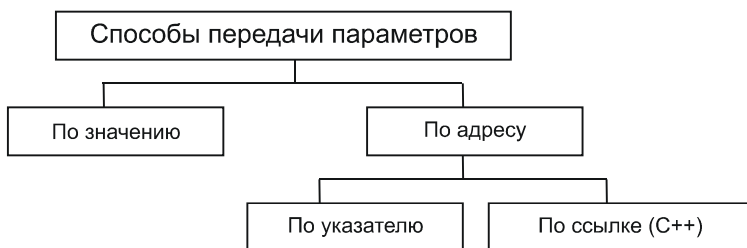


Рис. 8.6

8.2.1. Передача параметров по значению (*Call-By-Value*)

Передача параметров по значению (англоязычный эквивалент *Call-By-Value*) — это простая передача копий переменных в стек, не оставляющая никаких возможностей для изменения значений самих переменных в вызывающей функции. Пример приведен в листинге 8.3.

Листинг 8.3. Передача параметра по значению

```
double mupow(double x, int n) //возведение в целую положительную степень
{
    if(n<0) return -1.; //проверка корректности параметра
```

```

double res=1; //здесь будем копить результат
for(; n>0;n--) //при модификации переменной цикла изменяется копия
                второго параметра, помещенная в стек!
{ res*=x;}
return res;
}

int main(void)
{
    double d= mypow(2.2,10); //в функцию передаются значения 2.2 и 10
//или
    double y=2.2;
    int m=10;
    double d=mypow(y,m); //в функцию в стеке передаются копии значений
                        переменных y и m, поэтому, что бы мы
                        ни делали с копиями в вызываемой функции,
                        в вызывающей функции значения переменных
                        y и m не изменятся!
}

```

Передача параметров по значению применяется в тех случаях, когда:

- ❑ общий объем передаваемых аргументов невелик;
- ❑ вы намеренно предоставляете функции копию большого объекта для использования (модификации), гарантируя неизменность оригинала в вызывающей части программы.

8.2.2. Передача параметров по адресу

Этот способ предполагает, что в качестве аргументов функция получает не копии объектов, а их адреса. Передача адреса объекта во многих случаях гораздо эффективнее (и одновременно потенциально опаснее) передачи копии самого объекта, т. к. посредством адреса функция получает доступ к самому объекту, а накладные расходы на передачу адреса обычно значительно меньше, чем при передаче копии объекта. Но следует помнить, что, обладая адресом объекта, программист получает возможность изменить значение объекта по этому адресу (что не всегда желательно!).

Вызовы функций с передачей адреса объекта в языке C++ подразделяются на вызовы с передачей по указателю и с передачей по ссылке. Такое деление является условным, но имеет смысл, поскольку указатели и ссылки подчиняются разным синтаксическим правилам (см. разд. 7.2).

Передача адреса посредством указателя

Такая передача используется для получения в функции доступа к массивам и другим большим объектам.

Пример приведен в листинге 8.4.

Листинг 8.4. Передача параметров по указателю

```
//1. Функция подсчета символов в строке (эмуляция функции стандартной
    библиотеки strlen)
int MyStrlen(const char* p) //в качестве параметра функция получает адрес
                             начала массива. Ключевое слово const не
                             разрешит компилятору внутри функции
                             посредством адреса модифицировать
                             содержимое строки
{
    if(p==0) return -1; //защита от обращения по нулевому адресу
    int n=0; //инициализация счетчика символов
    while(*p) //пока с помощью указателя не дошли до конца строки
               (признаком конца строки в C/C++ является нулевой байт)
    {
        p++; //перемещаем указатель на следующий элемент
        n++; //учитываем очередной символ
    }
    return n;
}

//Вызов:
int main()
{
    char ar[10]="QWERTY";
    int count = MyStrlen(ar); //имя массива эквивалентно указателю
                              на нулевой элемент
    count = MyStrlen("ABC"); //компилятор отправит в функцию адрес
                              строкового литерала
}

//2. Функция, выполняющая копирование строки (эмуляция функции
    стандартной библиотеки strcpy):
int MyStrcpy(char* dest, const char* source) //при этом dest (приемник)
                                              должен содержать достаточно места для копирования!
```

```

{
    if(dest==0 || source==0) return 0; //защита от обращения по
                                     нулевому адресу

    int n=0; //счетчик символов
    while(*source !=0) //пока не дошли до конца строки-источника
    {
        *dest = *source; //копируем в приемник очередной
                        байт источника
        source++; //перемещаем указатель на следующий символ
        dest++;   //подготавливаем приемник для копирования
                следующего символа
        n++;      //увеличиваем счетчик символов
    }
    *dest=0; //формируем в приемнике признак конца строки
    return n;
}
//или:
int MyStrcpy(char* dest, const char* source)
{
    if(dest==0 || source==0) return 0;
    int n=0;
    while(*dest=*source) //Сначала по адресу приемника будет
                        скопирован очередной байт источника,
                        а потом скопированное значение будет
                        использовано для формирования условия
                        (нулевое значение эквивалентно false)
    {
        source++;
        dest++;
        n++;
    }
    return n;
}

```

Передача параметров по ссылке

Если функция принимает ссылку на объект, то компилятор должен при вызове функции сформировать в стеке адрес объекта (см. разд. 7.2). Синтаксис

передачи параметра по ссылке на первый взгляд кажется странным, поэтому попробуем сформулировать формальные правила:

- ❑ для того чтобы сформировать параметр-ссылку, при вызове следует указать имя объекта, адрес которого мы хотим таким образом передать. При этом компилятор, видя прототип функции, где в качестве типа параметра указана ссылка, сформирует в стеке адрес объекта;
- ❑ внутри функции синтаксически пользуемся ссылкой так же, как обращались бы к самому объекту (а компилятор, зная о том, что ссылка — это замаскированный адрес, получает значение по этому адресу).

Пример.

Пусть функция должна сформировать несколько результатов: сумму, разность... (а возвращаемое значение только одно — в такой ситуации не поможет!). Это означает, что следует передать в функцию адреса, по которым она сформирует результаты. Для сравнения реализуем такую функцию двумя способами (табл. 8.4). И в одном, и в другом случае в функцию передаются адреса переменных, в которых требуется сформировать результаты, однако синтаксически использование адресов выглядит по-разному.

Таблица 8.4. Сравнение передачи параметров по указателю и по ссылке

Передача параметров по указателю	Передача параметров по ссылке
<pre>//Определение функции void fPointer(int x, int y, int* sum, int* sub) { *sum=x+y; *sub=x-y; } //Вызов int main() { int a=2,b=1,r1,r2; fPointer(a,b,&r1,&r2); }</pre>	<pre>//Определение функции void fReference(int x, int y, int& sum, int& sub) { sum=x+y; sub=x-y; } //Вызов int main() { int a=2,b=1,r1,r2; fReference (a,b,r1,r2); }</pre>

Если при передаче адреса требуется запретить функции модифицировать значение по этому адресу, следует использовать ключевое слово **const**:

```
void f(const int& r)
{
```



```
    //r++; //ошибка: запрещено модифицировать значение параметра
}
```

Здесь стоит вспомнить о таком понятии, как константная ссылка (разд. 7.2).

Например:

```
void f(int& r){}
int main()
{
    f(5); //ошибка: компилятор не может сформировать адрес
          константы
}
//Но!
void f(const int& r){}
int main()
{
    f(5); //ОК: компилятор создает фиктивную переменную,
          заносит туда значение 5, а адрес переменной
          отправляет в качестве параметра в функцию
}
```

8.2.3. Специфика передачи параметров

Выражения в качестве параметров

Язык C/C++ позволяет указывать в качестве параметров достаточно сложные выражения (листинг 8.5).

РЕКОМЕНДАЦИЯ

Далеко не всегда следует усложнять текст своей программы только потому, что язык программирования обладает такими возможностями. Пишите программы как можно проще и элегантнее.

Листинг 8.5. Примеры использования выражений в качестве параметров функции

//1. Если требуется передать в функцию адрес переменной, то совершенно необязательно для этих целей определять вспомогательную переменную (указатель или ссылку). Можно сформировать требуемый адрес непосредственно при вызове:

```
void f(int*);
```

```
int main()
```

```
{
```

```
    int x=1;
```

//Два варианта формирования в качестве параметра адреса переменной X

<pre>//для того чтобы в функцию отпра- вить адрес <i>объекта</i>, можно завести вспомогательную переменную- указатель (компилятор отведет под нее память)</pre>	<pre>//можно не засорять программу та- кими вспомогательными переменными, а использовать выражение в качест- ве параметра. Тогда компилятор сразу в качестве параметра в стеке сформирует адрес переменной</pre>
---	--

```
    int* p = &x;
```

```
    f(p);
```

```
    f(&x);
```

```
}
```

//2. Пусть в программе реализованы две функции:

```
int f1();
```

```
void f2(int,int);
```

```
int main()
```

```
{
```

```
    int x=1, y=5;
```

//Параметры при вызове f2 можно формировать по-разному:

<pre>int z = f1(); int w = x+y; f2(z, w); //понятнее</pre>	<pre>f2(f1(), x+y); //компактнее</pre>
--	--

```
}
```

//3. Использование оператора ", " при формировании параметров:

```
void f(int);
```

```
int main()
```

```
{
```

```
    int x;
```

//Пусть значение параметра функции f должен сформировать пользователь:

<pre>std::cin>>x; f(x);</pre>	<pre>//f(cin>>x,x);//ошибка: указаны два параметра, а функция принимает один! f((cin>>x,x)); ОК!</pre>
-------------------------------------	--

```
}
```

Указатели на массивы в качестве параметров функции

Довольно часто возникает необходимость передать в функцию массив (листинг 8.6).

ВАЖНО!

Компилятор C/C++ никогда не передает массивы по значению (помещать в стек копию мегабайтного массива было бы слишком неэффективно!) — передается только указатель на массив.

Листинг 8.6. Передача массивов в функцию

```
#include "1.h"

int main()
{
    //Даны три массива. Тип элементов массивов — любой (T):
        T ar1[K];
        T ar2[K][N];
        T ar3[K][M][N];

    //Требуется передать в функцию все три массива
        F(ar1, ar2, ar3);
}

//Файл 1.h
const int N = 5, M = 3, K = 2; //константы для размерностей рекомендуется
                                задавать в заголовочном файле

//Прототип функции:
void F(T* p1, T (*p2)[N], T (*p3)[M][N]);
//или void F(T p1[], T p2[][N], T p3[][M][N]); //поскольку в функцию
                                передается указатель, и старшая размерность массива
                                не участвует в вычислении адреса элемента, то можно
                                оставить самые левые [] пустыми или даже написать
                                там любое значение!

//Файл 1.cpp
#include "1.h" //необходимо подключить 1.h, т. к. в нем определены
                значения размерностей массивов

void F(T* p1, T (*p2)[N], T (*p3)[M][N])
//или void F(T p1[], T p2[][N], T p3[][M][N])

//Замечание. Оба варианта задания типов параметров имеют для компилятора
                один и тот же смысл. Как бы не был объявлен каждый параметр,
                вы можете пользоваться именем параметра так же, как
                пользовались бы именем соответствующего массива, указатель
                на который получает функция

{
    p1[i].        ..//или *(p1+i)
```

```

    p2[i][j].    ..//или *(* (p2+i) +j)
    p3[i][j][k]... //или *(* (* (p3+i) +j)+k)

//Но!

    size_t n = sizeof(p1); //n=4 (количество байтов, занимаемых
                               указателем)

}

```



Подумайте, каким будет результат оператора **sizeof** в следующих выражениях?

```

size_t n1 = sizeof(p2);
size_t n2 = sizeof(p3);

```

Ссылки на массивы в качестве параметров

В языке C++ можно использовать ссылки на массив. Имя такой ссылки является псевдонимом имени самого массива. Пример использования ссылки для передачи массива в функцию в качестве параметра приведен в листинге 8.7.

Листинг 8.7. Передача ссылки на массив в функцию

```

void f(int (&ar)[5][10] )//а) выражение (&ar) должно быть заключено в
                               скобки, иначе компилятор решит,
                               что & относится к типу элемента массива
                               б) при передаче ссылки должны быть указаны
                               все размерности: int (&ar)[5][10] - ошибка!

{
    size_t n = sizeof(ar); //количество байтов, занимаемых массивом
                           равно 200
    ar[0][0]=1;           //синтаксически пользуемся ссылкой так же,
                           как именем массива
}

int main()
{
    int ar[5][10];
    f(ar);
}

```

Значения параметров по умолчанию (только в C++)

В языке C++ можно задавать значения параметров по умолчанию. Это означает, что если программист не указал значение одного или нескольких параметров при вызове функции, то компилятор по умолчанию сам может под-

ставить эти значения (при условии соблюдения программистом определенных правил).

Опишем эти правила:

- просто так компилятор ничего подставлять не будет. Если объявлена функция с двумя параметрами:

```
void f(int, char);
```

то при попытке ее вызова с одним параметром:

```
f(1);
```

компилятор выдаст ошибку о несоответствии количества параметров (он ожидает два).

Если же при объявлении функции программист укажет, что данная функция может иметь параметры по умолчанию и приведет их значения, например:

```
void f(int, char = 'A'); //таким образом задается значение
```

по умолчанию для второго параметра

то компилятор поймет, что в том случае, когда программист при вызове указал один параметр, значение второго требуется сформировать по умолчанию.

Теперь такую функцию можно вызывать двумя способами:

```
f(1, 'B'); //компилятор сформирует в стеке значения, указанные
```

программистом (код символа 'B' и 1)

```
f(1); //а можно и так. Компилятор автоматически добавит значение
```

для второго параметра, заданное в прототипе, и в стеке будут

сформированы значения 'A' и 1

- при объявлении можно указать, что функция принимает несколько параметров по умолчанию, но все они (обязательно!) должны располагаться в конце списка аргументов, например:

```
//Объявлена функция, у которой два параметра имеют значения
```

по умолчанию

```
void f(char, int=2, double=1.1); //OK
```

```
//void f(char='A', int, double=1.1); //ошибка: не указано значение
```

по умолчанию для второго параметра

```
//Вызывать такую функцию можно следующими способами:
```

```
f('Q',1,2.2); //функции будут переданы значения 'Q', 1, 2.2
```

```
f('W',5); // 'W', 5, 1.1
```

```
f('Z'); // 'Z', 2, 1.1
```

```
//f(); //ошибка: несоответствие количества параметров
```

- при вызове функции, имеющей параметры по умолчанию, опускать (не указывать) значения можно только подряд с конца списка параметров (т. е. оставить умолчание в середине списка параметров нельзя):

```
//Объявлена функция, у которой все параметры имеют значения по умолчанию:
```

```
void f(int n=1, const char* p="ABC", double d=1.1);
```

```
//Вызывать такую функцию можно следующими способами:
```

```
f(5, "QWERTY", 2.2);
```

```
f(5, "QWERTY");
```

```
f(5);
```

```
f();
```

```
//Но!
```

```
f("QWERTY", 2.2); //ошибка: компилятор пытается преобразовать тип  
строкового литерала к типу первого  
параметра int, но не может этого сделать
```

```
//поставить запятую вместо пропущенного по умолчанию параметра тоже  
не допускается
```

```
//f(5_,2.2); //ошибка: лишний символ , (запятая)
```

- значения по умолчанию обязательно должны быть указаны при объявлении функции, т. к. компилятор видит только прототип функции, когда формирует вызов функции (т. е. именно из прототипа он может взять значения по умолчанию). Лучше не дублировать эти значения при определении функции, т. к. в одних реализациях у компилятора возникнет двойственность и он выдаст ошибку, а в других — компилятор просто проигнорирует значение, указанное при определении!

Неиспользуемые параметры (Visual C++)

Бывают случаи, когда после модификации функции оказывается, что какой-либо из параметров становится ненужным. В Visual C++ (для совместимости со старыми версиями) оставляют объявления прежними, а в определении опускают имя такого параметра.

Например:

```
void Func(int x, int y, char*) //отсутствие имени формального параметра —  
это указание компилятору о том, что параметр  
использоваться не будет (иначе компилятор выдаст  
предупреждение (warning) о том, что данная  
переменная не используется)  
  
{  
    ...  
}
```

Параметры функции *main*

Функция *main* (головная функция программы) может иметь несколько форм (см. разд. 1.2.2):

- ❑ в версии для ANSI-символов:

```
int main( void );
int main( int argc [ , char * argv [ ] [, char *envp[ ] ] ] );
```

- ❑ в версии для Unicode-символов:

```
int wmain( void );
int wmain( int argc [ , wchar_t * argv [ ] [, wchar_t *envp[ ] ] ] );
```

Расшифровка параметров для обеих версий ANSI и Unicode поясняется рис. 8.7, где:

- ❑ *argc* — число параметров командной строки (формируемой при вызове функции *main* в ходе запуска программы), указанных посредством *argv*. Имя программы тоже является одним из параметров (поэтому значение *argc* всегда ≥ 1);
- ❑ *argv* — массив указателей на строки. Содержимое строк — это параметры командной строки, разделенные пробелом (эту информацию может сформировать запускающая программа или указать пользователь при вызове программы в командной строке). Каждый элемент массива *argv* содержит указатель на строку, а строка содержит слово из командной строки и завершающий ноль (0).

По соглашению:

- *argv[0]* — это спецификация запускаемого на выполнение исполняемого файла (с полным путем);
- *argv[1]* — первый аргумент командной строки;
- *argv[argc]* — всегда 0.

Таким образом, первый аргумент командной строки всегда *argv[1]*, а последний — *argv[argc - 1]*;

- ❑ *envp* — это массив указателей на строки, каждая из которых описывает переменную операционной среды или операционного окружения (user environment). Количество строк зависит от текущей конфигурации, а признаком конца является нулевой указатель. Это специфика Microsoft — расширение стандарта ANSI C. В качестве переменных окружения может передаваться следующая информация:

- то, что связано с конкретным пользователем (различные пути): где находится профиль пользователя, папка Application Data...;

- имя компьютера;
- количество процессоров;
- тип ОС;
- местоположение системного и основного каталога ОС;
- многое другое (подробную информацию об операционном окружении см. в справочной системе (MSDN Library) по ключевым словам: `environment variables`).

Сами строки, передаваемые с помощью `envp`, — это копия текущего окружения (на момент запуска вашей программы). Поэтому если вы изменяете окружение с помощью соответствующих функций стандартной библиотеки (`putenv()` или `_wputenv()`) в процессе выполнения программы, то содержимое блоков, на который указывает каждый указатель `envp[i]`, не изменяется, хотя текущее окружение реально изменяется. В такой ситуации получить информацию о реальном текущем окружении можно посредством вызова `getenv()/_wgetenv()`.

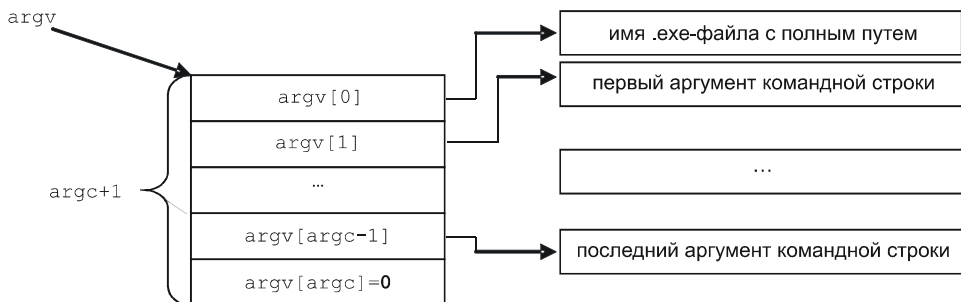


Рис. 8.7

Листинг 8.8 демонстрирует, как можно распечатать параметры командной строки (сформированные при запуске). Пусть программа имеет имя `my.exe` и находится в текущем каталоге. Вызовем ее следующим образом:

```
my.exe one two three:
```

Листинг 8.8. Вывод параметров командной строки

```
int main(int argc, char* argv[])
{
    //Параметры командной строки
    for(int i=0;i<argc;i++)
```



```

    {
        std::cout<<argv[i]<<std::endl;
    }
}

```

Если исполняемый файл `my.exe` находился, например, в каталоге `c:\mydir`, то будет выведено:

```

c:\mydir\my.exe
one
two
three

```

Листинг 8.9 демонстрирует, как можно вывести информацию о переменных окружения, полученную приложением посредством параметра `envp`.

Листинг 8.9. Вывод переменных окружения

```

int main(int argc, char* argv[], char *envp[] )
{
    //Переменные среды программы
    for(int i=0; envp[i]!=0; i++)
    {
        std::cout<< envp[i]<<std::endl;
    }
}

```

8.2.4. Переменное число параметров

Язык C/C++ допускает использование переменного числа параметров.

ЗАМЕЧАНИЕ

Использование переменного числа параметров возможно только для функций с соглашением по вызову в стиле Си (`__cdecl`), т. к. только при таком соглашении стек восстанавливает вызывающая функция. Генерируя вызов функции, компилятор всегда знает, сколько параметров он сформировал в стеке, поэтому может в каждом случае организовать в вызывающей части программы коррекцию стека на требуемую величину (эта величина для разных вызовов изменяется в зависимости от фактического числа параметров).

Специфика использования функций с переменным числом параметров:

- ❑ признаком функции с переменным числом аргументов является многоточие (...) в списке параметров (как при объявлении, так и при определении);

- ❑ встретив многоточие в объявлении функции, компилятор прекращает контроль соответствия типов параметров при вызове, поэтому за все остальные аргументы несет ответственность программист;
- ❑ у такой функции должен быть хотя бы один обязательный параметр;
- ❑ функция с переменным числом параметров должна иметь способ определения точного их числа при каждом вызове (в каждом конкретном случае разработчик такой функции придумывает способ, позволяющий определить количество и типы параметров, а программист, вызывающий эту функцию, должен следовать правилам разработчика).

Примеры объявления функции с переменным числом параметров:

- ❑ `int Func1(int i, ...);` //функцию можно вызывать с одним и более параметрами
- ❑ `int Func2(int i, char byte, ...);` //функция должна иметь не менее двух параметров при вызове

Листинг 8.10 иллюстрирует вызов функции с переменным числом параметров (пояснение — на рис. 8.8). В нем разработчик функции использовал в качестве признака окончания списка параметров дополнительный параметр — признак, всегда равный `-1`.

Листинг 8.10. Функция получает любое количество целых положительных параметров типа `int` и возвращает их среднее значение

```
//файл 1.h
//прототип функции с переменным числом параметров
double Average( int nFirst, ... );

//файл main.cpp (вызов функции)
#include <iostream>
int main( void )
{
    //вызов функции с тремя целыми параметрами (-1 — признак конца)
    double aver = Average( 2, 3, 4, -1 ) //на рис. 8.8 показано,
                                         каким образом компилятор
                                         формирует стек при вызове
                                         функции

    std::cout<< "Average = "<< aver <<std::endl;
    //вызов функции с четырьмя целыми параметрами
    std::cout<< "Average = "<< Average( 2, 3, 4, 5,-1 ) <<std::endl;
```

```
//вызов функции только с признаком конца
std::cout<< "Average = "<< Average(-1) <<std::endl;
}
//файл 1.cpp (реализация функции).
//вариант 1 (без использования макросов стандартной библиотеки).
double Average( int first, ... )
{
    int count = 0, sum=0;
    int* p = &first; //установили вспомогательный указатель на первый
                      //параметр
    while(*p != -1)
    {
        sum += *p++; //суммируем значение по текущему адресу
        p++; //перемещаем указатель на следующий параметр
        count++;
    }
}
```



Подумайте, как корректно сформировать возвращаемое значение, чтобы:

- а) не потерять точность;
- б) можно было бы вызвать такую функцию без единого содержательного параметра, а только с признаком конца (-1)

```
return <возвращаемое_значение>;
```



Рис. 8.8

Приведенный в листинге 8.10 пример довольно простой, т. к. все необязательные параметры одного и того же типа — `int`. На самом деле задача усложняется, когда параметры разного типа, и нужно иметь возможность определять не только количество параметров, но и тип каждого, чтобы правильно модифицировать указатель!

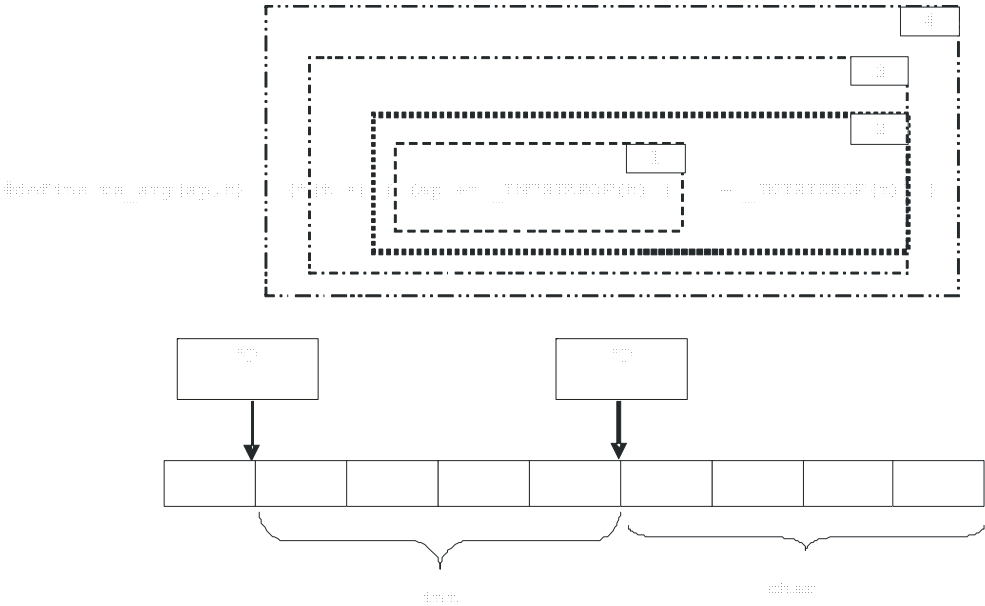


Рис. 8.9

Стандартная библиотека предоставляет несколько макрокоманд для манипуляции параметрами такой функции (макросы определены в заголовочном файле `<stdarg.h>`): `va_start`, `va_arg`, `va_end` и `va_list`. Выполнение макроса `va_arg` поясняется рис. 8.9.

```
typedef char *va_list; //используется для описания универсального
                        //указателя (т. к. байт является наименьшей
                        //адресуемой единицей, то в такой указатель
                        //можно занести адрес объекта любого типа)

#define va_start(ap,v) ( ap = (va_list)&v + _INTSIZEOF(v) ) //где:
                        //v-это имя последнего обязательного параметра,
                        //ap — имя универсального указателя. Макрос
                        //направляет универсальный указатель на первый
                        //необязательный параметр
```

```
#define va_arg(ap,t) (*(t*)((ap += _INTSIZEOF(t))-_INTSIZEOF(t))) //
```

ap — универсальный указатель,
t — тип текущего параметра (который должен быть известен). Макрос формирует значение текущего параметра, а универсальный указатель сдвигает на следующий аргумент (рис. 8.9).
Последовательность выполнения тела макроса помечена цифрами 1–4

```
#define va_end(ap) ( ap = (va_list)0 ) //обнуляет универсальный  
указатель
```

При перемещении указателя используется еще один вспомогательный макрос `_INTSIZEOF`, который очень хитро вычисляет, сколько байтов в стеке занимает переменная типа `t`:

```
#define _INTSIZEOF(t) ((sizeof(t)+sizeof(int)-1) & ~(sizeof(int)-1))
```

Используем перечисленные макросы для модификации предыдущего примера (листинг 8.11).

Листинг 8.11. Пример использования макросов стандартной библиотеки

```
//файл 1.cpp
#include <stdarg.h>
double Average( int first, ... )
{
    int count = 0, sum = 0;
    int i = first; //значение обязательного параметра (в частности,
                  //может быть -1)
    va_list p;     //универсальный указатель
    va_start( p, first ); //направили универсальный указатель на
                          //первый необязательный параметр
    while( i != -1 )
    {
        sum += i;
        count++;
        i = va_arg( p, int ); //получили значение текущего
                              //параметра, а p переместили на
                              //следующий
    }
}
```

```
va_end( p ); //в данном случае обнулять универсальный указатель
              //необязательно, т. к. область видимости и время
              //жизни локальной переменной p заканчивается по
              //закрывающей скобке функции
return (count !=0 ) ? static_cast<double>(sum)/count : 0;
//подсказка для тех, кто выполнил предыдущее задание
```

Функции стандартной библиотеки *printf*, *scanf* (процедурно-ориентированные потоковые функции ввода/вывода)

Поток (stream) — это устоявшийся термин для обозначения программных средств по организации ввода/вывода. А т. к. задачи ввода/вывода возникают практически в каждой программе и достаточно сложны в реализации, то разработчики стандартной библиотеки предоставляют прикладному программисту эти возможности посредством функций и специальных структур данных. При запуске программы в стартовом коде стандартной библиотеки создаются и инициализируются структуры данных типа FILE (структуры рассматриваются в гл. 9, тип FILE описан в справочной системе), которые по умолчанию связываются с устройствами ввода/вывода (ввод осуществляется с клавиатуры, вывод — на экран). Поэтому эти структуры данных принято называть *стандартными* потоками ввода/вывода. Именно стандартные потоки ввода/вывода по умолчанию используются функциями стандартной библиотеки для вывода на экран консольного приложения (в частности, функцией printf) и приема пользовательского ввода с клавиатуры (в частности, функцией scanf). Для явного использования этих структур данных программисту предоставляются соответствующие глобальные указатели типа FILE*:

- ❑ `stdout` — указатель на структуру данных, связанную с потоком вывода;
- ❑ `stdin` — аналогичный указатель для потока ввода.

Функции `printf()` и `scanf()` используются для форматированного вывода и являются классическими примерами функций с переменным числом параметров (объявлены в заголовочном файле `<stdio>`).

Форматированный вывод

Рассмотрим особенности функции `printf()` (и ее аналогов: `fprintf()` — для файлового вывода и `sprintf()` — для форматирования строк в памяти).

Прототип:

```
int printf(const char *,...); //выводит данные на экран. При переводе значений в строковое представление
```

использует однобайтовые символы

```
int wprintf(const wchar_t*,...); //при переводе значений в строковое
                                представление использует расширенные
                                символы (UNICODE)

int _tprintf(const TCHAR*,...); //макрос, который превращается
                                препроцессором в printf или wprintf
                                в зависимости от определенности
                                идентификатора _UNICODE. Это специфика
                                Microsoft
```

Специфика использования функции printf():

□ функция выполняет два действия:

- исходя из указанных параметров, формирует результирующую строку;
- выводит ее на экран;

□ программист, безусловно, может сам сформировать любую строку и воспользоваться другой функцией стандартной библиотеки (например, puts()) для ее вывода, но разработчики стандартной библиотеки решили облегчить задачу форматирования строк и предложили посредством printf() свой способ перевода в текстовый вид и форматирования любых значений базового типа. Функция printf() принимает один обязательный параметр — это строка, в которой программист может привести как текст, который требуется распечатать, так и специальным образом (посредством символа % и некоторой дополнительной информации) указать, что он хочет включить в результирующий текст строковое представление значения одной или нескольких переменных. А т. к. количество переменных в разных случаях разное, то список этих переменных задается посредством необязательных параметров функции. Функция анализирует обязательный параметр и, встретив символ %, вставляет вместо него в результирующий текст значение очередного необязательного параметра, форматируя его в соответствии с указаниями программиста. На рис. 8.10 представлена подобная ситуация. В результате будет выведено:

```
I want to print 1,    A;
```

□ наличие необязательного параметра обозначается символом % в единственном обязательном параметре — строке. За символом % программист должен указать, что он собирается выводить (целое, плавающее, строку и т. д.), и может уточнить, каким образом он хотел бы, чтобы это значение появилось на экране (сколько цифр после запятой, сколько знакомест отвести под значение, как позиционировать в поле вывода и т. д.);

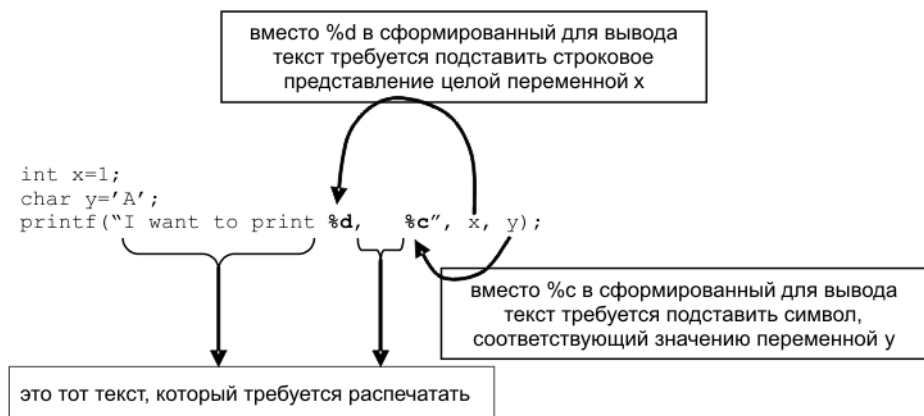


Рис. 8.10

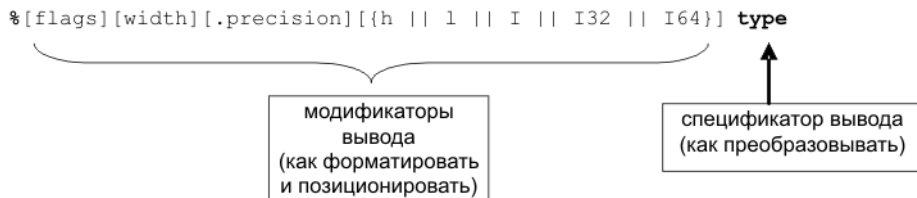


Рис. 8.11

□ управлять выводом можно с помощью дополнительных элементов, которые помещаются за символом % (рис. 8.11):

- обязательное поле *type*, называемое *спецификатором вывода* (спецификаторы указывают, какого типа очередной необязательный параметр, а следовательно, как следует его преобразовывать в строковое представление (табл. 8.5));
- совокупность необязательных полей, называемых *модификаторами вывода* (указывают, как форматировать и позиционировать при выводе (табл. 8.6)).

ЗАМЕЧАНИЕ 1

Если указанного посредством модификатора количества позиций для вывода значения недостаточно, то функция выделяет большее количество позиций — усечения не произойдет!

ЗАМЕЧАНИЕ 2

В табл. 8.6 даны далеко не все возможности форматирования — исчерпывающее описание имеется в справочной системе MSDN Library. Некоторые наиболее полезные возможности приведены в последующих примерах на стр. 282 и 283.

Таблица 8.5. Спецификаторы ввода/вывода

Спецификаторы	Пример задания аргумента	Будет выведено
%c	char x='A', y = 'B'; printf("%c\n%c", x, y);	A B
%d или %i	int x=1, y=2; printf("x=%d, y=%i", x, y);	x=1, y=2
%e или %E	double x=1.1; float y=2.2; printf("x=%e\ny=%E", x, y); [-]m.nnnnnn[+-]kkk (поле nnn — по умолчанию 6)	x=1.100000e+000 y=2.200000e+000
%f	float y=2.2; printf("y=%f", y); [-]m.nnnnnn (поле nnn — по умолчанию 6)	y=2.200000
%g или %G	используется наиболее подходящий (короткий) из форматов e и f (незначащие нули не печатаются) double x=1.1, y=2.222e2; printf("x=%g\ny=%G", x, y);	x=1.1 y=222.2
%o	восьмеричное представление int k = 10; printf("k=%o", k);	k=12
%p	значение адреса (указателя) в шестнадцатеричном виде printf("address=%p", &k);	address=значение_адреса
%s	печать строки char ar [] = "QWERTY"; printf("%s", ar);	QWERTY
%u	преобразование аргумента к целому беззнаковому в десятичном виде int m = -1; printf("%u", m);	4294967295
%x или %X	шестнадцатеричное представление int m = -1; printf("%x", m);	ffffffff

Таблица 8.6. Модификаторы ввода/вывода

Модификаторы вывода	Пример	Будет выведено *
Для строки <code>%[+-n]s</code> отводит минимум n позиций и в отведенном поле вывода позиционирует строку в зависимости от знака: минус — сдвигает влево, плюс (или знак опущен) — сдвигает вправо	<pre>char ar[] = "QWERTY"; printf("%10s", ar); printf("%+10s", ar); printf("%-10sw", ar);</pre>	^^^^QWERTY — в указанном диапазоне смещает вправо (выравнивается по правому краю) QWERTY^^^^^W — в указанном диапазоне смещает влево
Для вещественного числа — <code>%[+-n.m]f</code> или <code>%[+-n.m]e</code> или <code>%[+-n.m]g</code> где: n — общее количество знакомест для представления значения, включая точку, m — количество цифр после десятичной точки	<pre>double x=1.11111; float y=2.2222; printf("x=%10.2e, \ny=%10.2f", x, y); printf("x=%10.2e, \ny=%-10.2f", x, y);</pre>	$x=^{1.11}e+^{000}$, $y=^{2.22}$, $x=1.11e+^{000}$, $y=2.22^{2.22}$,
Для вещественного числа — <code>%[n]f</code> или <code>%[n]e</code> или <code>%[n]g</code> n — количество знакомест для преобразованного значения, а количество цифр после десятичной точки определяется по умолчанию	<pre>double x=1.1; float y=2.2; printf("x=%4g\nty=%5g", x, y); printf("x=%4f\nty=%5f", x, y);</pre>	$x=^{1.1}$ $y=^{2.2}$ $x=1.1^{00000}$ $y=2.2^{00000}$
Для целого числа — количество знакомест для расположения преобразованного значения <code>%[+-n]d</code> или <code>%[+-n]i</code>	<pre>int x = -1; printf("%5d\n", x);</pre>	^^^ -1

* — вместо пробела для демонстрации результата используем символ ^.

Пример 1.

Использование одного обязательного параметра функции `printf()`:

```
printf("one\ntwo\nthree"); //будет выведено:
                                one
                                two
                                three
```

Пример 2.

Использование спецификаторов вывода (листинг 8.12).

Листинг 8.12. Использование спецификаторов для преобразования значений

```
{
    int x=1;
    double y = 2.22;
    char ar[] = "ABC";
    //Функция принимает три необязательных параметра: x,y,ar
    printf("x=%d\ny=%f\nar=%s",x,y,ar); //вывод:
                                x=1
                                y=2.220000 - по умолчанию 6 символов после точки
                                ar=ABC
}
```

Пример 3.

Использование модификаторов вывода.

Без применения позиционирования текста столбцы таблицы съезжали бы в сторону в зависимости от значения. Красивый вывод таблицы целых чисел представлен в листинге 8.13.

Листинг 8.13. Использование модификаторов для позиционирования значений

```
int ar[3][4] = {1,35000,4,5000000,6};
for(int i=0; i<3; i++)
{
    for (int j=0; j<4; j++)
    {
        printf("ar[%d][%d]=%-7d  ",i,j,ar[i][j]);
    }
    printf("\n");
}
```

Пример 4.

Использование функции `printf()` для форматирования текста в памяти.

Эта функция имеет два обязательных параметра:

- адрес зарезервированной области памяти, куда будет помещена результирующая строка;
- строку, содержащую ключи форматирования (листинг 8.14).

Листинг 8.14. Форматирование строки в памяти

```
{  
    char ar[100]; //место для результирующей строки  
    char name[] = "Betty";  
    int age = 13;  
    printf( ar, "My dog %s is %d years old", name ,age);  
    //в массиве ar будет сформирована строка: My dog Betty is 13 years old  
}
```

Пример 5.

Использование функции `fprintf()` для форматированного вывода в файл.

Функция принимает два обязательных параметра:

- указатель на структуру типа `FILE` (`FILE` — это структура, которую предоставляет стандартная библиотека для обмена данными с устройством);
- строку, содержащую ключи форматирования.

Если для вывода на экран программист мог использовать по умолчанию стандартный поток вывода (`stdout`), то при записи в файл требуется явно указать, с каким файлом должен быть связан поток вывода и каким образом программист собирается этим файлом пользоваться (читать/писать). Ассоциировать поток вывода с файлом можно с помощью функции стандартной библиотеки `fopen()` (листинг 8.15).

Листинг 8.15. Форматированный вывод в файл

```
{  
    FILE* f = fopen("my.txt", "w"); //открываем файл my.txt  
                                   для записи ("w")  
    if(f) //если файл удалось открыть, будет возвращен  
        ненулевой указатель  
    {
```

```

    int x=1;
    char ar[] = "QWERTY";
    fprintf(f, "x:%d string:%s",x,ar); //пробел обязателен
    fclose(f);

//содержимое файла: x:1 string:QWERTY
}else ...//иначе файл открыть не удалось
}

```

Форматированный ввод

Семейство функций `scanf` предназначено для преобразований из строкового представления в значение, указанное программистом посредством спецификатора ввода.

```

int scanf(const char *,...); //вариант для однобайтовых символов
int wscanf(const wchar_t*,...); //аналог для расширенных символов UNICODE
int _tscanf(const TCHAR*,...); //макрос, предоставляемый Visual C++,
                                который превращается в scanf()
                                или wscanf() в зависимости от
                                определенности имени _UNICODE

```

Специфика:

- ❑ функция `scanf()` принимает пользовательский ввод с клавиатуры и использует для ввода стандартный поток ввода — `stdin`;
- ❑ `scanf()` принимает один обязательный параметр — строку, посредством которой интерпретируются вводимые значения, и любое количество необязательных параметров. Символ `%` в форматной строке говорит о том, что нужно считать очередное значение.

Спецификация формата:

`%[*][width][{h || l || I || I32 || I64}] type`

- ❑ разделителями при вводе являются: пробел, табуляция, перевод строки;
- ❑ при задании необязательных параметров принципиальным является указание адреса переменной, по которому будет помещен результат перевода из строкового представления! Так как компилятор не может проконтролировать тип необязательного параметра, то сообщения об ошибке в случае некорректного значения он не выдаст, зато, скорее всего, возникнет ошибка во время выполнения;
- ❑ с помощью спецификатора формата `*` можно предписать функции проигнорировать очередное введенное значение. В таком случае в списке необязательных параметров соответствующий параметр просто отсутствует.

Пример 1.

Использование спецификаторов ввода для преобразования из строкового представления в значение требуемого типа (листинг 8.16).

Листинг 8.16. Использование спецификаторов ввода

```
int x;
float y ;
double z ;
char ar[5] ;
scanf("%d %f %lf %s",&x,&y,&z,ar); //введенный пользователем текст будет
                                   преобразован в соответствии с
                                   указанным после % спецификатором
                                   ввода, а полученное значение будет
                                   записано по указанному в качестве
                                   необязательного параметра адресу
```

Пример 2.

Использование спецификатора формата *:

```
scanf("%d %*f %s",&x,ar); /* указывает, что соответствующее значение
                           нужно считать из потока ввода, но не
                           сохранять в переменной (поэтому
                           соответствующий адрес в списке
                           необязательных параметров отсутствует)
```

Пример 3.

Если пользователь набрал: <n=1>, то n= нужно считать и проигнорировать, а 1 перевести в целое представление. Для этого нужно явно в строке формата привести тот текст, который функция должна проигнорировать:

```
int n;
scanf("n=%d",&n); //если набираем n=1, то n= игнорируется, а 1 вводится
                  в переменную n
```

Пример 4.

Если пользователь вводит строку, то всегда есть шанс, что он введет больше символов, чем программист зарезервировал памяти. Поэтому нужно уметь защищаться от таких ситуаций с помощью модификаторов ввода:

```
char ar[5]; //зарезервировано только 5 байтов!
int n=scanf("%4s ",ar); //поэтому позволяем ввести только 5 символов,
                        включая завершающий 0
```

```
fflush(stdin); //а если строка оказывается длиннее, то не считанный
                остаток остается в буфере и будет считан при следующем
                вызове scanf(), поэтому нужно очистить буфер потока
                ввода функцией fflush()
```

Пример 5.

Файловый ввод с помощью функции `fscanf()`.

Считаем из файла `my.txt` ту информацию, которую записали туда в листинге 8.15. Листинг 8.17 демонстрирует считывание текстовой информации из файла и преобразование значащей информации в требуемый вид.

Листинг 8.17. Форматированный файловый ввод

```
{
FILE* f = fopen("my.txt", "r"); //открываем файл для чтения
if(f) //если удалось открыть файл
{
    //резервируем память для вводимых значений
    int x;
    char ar[10];
    //считываем те данные, которые нас интересуют, а
    //вспомогательную информацию игнорируем
    fscanf(f, "x:%d string:%s", &x, ar); //x: считывается, но
                                         игнорируется, аналогично
                                         string: считывается, но
                                         игнорируется

    //в результате: x=1, ar = "QWERTY"
    fclose(f);
} else...//иначе файл открыть не удалось
}
```

Пример 6.

Форматирование строк в памяти посредством функции `sscanf()` (листинг 8.18).

Листинг 8.18. Считывание данных из строки в памяти

```
{
char s[] = "1 2 3 4"; //строка, из которой будем считывать очередную
                       порцию текста до разделителя (пробела) и
                       преобразовывать в указанные нами значения
```

```
char buf[10];
char c;
int i;
float f;
scanf( s, "%s %c %d %f", buf,&c,&i,&f );
//В результате: buf="1", c='2', i=3, f=4.0
}
```

8.3. Возвращаемое значение

8.3.1. Виды возвращаемых значений и механизмы их формирования

Функция может возвращать (рис. 8.12):

- а) значение одного из базовых типов,
- б) объект пользовательского типа;
- в) адрес.

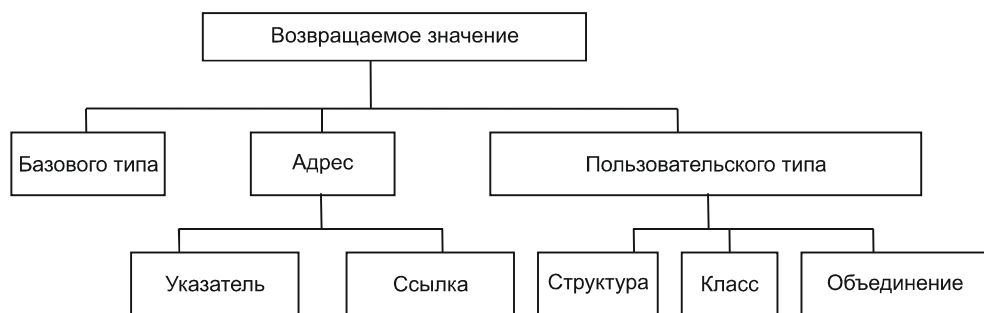


Рис. 8.12

Данные разного типа возвращаются по-разному:

- ❑ значения из а) и в), помещающиеся в регистр, возвращаются на регистре (для x86 процессоров используется регистр EAX);
- ❑ длинные целые (**long long**) возвращаются на паре регистров (EAX, EDX);
- ❑ значения типа **float** и **double** возвращаются в стеке плавающей точки посредством низкоуровневых команд: **fld** (load real — занести в стек плавающей точки), **fstp** (store real — извлечь из стека);

- массивы по значению не возвращаются, можно вернуть только адрес начала массива;
- механизм возвращения структур, объединений или классов более сложен и будет рассмотрен в *разд. 9.12.2*.

Пример 1.

Возвращаемое функцией значение можно присвоить переменной подходящего типа или использовать другими способами.

```
bool Func();
int main()
{
    bool b = Func(); //возвращаемое функцией значение будет присвоено
                     //переменной b
    //или
    if(Func()){...} //возвращаемое значение используется для формирования
                   //условия
}
```

Пример 2.

Поиск минимального значения в массиве (листинг 8.19).

Листинг 8.19. Возвращение указателя

```
//файл 1.cpp
int* Min(int ar[ ], unsigned int n)
{
    int* p = ar; //предполагаем, что минимальным является нулевой
                 //элемент массива и направляем на него указатель
    for(int i = 0; i < n; i++)
    {
        if(ar[i] < *p) //а если i-тое значение оказалось меньше
        {
            p = &ar[i]; //перенаправляем на него указатель
        }
    }
    return p; //в результате в переменной p будет адрес элемента
             //массива с самым маленьким значением
}
//файл 1.h
int* Min(int ar[ ], unsigned int n);
```

```
//файл main.cpp
#include <stdio>
#include "1.h"
int main()
{
    int mas[] = {3, 6, -1, 0, -10, 55};
    int* pMin = Min(mas, sizeof(mas)/sizeof(mas[0]));
    printf("min=%d", *pMin);
    return 0;
}
```

Бывают случаи, когда вы не используете возвращаемое функцией значение (вам достаточно действий, которые осуществляет функция), тогда совершенно не обязательно заводить переменную, которая будет ненужное возвращаемое значение принимать. В этом случае в вызываемой функции возвращаемое значение формируется, а в вызывающей не принимается. Например:

```
bool Func();
int main()
{
    Func();
}
```

Если функция возвращает адрес, то ее вызов можно использовать как lvalue (например, слева от присваивания), в этом случае значение rvalue будет помещено в память по адресу, возвращенному функцией:

```
int& f1()
{
    static int n;
    return n;
}
int* f1()
{
    static int n;
    return &n;
}
int main()
{
    f1() = 1; //по возвращаемому адресу занести новое значение
    *f1() = 1; //аналогично
}
```



Подумайте, почему в функциях `f1` и `f2` переменная `n` объявлена с ключевым словом **static**, и сверьте ваши предположения с предостережением в разд. 8.3.2.

8.3.2. Проблемы при возвращении ссылки или указателя

При возвращении ссылки или указателя из функции объект, на который указывает возвращаемое значение, должен существовать после возврата из функции. Пример, приведенный в листинге 8.19, корректен, т. к. возвращает адрес элемента массива, который, безусловно, существует после возврата из функции. При написании функций, возвращающих указатель или ссылку, начинающий программист часто допускает ошибки, продемонстрированные в листинге 8.20.

Листинг 8.20. Примеры некорректных возвращаемых значений

```
int* f1(int n)
{
    int nN = n*5;
    ...
    return &nN; //никогда так не делайте! Ваша функция вернет
                указатель на область памяти в стеке, которая после
                возвращения из функции может быть задействована
                компилятором для других целей
}

int& f2(int n)
{
    int nN=n*5;
    ...
    return nN; //и так тоже никогда не делайте! Ваша функция
                по-прежнему возвращает адрес временной переменной!
}

int main()
{
    int z = f2(3); //переменной z будет присвоено значение по адресу,
                  возвращенному из функции. На первый взгляд
                  проблем нет, и в данном примере z=15,
                  как и ожидалось
}
```



Попробуйте спрогнозировать и объяснить результат вычисления выражения:

$$z = f2(1) + f2(2) + f2(3);$$

А сейчас рассмотрим подробнее пример, в котором станет очевидным, почему нельзя возвращать адреса локальных (временных) объектов (листинг 8.21 и рис. 8.13).

Листинг 8.21. Пример, из которого вполне очевидно, что так делать не годится

```
char* f1()
{
    char ar[] = "ABC"; //это локальный массив, для него компилятор
                        выделяет память в стековом кадре функции
                        только на время ее выполнения
    ...
    return ar; //возвращается адрес локального массива!
}

void f2() //функция ничего не возвращает
{
    int tmp = 0x44332211; //но использует локальную переменную,
                        для которой компилятор выделяет память
                        в стековом кадре
    ...
}

int main()
{
    char* p = f1(); //при вызове функции компилятор сформировал
                    стековый кадр, как показано на рис. 8.13.
                    Возвращенный адрес запомнили в переменной p.
                    После возврата из функции компилятор считает,
                    что эта область стека свободна и использует
                    ее при вызове следующей функции
    f2(); //вызываем другую функцию. Для ее локальных переменных
                    компилятор использует ту же самую область стека и
                    располагает в этой области совершенно другое значение
}
```



А теперь попробуем воспользоваться указателем `p`, рассчитывая, что он указывает на нужную нам строку. Попробуйте догадаться, что будет напечатано?

```
printf("%c", p[0]);
```

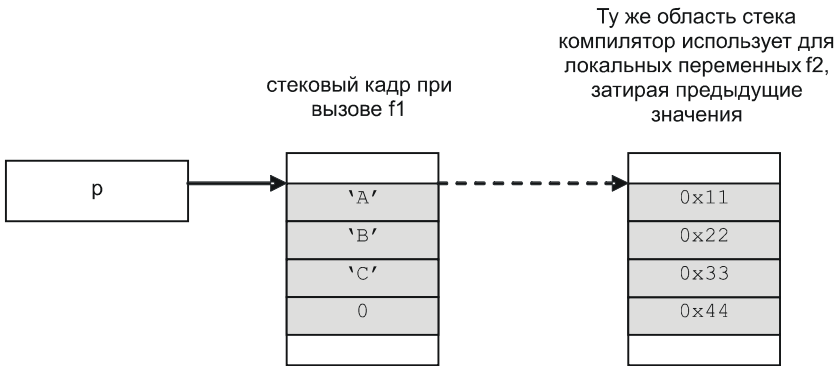


Рис. 8.13

ПРЕДОСТЕРЕЖЕНИЕ

Нельзя возвращать адреса локальных объектов! Ситуация будет еще хуже, если локальный объект — пользовательского типа (класса) (по закрывающей скобке функции для такого объекта компилятор будет автоматически вызывать деструктор).

Возвращать можно адреса таких объектов, которые гарантированно существуют после возврата из функции:

- ❑ указатель или ссылку на объект, который располагается в стековом кадре вызвавшей функции (см. листинг 8.19);
- ❑ указатель или ссылку на объект со статическим временем существования;
- ❑ указатель на строковый литерал (для него память будет выделена на все время выполнения программы);
- ❑ указатель на динамически созданный объект. Следует отметить, что в этом случае возникает другая проблема — вызывающая функция должна освободить память, т. е. программист должен, как минимум, знать о том, что функция возвращает указатель на динамически захваченный блок памяти.

8.4. Ключевое слово `const` и функции

Применительно к функциям ключевое слово `const` может относиться как к принимаемым функцией параметрам, так и к возвращаемому значению.

8.4.1. Передача функции константных параметров

Если параметр объявлен с ключевым словом **const**, то это является гарантией того, что компилятор не позволит модифицировать такой параметр в теле функции. Обычно константные параметры используют в тех случаях, когда речь идет о передаче в функцию адреса объекта, например:

```
void SomeFunc(const int* pn);
```

Такое объявление параметра `pn` обеспечивает неизменяемость объекта, на который указывает `pn`, функцией `SomeFunc()`. Объявление функций с константными параметрами позволяет компилятору блокировать нежелательные побочные эффекты вызова функций (когда модификация значения по передаваемому адресу не желательна). Пример приведен в листинге 8.22.

Листинг 8.22. Передача функции константных параметров

```
void Func(const char* pc) //объявление параметра const гарантирует, что
                           в области видимости функции значение самого
                           объекта не изменится
{
    /*pc = 'A'; //ошибка: запрещено модифицировать константу
}
int main()
{
    Func("1234");
}
```

ЗАМЕЧАНИЕ

При объявлении константных параметров ключевое слово **const** должно фигурировать как в объявлении функции, так и при определении. Иначе компилятор решит, что это две разные функции и на этапе компоновки будет выдана ошибка.

Например:

```
//1.h
void f(const int* k); //прототип функции

//1.cpp
void f(int* k) //определение функции (ключевое слово const опущено)
{
    ...
}
//main.cpp
```

```
int main()      {
    int n = 1;
    f(&n);
}
```

Компоновщиком будет выдано сообщение об ошибке: не найдена реализация функции **void f(const int* k)**;

Нет необходимости объявлять с модификатором **const** параметры, которые передаются по значению, т. к. функция получает только копии исходных переменных (см. *разд. 8.2.1*), а вызывающей функции все равно — модифицирует вызванная функция свою копию или нет.

```
void f(const int n) //необязательно объявлять такой параметр const
{...}
```

8.4.2. Возвращение функцией константных значений

Возвращение константных значений, пожалуй, имеет смысл только при возврате адреса, т. к. если функция возвращает просто значение, то вызов функции в любом случае можно использовать лишь справа от знака равенства. Пример использования ключевого слова **const** приведен в листинге 8.23.

Листинг 8.23. Возвращение константных значений

```
char* GetName1() //тип возвращаемого значения задан некорректно, т. к.
                //посредством возвращаемого адреса компилятор позволит
                //модифицировать значение, что скорее всего приведет к
                //ошибке времени выполнения (см. разд. 3.2.4)
{return "Name1";}

const char* GetName2() //корректное возвращаемое значение, т. к.
                       //компилятор предотвратит модификацию значения по
                       //возвращаемому адресу
{return "Name2";}

int main()
{
    char* p1 = GetName1();
    p1[2] = 'A'; //компилятор сообщения об ошибке не выдаст, т. к.
                //синтаксически все корректно, но при выполнении
```

программы в защищенном режиме произойдет ошибка
времени выполнения

```
char tmp = GetName1()[0]; //OK - получили код символа N
GetName1()[0]='A'; //ошибка времени выполнения
```

```
//но!

//char* p2 = GetName2(); //ошибка: см. разд. 6.1.8
const char* pc2 = GetName2(); //OK
//pc2[2] = 'A'; //ошибка: указываемое значение - константа
char tmp = GetName2()[0]; //OK - получили код символа N
//GetName2()[0]='A'; //ошибка: возвращается указатель на константу
}
```



Подумайте, имеет ли смысл и корректно ли объявлена такая функция?

```
char* const GetName3()
{return "Name3";} //функция возвращает константный указатель
```

8.5. Перегрузка имен функций

Одной из особенностей C++ является возможность перегрузки имен функций, т. е. использования одного и того же имени для нескольких разных функций. Техника перегрузки неявно используется компилятором для базовых операций C/C++. Например, существует только одно имя для операции сложения (+), но вы используете это имя для сложения целых чисел, чисел с плавающей точкой, прибавляете целое к указателю, может быть, даже не задумываясь о том, что компилятор при этом генерирует совершенно разные машинные инструкции.

Идея перегрузки операций легко распространяется на функции, определяемые пользователем. Иногда такой подход бывает очень удобен, когда функции выполняют аналогичные по семантике действия (например, сложение). Компилятор различает такие функции по числу и/или типу параметров. При каждом конкретном вызове перегруженной функции компилятор должен определить, какую из функций с данным именем вызвать. Цель компилятора состоит в том, чтобы использовать функцию с наиболее подходящими аргументами или выдать сообщение об ошибке, если подходящей функции не найдено.

ЗАМЕЧАНИЕ

Механизм перегрузки имен функций возможен ввиду того, что функции с одним и тем же именем, но с разным количеством и/или разными типами параметров компилятор декорирует по-разному (см. разд. 3.6.1 и 8.1.5).

Пример 1.

Если такой возможности, как перегрузка имен функций, нет (например, в языке ANSI Си), то программист должен давать функциям, выполняющим близкие по смыслу действия, уникальные имена (листинг 8.24).

Листинг 8.24. Без перегрузки имен функций все функции должны иметь уникальные имена

```
int MaxInt(int x, int y) //функция находит максимальное значение из двух
                        параметров типа int
{ return (x>y) ? x : y; }

double MaxDouble(double x, double y) //тоже находит максимум, только
                                     принимает параметры другого типа, а это означает,
                                     что компилятор должен сгенерировать совершенно
                                     разные низкоуровневые инструкции при действиях
                                     с этими значениями (несмотря на то, что текст
                                     на языке высокого уровня выглядит одинаково)
{ return (x>y) ? x : y; }

int main()
{
    int i = MaxInt( 12, 8 );
    double d = MaxDouble( 32.9, 17.4 );
}
```

Пример 2.

В языке C++ программист может давать разным функциям одно и то же имя. В зависимости от количества и/или типа аргументов компилятор сам сгенерирует вызов нужной функции (листинг 8.25).

Листинг 8.25. Перегрузка имен функций

```
int Max(int x, int y)
{ return (x>y) ? x : y; }

double Max(double x, double y)
{ return (x>y) ? x : y; }

//Следующая функция по смыслу тоже претендует на имя Max:
int Max(int ar[], int n) //функция находит максимальный из элементов
                        массива
{
```

```

    int max = ar[0];
    for(int i=1; i<n; i++)
    {
        if(ar[i]>max) max=ar[i];
    }
    return max;
}

int main()
{
    int m = Max( 12, 8 ); //вызов Max(int, int)
    double d = Max( 32.9, 17.4 ); //вызов Max(double, double)
    //double dd = Max( 1.1 , 2 ); //ошибка: компилятор считает, что
                                   не может найти однозначного
                                   соответствия

    int array[] = {5, -3, 0, 10, 2};
    m = Max(array, sizeof(array)/sizeof(array[0]) );
                                   //вызов Max(int [], int)
}

```

При перегрузке имен функций действуют следующие ограничения:

- ❑ не могут перегружаться функции, имеющие совпадающие тип и число аргументов, но разные типы возвращаемых значений. Например:

```

void f(int);
double f(int);
int main()
{
    f(5); //компилятор не понимает, какую функцию хочет вызвать
          программист, т. к. возвращаемое значение принимать
          необязательно (см. разд. 8.3.1)
}

```

- ❑ не могут перегружаться функции, имеющие неявно совпадающие типы аргументов, т. к. компилятор при синтаксическом анализе не сможет разобраться по указанному параметру, какую вызывать. Например:

```

void f(int);
void f(int&);
int main()
{
    int n=5;
}

```

```
f(n); //компилятор не понимает, какую функцию хочет вызвать
      программист, т. к. использование ссылки в качестве
      параметра синтаксически выглядит так же, как передача
      параметра по значению (см. разд. 8.2.2)
}
```

Механизм перегрузки основывается на относительно сложном наборе правил для неявного преобразования типов аргументов, поэтому в некоторых случаях (даже если компилятор не выдает ошибки) на первый взгляд неочевидно, какая из функций будет вызвана. Если есть сомнения, то чтобы избежать ошибок при компиляции или неприятностей при неявном преобразовании типов аргументов, пользуйтесь явным преобразованием для точного соответствия типов параметров:

```
double dd = Max( 1.1 , static_cast<double>(2) );
              //вызов Max(double, double)
```

8.5.1. Возможные конфликты при использовании параметров по умолчанию

При перегрузке имен функций и одновременном использовании значений параметров по умолчанию может возникнуть неоднозначная ситуация (листинг 8.26).

Листинг 8.26. Пример появления неоднозначности при использовании параметров по умолчанию

```
void f(int x, int y=0);
void f(int);
int main()
{
    a(1, 2); //ОК – компилятор однозначно понимает, что нужно
              вызвать функцию f, которая принимает два параметра
    f(5);     //ошибка: компилятор не может определить, какую
              из функций следует вызвать
}
```

8.6. Рекурсивные функции

Рекурсивные функции — это функции, вызывающие сами себя. Рекурсивные вычисления выполняются повторным выполнением одного и того же кода с разными наборами данных. Каждое выполнение тела функции имеет свою область стека (стековый кадр stack frame) для параметров и локальных пере-

менных. Поэтому каждое вхождение в функцию можно рассматривать как абсолютно не зависящее от других вхождений.

Достоинством рекурсивных функций является возможность создания компактного кода (особенно при использовании сложных типов данных).

Недостатками рекурсивных вычислений являются: затраты времени на вызов функции и передачу ей копий параметров, а также затраты памяти (стека) для организации каждого вложенного вызова.

ВАЖНО!

Рекурсия возможна, т. к. каждый вызов работает со своей копией данных!

Специфика рекурсивных функций:

- ❑ программист должен обеспечить внутри рекурсивной функции не только анализ, но и обязательное выполнение условия, при котором произойдет выход из рекурсии (иначе случится бесконечная рекурсия, т. е. зависание программы);
- ❑ по мере возможности следует избегать использования в рекурсивной функции локальных переменных, т. к. это увеличивает размер стекового кадра для каждого вложенного вызова.

Без рекурсии в большинстве случаев можно обойтись.

Пример — вычисление факториала без использования рекурсии:

```
{  
    int n=5;  
    int res=1;  
    for(int i=n;i>1;i--) res*=i;  
}
```

В листинге 8.27 приведен классический пример использования рекурсии для вычисления факториала. Хотя код выполняется один и тот же, но каждый вложенный вызов использует свою область стека (рис. 8.14).

Листинг 8.27. Использование рекурсии при вычислении факториала

//Вариант 1. Используется вспомогательная локальная переменная x

```
int F(int n)  
{  
    if(n<=1)return 1; //обеспечили выход из рекурсии  
    else  
    {  
        int x = F(n-1); //рекурсивный вызов функции  
                        с параметром (n-1)
```

```
        return n*x;
    }
}
int main()
{
    int n=5;
    int res = F(n);
}
```

//Вариант 2. на рис. 8.14 очевидно, что при каждом вызове функции компилятор должен отвести в стековом кадре память под локальную переменную X. Это как раз тот случай, когда без этой локальной переменной можно обойтись

```
int F(int n)
{
    if(n<=1)return 1; //обеспечили выход из рекурсии
    else
    {
        return n * F(n-1);
    }
}
```

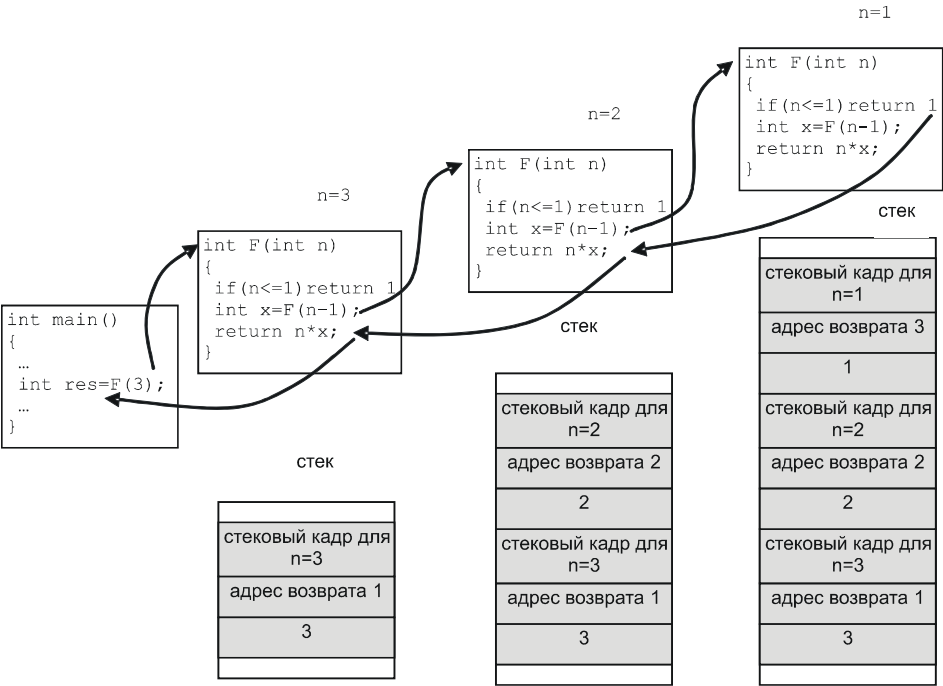


Рис. 8.14

ЗАМЕЧАНИЕ

Несмотря на большие затраты, преимущество использования рекурсивных функций начинает сказываться при работе со сложными структурами данных — деревьями, стеками. Есть мнение, что рекурсия — это краеугольный камень сложных алгоритмов.

8.7. Указатель на функцию

Сочетая основные типы данных в языке C/C++, можно образовать неограниченное число более сложных производных типов. В частности, одним из таких производных типов является указатель на функцию. Указатели на функции можно использовать как аргументы при вызове других функций, хранить в массиве и находить им применение другим способом.

8.7.1. Определение указателя на функцию

Тип указателя на функцию зависит от количества и типа параметров, а также от типа возвращаемого значения. Пусть требуется определить указатель на функцию, которая принимает один параметр типа **double** и возвращает значение типа **int**. На рис. 8.15 приведено определение переменной с именем **Func**, которая является указателем требуемого типа. Компилятор, встречая такое определение, выделяет память для переменной-указателя и по общим правилам может такую переменную неявно проинициализировать нулем в зависимости от контекста определения.

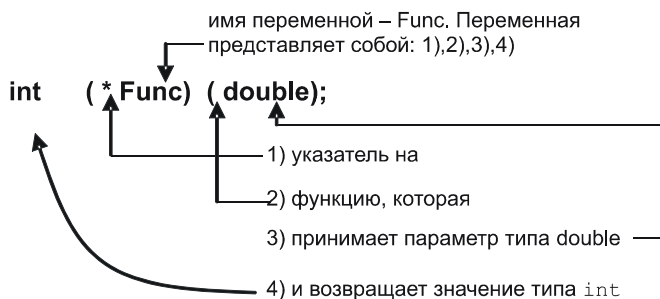


Рис. 8.15

ЗАМЕЧАНИЕ

Скобки вокруг `(*Func)` обязательны, поскольку объявление `int *Func(double);` означает прототип функции, возвращающей указатель `int*`.

Если функция принимает несколько параметров, то их типы указываются в скобках через запятую так же, как при объявлении функции:

```
char* (*pf)(char*, int); //указатель на функцию, которая принимает два
                           параметра (указатель char* и целое)
                           и возвращает указатель char*
```

8.7.2. Инициализация указателя на функцию

Пусть в программе определено несколько функций, прототипы которых выглядят одинаково. Требуется определить указатель на функцию такого вида и направить указатель на одну из реально существующих функций (листинг 8.28).

Листинг 8.28. Инициализация указателя на функцию

```
int RealFunc(void);
int OtherFunc(void);
int main()
{
    ...

    int (*pf1)(void)= &RealFunc; //определение и инициализация.
                                   Компилятор заносит в переменную-указатель
                                   адрес начала функции

    //Так как в языке Си имя функции эквивалентно указателю, то можно и так:
    int (*pf2)(void)= RealFunc; //выражения эквивалентны
                                   //А теперь ничто не мешает перенаправить указатель на
                                   другую подходящую функцию:
    pf1 = OtherFunc; //компилятор занесет в ту же переменную
                                   другой адрес
}
```

8.7.3. Вызов функции посредством указателя

Пусть в программе используется функция, прототип которой:

```
int f(const char*, double);
```

Определим указатель подходящего типа и вызовем посредством этого указателя функцию `f()`. Вызвать функцию посредством указателя можно двумя способами:

- синтаксически любой указатель можно разыменовывать. Формально разыменованный указатель на функцию является синонимом имени функции.

Последовательность, в которой компилятор разбирает выражение, представлена на рис. 8.16.

```
{
    int (*pf)(const char*, double) = &f;
    int res = (*pf)( "ABC", 1.1);
}
```

□ вспомнив о том, что в C/C++ имя функции эквивалентно указателю, можно написать и так:

```
pf("ABC", 1.1);
```

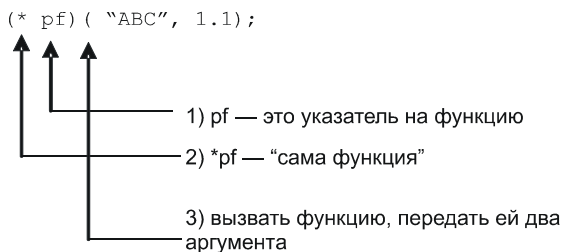


Рис. 8.16

8.7.4. Использование указателей на функции в качестве параметров

Этот прием довольно часто используется как стандартной библиотекой, так и системными функциями Windows для передачи библиотечной функции указателя на вашу пользовательскую функцию. Таким образом, библиотечная функция посредством полученного указателя может выполнять специфические для вашей программы действия. Например, функция стандартной библиотеки `qsort()` способна сортировать массивы элементов любого типа, т. к. в качестве одного из параметров она принимает указатель на пользовательскую функцию, в которой программист объясняет, каким образом следует сравнивать два элемента своего массива, а `qsort()` в соответствующие моменты просто вызывает по указателю пользовательскую функцию:

//Прототип библиотечной функции `qsort()` находится в заголовочном файле `<stdlib.h>`

```
void qsort(
    void *base,    //указатель на начало массива
    size_t num,    //количество элементов в массиве
```



```

size_t width, //размер элемента массива в байтах
int (__cdecl *compare )(const void *, const void *) //пользовательская
                               функция сравнения двух элементов массива
);

```

Пример использования библиотечной функции `qsort()` для сортировки объектов любого типа представлен в листинге 8.29.

Листинг 8.29. Пример использования библиотечной функции `qsort()`

```

#include <stdlib> //прототип функции qsort()
#include <string> //прототип функции strcmp()
//функции сравнения, предоставляемые программистом, должны принимать
    два const void * указателя и возвращать значение:
                                >0 – если *p1>*p2,
                                =0 – если *p1=*p2,
                                <0 – если *p1<*p2

//функция сравнения двух значений типа int предназначена для сортировки
    по возрастанию значений
int cmp_int_accended(const void* p1, const void* p2)
{
    const int* ptmp1 = static_cast<const int*>(p1); //программист знает,
                                что функция на самом деле принимает
                                два адреса типа int*, поэтому приводим
                                тип void*-указателя к требуемому типу
    const int* ptmp2 = static_cast<const int*>(p2);
    return (*ptmp1 - *ptmp2);
}

//функция сравнения двух значений типа int предназначена для сортировки
    по убыванию значений
int cmp_int_descended(const void* p1, const void* p2)
{
    const int* ptmp1 = static_cast<const int*>(p1);
    const int* ptmp2 = static_cast<const int*>(p2);
    return (*ptmp2 - *ptmp1);
}

//функция сравнения двух значений типа char* предназначена для сортировки
    строк по возрастанию в лексиграфическом порядке
int cmp_str(const void* p1, const void* p2)

```

```
{
const char** ptmp1 = static_cast<const char**>(const_cast<void*>(p1));
    //в этом случае преобразование хитрее, т. к. привести
    непосредственно const void*-указатель к const char**
    компилятор не позволит, поэтому сначала нужно снять
    константность с void*-указателя
const char** ptmp2 = static_cast<const char**>(const_cast<void*>(p2));
return strcmp(*ptmp1,*ptmp2); //теперь строки можно лексиграфически
    сравнивать с помощью функции стандартной
    библиотеки - strcmp()
}
int main()
{
    int arn[] = {5, -6, 10, 0, -1};
    //сортируем массив по возрастанию
    qsort(arn,sizeof(arn)/sizeof(int),sizeof(int), cmp_int_accended);
    //теперь тот же самый массив по убыванию
    qsort(arn,sizeof(arn)/sizeof(int),sizeof(int), cmp_int_descended);
    //а теперь массив указателей на строки в лексиграфическом порядке по
    возрастанию
    const char* arstr[] = {"ten", "green", "bottles", "hanging", "on",
        "the", "wall"};
    qsort(arstr,sizeof(arstr)/sizeof(char*),sizeof(char*), cmp_str);
    //в результате указатели на строки будут располагаться в массиве в
    следующем порядке: "bottles", "green", "hanging", "on", "ten", "the",
    "wall"
}
```

ЗАМЕЧАНИЕ

Использование указателей на функции в качестве параметров не является прерогативой библиотечных функций. Программист может применять этот прием при разработке собственных универсальных функций.

8.7.5. Использование указателя на функцию в качестве возвращаемого значения

Этот прием также часто используется библиотечными функциями (например, при подмене библиотечных обработчиков аварийных ситуаций — `_set_new_handler()`).

Приведем более простой пример (листинг 8.30). Пусть в зависимости от обстоятельств нужно вызывать с одним и тем же параметром одну из функций

`cos()`, `sin()` и т. д. Причем выяснение того, какую функцию следует вызвать, находится в одной функции — `f()`, а вызов должен быть в другой функции — `main()`.

Листинг 8.30. Пример использования указателя на функцию в качестве возвращаемого значения

```
#include <cmath> //прототипы всех тригонометрических функций.  
  
    Все тригонометрические функции типа sin(), cos()  
    принимают один параметр типа double и возвращают  
    значение типа double  
  
double (* f(void)) (double) //f — это функция, которая не принимает  
    параметров, а возвращает указатель на  
    функцию, принимающую параметр типа double  
    и возвращающую значение типа double  
  
{  
    if(условие) return sin;  
    else return cos;  
}  
  
int main()  
{  
    double (*pf)(double) = f(); //объявили подходящий указатель  
        и проинициализировали его  
        возвращаемым функцией значением:  
        это может быть адрес функции sin()  
        или адрес функции cos()  
  
    double res = pf(.9); //а теперь вызвали посредством указателя  
        одну из функций  
  
}
```

8.7.6. Массивы указателей на функции

Совокупность указателей на функции одного вида можно объединить и хранить (например, в массиве). Используем для примера тригонометрические функции стандартной библиотеки `sin()`, `cos()`, `tan()` и т. д. Прототипы таких функций по своей структуре выглядят одинаково, поэтому указатели на них можно сгруппировать в массиве. Пример приведен в листинге 8.31, сложное объявление поясняется рис. 8.17.

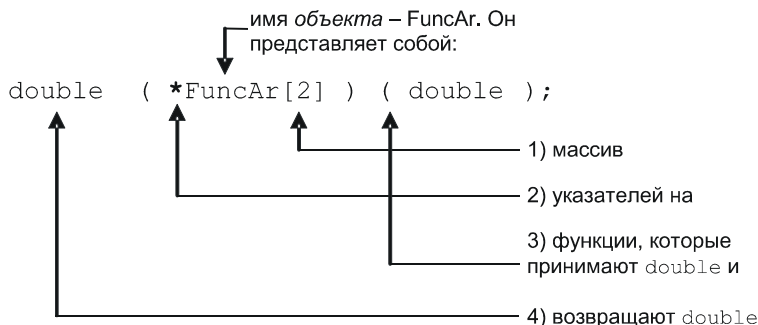


Рис. 8.17

Листинг 8.31. Использование массива указателей на функции

```
#include <cmath> //прототипы тригонометрических функций
#include <stdio.h> //прототипы функций printf() и scanf()
int main()
{
    double (*FuncAr[])(double) = {sin, cos, tan}; //FuncAr – это
                                                    массив указателей на функции (рис. 8.17),
                                                    которые принимают double и возвращают double.
                                                    По общим правилам при использовании списка
                                                    инициализаторов, размерность массива можно
                                                    оставить открытой, компилятор сам подсчитает,
                                                    сколько нужно зарезервировать памяти
    int n; //в этой переменной пользователь сформирует свой выбор:
           какую из функций он хочет вызвать
    double res=0; //здесь получим результат выбранной пользователем
                  операции
    printf("sin (0), cos (1), tan (2)\n"); //выводим пользователю
                                           приглашение
    scanf("%d", &n); //принимаем пользовательский ввод
    if( (n>=0) && (n<sizeof(ar)/sizeof(double (*)(double)) ) ) //а
        если пользователь несмотря на подсказку ввел что-нибудь
        другое, ставим защиту
    {
        res=FuncAr[n](0.5); //вызов требуемой функции
    }
}
```

```
//или так:
//          res>(* FuncAr [n])(0.5);
//      }
//      ...
}
```

8.8. Ключевое слово *typedef* и сложные указатели

Как вы уже могли заметить, далеко не сразу можно догадаться, каким образом следует объявлять сложные указатели. В таких ситуациях программисту достаточно вспомнить про ключевое слово **typedef** (см. разд. 3.5.1) и ввести полезный псевдоним типа.

8.8.1. Ключевое слово *typedef* и указатели на функции

Введем псевдоним типа для указателя на функцию, которая принимает **double** и возвращает **double**:

```
typedef double (*PF)(double); //с этого момента PF является псевдонимом
                               типа для указателя на функцию требуемого вида
```

Теперь объявление и использование указателей такого типа выглядит проще и привычнее:

```
PF f[] = { sin, cos, tan }; //PF – тип элемента массива
```

8.8.2. Функции, возвращающие сложные указатели

В зависимости от типа возвращаемого значения такие функции подразделяются на два вида:

- функции, у которых возвращаемое значение является указателем на массив;
- функции, у которых возвращаемое значение является указателем на функцию.

Возвращаемое значение является указателем на массив

Если внутри функции создается одномерный массив, то тип возвращаемого значения указать просто:

```
int* f()
{
    static int ar[10]; //хотя область видимости у такого массива
```

локальная, время жизни — статическое, поэтому
возвращать адрес такого массива можно
(см. разд. 8.3.2)

```
...
return ar;
```

```
}
```

Гораздо сложнее выглядит объявление функции, которая возвращает указатель на многомерный массив (листинг 8.32). Пояснение к такому сложному объявлению дано на рис. 8.18.

Листинг 8.32. Функция, возвращающая указатель на многомерный массив

```
int (* f(void))[20]
{
    static int ar[10][20];
    ...
    return ar;
}

int main()
{
    int (* p)[20] = f();
    p[1][1] = 1; //помня о связи массивов и указателей, пользоваться
                //возвращенным значением можно точно так же, как
                //именем двухмерного массива
}
```

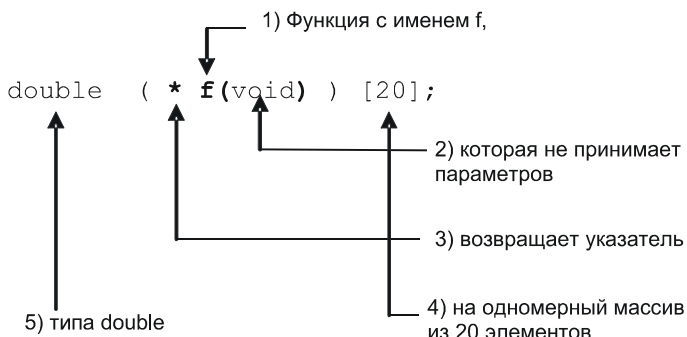


Рис. 8.18

Поскольку возвращаемое значение имеет достаточно сложный для интерпретации тип, то можно ввести псевдоним, который сделает объявление проще для понимания (листинг 8.33).

Листинг 8.33. Использование псевдонима для указателя на многомерный массив

```
typedef int (*PAR)[20]; //PAR - это псевдоним указателя на одномерный
                        массив из 20 элементов типа int

PAR f() //используем псевдоним в качестве типа возвращаемого значения
{
    static int ar[10][20];
    return ar;
}

int main()
{
    PAR p = f(); //используем псевдоним для объявления переменной для
                приема возвращаемого значения
    p[1][1] = 1;
    ...
}
```

Возвращаемое значение является указателем на функцию

Встречаются задачи, в которых требуется в качестве возвращаемого значения сформировать указатель на функцию (листинг 8.34).

Листинг 8.34. Функция, возвращающая указатель на другую функцию

```
int f1() //это функция, указатель на которую требуется вернуть
{
    return 1;
}

int (*f2(int))() //функция f2 принимает параметр типа int и возвращает
                указатель на функцию, которая не принимает
                параметров, а возвращает int
{
    return f1;
}

int main()
```

```
{  
    int (*pf)() = f2(5);  
    int res = (*pf)(); //теперь можем вызвать функцию f1 посредством  
                        указателя pf  
}
```

Перепишем пример, используя псевдоним типа указатель на функцию (листинг 8.35).

Листинг 8.35. Использование typedef для указателя на функцию

```
int f1()  
{  
    return 1;  
}  
  
typedef int (*PF)(void); //PF – псевдоним типа указатель на функцию,  
                          которая не принимает параметров и  
                          возвращает int  
  
PF f2(int) //используем псевдоним для типа возвращаемого значения  
{  
    return f1;  
}  
  
int main()  
{  
    PF pf = f2(5); //используем псевдоним для объявления переменной,  
                  которая принимает возвращаемое значение  
    int res = (*pf)();  
}
```




Глава 9

Структуры

В этой главе речь пойдет в основном о структурах языка Си, так как структуры языка C++ не имеют принципиальных отличий от *классов*, а рассмотрение классов выходит за рамки данной книги. *Классы* — это одно из основных понятий объектно-ориентированного программирования.

9.1. Зачем нужны структуры

В некоторых задачах удобно оперировать совокупностью переменных как одним программным объектом. Например, мы разрабатываем программу, которая содержит информацию о группе людей. О каждом человеке требуется хранить следующие данные: имя, пол, год рождения, рост, вес и т. д. Если бы такого понятия, как структуры, не было, то решение задачи выглядело бы примерно так, как показано в листинге 9.1.

Листинг 9.1. Без использования структур

```
{
    enum SEX{MALE, FEMALE}; //пол может быть мужским или женским,
                             //поэтому удобно для обозначения пола
                             //вести перечисление

//Для формирования информации о каждом человеке требуется определить
//набор переменных
//Для Васи:
    char name1[30] = "Вася";
    enum SEX sex1 = MALE; //использование ключевого слова enum при
                           //определении переменной перечислимого типа
                           //обязательно в языке Си,
```

в C++ его можно опустить

```
int age1 = 30;
...
//Для Мани:
char name100[30] = "Маня";
enum SEX sex100 = FEMALE;
int age100 = 2;
}
```

А если таких наборов данных много? Программист должен все время помнить, какие данные относятся к одному и тому же конкретному человеку, поэтому логично отдельные характеристики, относящиеся к одному объекту, объединить в составном элементе данных так, чтобы:

- ☐ хранить совокупность характеристик как единое целое;
- ☐ манипулировать этой совокупностью как единым целым;
- ☐ иметь возможность обращаться к характеристикам по отдельности.

Структуры языка Си как раз и предоставляют программисту возможность формирования таких новых составных (*агрегатных* — aggregate) типов данных, которые строятся на базе уже существующих (определенных ранее) типов. Структура может включать в свой состав произвольное количество типов данных, которые в дальнейшем будем называть *полями* структуры. В качестве поля можно использовать любой из ранее определенных типов данных, как базовых, так и более сложных: указатели, строки, массивы, другие структуры.

Разные типы структур могут различаться:

- ☐ количеством полей;
- ☐ их типами;
- ☐ порядком расположения полей в структуре.

ЗАМЕЧАНИЕ

Структуры языка Си — это удобное средство для *укрупнения* (агрегации) данных (ничего более!), в то время, как структуры языка C++ обладают гораздо большими возможностями.

9.2. Объявление структуры

Компилятор сам знает без дополнительных указаний программиста, как обращаться с переменными базового типа. Программисту нужно лишь объявить переменную требуемого (базового) типа. Однако только программист знает,

сколько и каких полей должен содержать агрегатный пользовательский тип. Поэтому программист должен описать компилятору свойства своего нового пользовательского типа данных, т. е. объявить структуру.

Объявление структуры — это описание внутреннего устройства нового агрегатного типа данных (количества, типа и порядка расположения полей), исходя из которого, компилятор будет создавать экземпляры пользовательского типа и манипулировать ими.

НАПОМИНАНИЕ 1

При объявлении структуры, равно как и любого другого программного объекта, никакой памяти не резервируется! Это только описание, исходя из которого, компилятор будет резервировать память при создании каждого экземпляра переменной структурного типа.

НАПОМИНАНИЕ 2

Обычно объявления помещают в заголовочный файл (рис. 9.1).

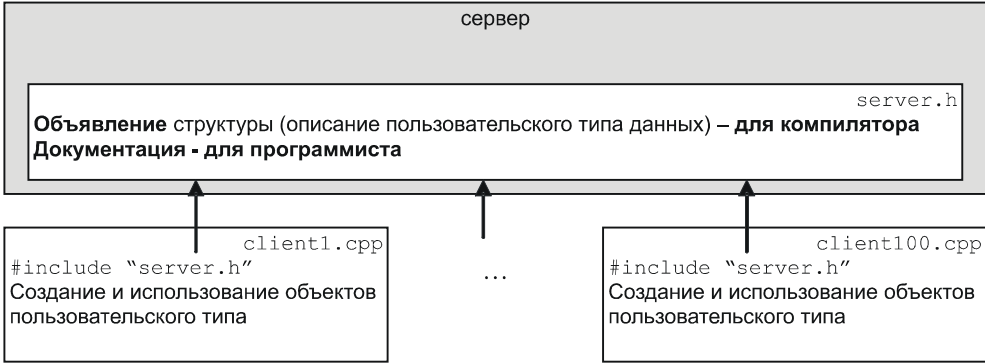


Рис. 9.1

Синтаксис объявления структуры:

```
struct имя_пользовательского_типа { // ключевое слово struct означает, что
                                     вводится новый пользовательский
                                     агрегатный тип данных с указанным
                                     именем
    список_полей_структуры (типы и имена переменных, которые будут
                             присутствовать в каждом создаваемом
                             экземпляре такой структуры)
}; // после закрывающей фигурной скобки точка с запятой обязательна!
```

Пример.

Спроектируем структуру `human` для описания свойств любого человека (эту структуру будем использовать в последующих примерах). Поместим ее объявление в заголовочный файл `human.h`, здесь же зададим перечисление для обозначения пола:

```
//файл human.h
enum SEX {MALE, FEMALE};
struct human{//ввели свой пользовательский агрегатный тип human,
               в котором сгруппировали все нужные поля данных
enum SEX sex; //в C++ ключевое слово enum можно опустить
int age;
char name[30];
...
};
```

9.3. Создание экземпляров структуры и присваивание значений полям структуры

Создание объекта пользовательского типа выглядит так же, как создание переменной базового типа (листинг 9.2). Компилятор резервирует для переменной типа `human` `sizeof(human)` байтов:

Листинг 9.2. Создание экземпляров структуры

```
#include "human.h"
int main()
{
    //Создание переменной пользовательского типа:
    //В языке ANSI Си
    struct human man1; //тип переменной - struct human,
                       имя переменной - man1. Ключевое
                       слово struct обязательно в языке Си,
                       но не обязательно в C++, где достаточно
                       написать так:

    //В C++
    human woman1; //тип переменной - human, имя переменной - woman1,
                  а о том, что human - это структура, компилятор
                  помнит сам
}
```

ЗАМЕЧАНИЕ

Компилятор размещает и неявно инициализирует такие переменные по общим правилам (так же, как переменные базовых типов, см. разд. 3.7 и 3.9.2). В зависимости от контекста определения такие переменные могут быть: глобальными, заключенными в пространства имен, статическими или динамическими. Для глобальных структурных переменных, переменных в пространстве имен и статических переменных компилятор неявно инициализирует все поля нулями. Поля локальных и динамических переменных не инициализируются.

Если в распоряжении программиста есть переменная структурного типа или ссылка (только в C++) на такую переменную, то обратиться к полю структуры программист может посредством селектора поля структуры (. — точка).

Синтаксически обращение выглядит следующим образом:

имя_переменной.имя_поля

или

имя_ссылки_на_переменную.имя_поля

Пример обращения к полям структурной переменной приведен в листинге 9.3.

Листинг 9.3. Присваивание значений полям структуры посредством имени переменной или ссылки и селектора поля структуры (.)

```
#include "human.h"

int main()
{
    struct human man1;
    man1.sex = MALE; //встречая такое выражение, компилятор вычисляет
                     //адрес (куда нужно занести значение MALE) следующим образом:
                     //&man1+смещение_поля_sex_относительно_начала_экземпляра
                     //в нашем случае (&man1 + 0байт)
    man1.age = 30; //( &man1 + 4байта)

    //В C++ можно определить ссылку на структуру, при этом ключевое слово
    //struct можно опустить, т. к. в C++ оно необязательно
    human& rman1 = man1; //ссылка rman1 является псевдонимом
                         //структурной переменной man1
    rman1.age++; //man1 "повзрослел" на год

    //Но!
    //man1.name = "Вася"; //ошибка, т. к. name - это имя массива
                         //(см. разд. 6.3.1)
    strcpy(man1.name, "Вася"); //т. к. поле name - это массив,
                              //содержащий строку, требуемое значение
```

проще скопировать с помощью функции `strcpy()` стандартной библиотеки

```
...
//Можно для описания группы людей создать массив структур:
struct human people[10]; //определен массив из 10 элементов
                             типа human. Так как массив локальный,
                             то его поля компилятором не
                             инициализируются
//Предоставим пользователю возможность ввести значения:
for(int i=0; i < sizeof(people)/sizeof(human); i++)
{
    printf("Name=");
    scanf("%29s",people[i].name); //ограничили количество
                                     принимаемых символов размером массива
    fflush(stdin); //если пользователь ввел строку длиннее, чем
                                     размер массива, нужно очистить буфер ввода
    printf("Input sex: male (0), female (1)");
    scanf("%d", &people[i].sex);
    printf("age=");
    scanf("%d", &people[i].age);
}
}
```

9.4. Ключевое слово *typedef* и структуры

Если вы программируете на языке Си, то при любом использовании имени типа `human`, должны уточнять компилятору тип с помощью ключевого слова **struct** (т. е. везде должны писать **struct human**). Для того чтобы можно было при использовании структуры в языке Си опускать ключевое слово **struct**, вводят псевдонимы посредством ключевого слова **typedef** (см. разд. 3.5.1).

Например:

```
typedef struct { //имя пользовательского типа опущено
    SEX sex;
    int age;
    char name[30];
    ...
} human; //анонимной структуре с перечисленными полями
           сопоставляется псевдоним human
```

Теперь при программировании на Си можно везде, где требуется **struct** имени_пользовательского_типа, использовать псевдоним, а компилятор под псевдонимом будет подразумевать структуру с перечисленными полями, например:

```
human man;
```

ЗАМЕЧАНИЕ 1

Если вам придется программировать на уровне Win32 API, то увидите, что введение псевдонимов — это прием, который часто используется разработчиками для структур Windows.

ЗАМЕЧАНИЕ 2

В С++ ключевое слово **struct** при использовании структур необязательно, поэтому и применение **typedef** для обозначения псевдонима пользовательского типа тоже неактуально.

9.5. Совмещение объявления и определения. Анонимные структуры

Иногда для локального использования совмещают объявление структуры и создание переменных структурного типа. Например:

```
// Файл 1.cpp
```

```
struct human{ //ввели свой пользовательский агрегатный тип human
    SEX sex;
    int age;
    char name[30];
    ...
} man1, woman1, *phuman, people[10]; //создали две переменных типа
                                     human - man1, woman1,
                                     указатель типа human* - phuman
                                     и массив из 10 элементов
                                     типа human - people
```

Синтаксис языка Си позволяет объявить структуру, не указывая имени типа. При этом очевидно, что в дальнейшем использовать такую структуру невозможно. Поэтому целесообразность имеет место лишь в том случае, если такая структура нужна единожды для создания нескольких экземпляров, а в дальнейшем имя типа больше не потребуется (листинг 9.4).

Листинг 9.4. Использование анонимных структур

```
int main()
{
    //Создание переменных структурного типа:
    struct { //имя пользовательского типа опущено
        SEX sex;
        int age;
        char name[30];
        ...
    } man1, woman1, *phuman, people[10];
    //Использование переменных man1, woman1, phuman, people
}
```

ЗАМЕЧАНИЕ

Злоупотреблять таким способом создания переменных вряд ли стоит, т. к. это нарушает принципы структурного подхода (хотя синтаксически допустимо).

9.6. Инициализация структурных переменных

При определении структуры, ее поля можно проинициализировать явно, как и при определении переменной базового типа. Инициализация структур похожа на инициализацию массивов. Инициализаторы в фигурных скобках указываются в том же порядке, в котором в структуре объявлены соответствующие поля:

```
struct human man1 = {MALE, 30, "Вася"}; //определена структурная
                                         переменная с именем man1 и ее
                                         поля проинициализированы при
                                         создании указанными значениями
```

Инициализация массивов структур похожа на инициализацию многомерных массивов:

```
struct human people[100] = {
    {MALE, 30, "Вася"},
    {FEMALE, 20, "Маша"},
    ...
};
```


Для структур справедливы правила неполной инициализации (так же, как и для массивов):

```
struct human man2 = {MALE}; //все остальные поля компилятор
                             проинициализирует нулевыми значениями
struct human man3 = {0};    //компилятор обнулит все поля структуры
```

Так же, как и для массивов, проинициализировать поля структуры можно только при создании, поэтому попытка использовать список инициализаторов для уже существующей структурной переменной вызовет ошибку. Например:

```
man2 = {MALE, 30, "Вася"}; //ошибка: список инициализации можно
                             использовать только при определении
```

9.7. Действия со структурами

Компилятор умеет создавать копии существующих структурных переменных (это определение, совмещенное с инициализацией) и копировать поля одного (уже существующего экземпляра структуры) в поля другого (тоже уже существующего) экземпляра структуры того же типа (это присваивание). Действия компилятора при выполнении этих двух операций в языке Си отличаются только тем, что при инициализации компилятор выделяет память под новый объект. В C++ это две принципиально разные операции.

Например:

```
{
    struct human man1 = {"Вася", MALE, 30};
    struct human man2 = man1; //создание нового объекта и инициализация.
                             Компилятор выделяет память для man2 и
                             копирует значения всех полей экземпляра
                             man1 в соответствующие поля экземпляра
                             man2. Таким образом, man2 становится
                             копией man1
    man1 = man2; //присваивание: значения полей одного существующего
                  экземпляра заменяются значениями полей другого
                  существующего
}
```

ЗАМЕЧАНИЕ

В обоих случаях компилятор переписывает содержимое одной структурной переменной в другую очень эффективно — как копирование последовательности байтов (без разделения на поля). Для структур C++ и классов механизм копирования информации гораздо сложнее.

9.8. Поля структуры пользовательского типа

Поле структуры может быть:

- ❑ базового типа — `int age`;
- ❑ указателем на базовый тип — `int* p`;
- ❑ массивом элементов базового типа — `char name[30]`;
- ❑ пользовательского типа.

Пример приведен в листинге 9.5.

Листинг 9.5. Поля структуры пользовательского типа

```
//Файл point.h
#if !defined _POINT //защита от вложенных подключений point.h
    (см. разд. 5.6.6)

#define _POINT
struct Point{ //в этой структуре поля базового типа
    int x, y; //координаты точки
};
#endif

//Файл line.h
#include "point.h"
struct Line{ //отрезок линии, который задается двумя точками
    struct Point point1, point2; //а в этой структуре тип полей задан
                                программистом
};

//Файл main.cpp
#include "point.h"
#include "line.h"
int main()
{
    struct Point a = {1,1}, b = {5,5}, c = {-1,-2};
    //Инициализировать переменные типа Line можно следующими
    способами:
    struct Line l1 = {a,b};
    struct Line l2 = {-1,-2,2,3};
```

```
struct Line l3={0};  
//Обращаться к полям можно:  
l3.point1 = c;  
l3.point1.x = 1;  
}
```

9.9. Вложенные (nested) структуры

Объявление структуры может включать в свой состав объявления других вспомогательных пользовательских типов. Пример приведен в листинге 9.6.

Листинг 9.6. Вложенные объявления в структурах

```
//файл line.h  
struct Line  
{  
    struct Point //вложенное объявление  
    {  
        int x, y;  
    } point1, point2; //объявление полей структуры Line типа Point  
};
```

В С++ область видимости всех объявлений внутри структуры ограничена, в то время как в языке Си таких ограничений нет. В С++ все поля структуры как бы заключены в пространство имен с именем структуры. Для обращения к таким вложенным объявлениям требуется указание типа структуры и оператора разрешения области видимости — в нашем случае — `Line`: (листинг 9.7).

Листинг 9.7. Спецификатор разрешения области видимости применительно к структурам С++

```
//файл main.cpp  
#include "line.h"  
int main()  
{  
    //В языке Си:  
    struct Point a; //OK  
    //В языке С++:  
    //Point a; //ошибка: компилятор не знает, что такое Point
```

```

Line::Point b = {1,2}; //теперь создавать переменные типа
                        Point можно только посредством указания
                        типа структуры и оператора разрешения
                        области видимости - Line::
}

```

Вспомогательные вложенные структуры можно сделать анонимными (листинг 9.8).

Листинг 9.8. Анонимные вложенные структуры

```

//Файл line.h
struct Line
{
    struct //объявление анонимной структуры
    {
        int x, y;
    } point1, point2; //объявление полей
};

//Теперь можно обращаться к полям point1 и point2 следующим образом:
int main()
{
    struct Line l = {1,2,3,4};
    l.point1.x = -1;
}

```

9.10. Указатели и структуры

Иногда бывает удобно (а иногда это просто необходимо) манипулировать структурной переменной посредством указателя. В основном необходимость в указателе возникает в тех случаях, когда адрес структурной переменной передается в функцию в виде указателя или память под такую переменную/массив выделяется динамически. Для обращения к полям структуры посредством указателя используется селектор `->` (стрелка вправо). Пример приведен в листинге 9.9.

Листинг 9.9. Обращение к полям структуры посредством указателя

```

#include <malloc.h> //прототипы функций malloc и free
#include "human.h"
int main()

```

```
{
    //Создание одиночной динамической структурной переменной
    //В языке Си
    struct human* pman = (struct human*) malloc(sizeof(struct human));
    //В языке C++
    human* pman1 = new human;
    pman->sex = MALE; //обращение к полю структуры посредством
                      указателя
    pman->age = 30;
    strcpy(pman->name, "Вася");
    ...
    //Попользовавшись динамическим объектом, не забудьте
    освободить память:
    //В языке Си
    free(pman);
    //В языке C++
    delete pman1;

    //Создание динамического массива:
    int n = ... //вычисление размерности массива
    //В языке Си
    struct human* people = (struct human*) malloc(n*sizeof(struct human));
    //В языке C++
    human* people1 = new human[n];
    //Формирование полей элементов массива
    //Вывод информации о каждом человеке:
    for(int i=0; i<n; i++)
    {
        cout<<people1[i].name<<
        " age: "<< people1[i].age<<
        (" sex: "<< (people1[i].sex==MALE)? "male" : "female")<< endl;
    }
    ...
    //В языке Си
    free(people);
    //В языке C++
    delete[] people1;
}
```

9.11. Упаковка полей структуры компилятором. Оператор *sizeof* применительно к структурам

При выделении памяти под структурную переменную (независимо от того, в какой конкретно области выделяется память: в стеке, в статической области или в куче) компилятор гарантированно делает следующее:

- выделяет количество байтов, большее или равное сумме всех полей структуры;
- выделяет память для каждого поля в том порядке, в котором поля объявлены в структуре.

Количество выделяемой для структурной переменной памяти и смещения полей относительно начального адреса структуры зависят от:

- оптимизирующих возможностей конкретного компилятора;
- опций командной строки компилятору.

В общем случае трудно сделать предположение о том, сколько памяти будет выделено под структурную переменную и с каким смещением данные будут располагаться относительно начала структурной переменной (об этом знает компилятор).

СЛЕДСТВИЕ

Если программист обращается к полю структуры по имени, то компилятор всегда вычисляет адрес корректно. Если же программист самостоятельно пытается задать смещение интересующего его поля относительно начала структуры, то вероятность ошибки велика. Не имеет смысла выполнять работу компилятора!

Оптимизирующий компилятор (например, VC) при выделении памяти под каждое поле структуры может минимизировать время обращения к любому полю структуры следующим образом: он располагает данные так, чтобы любое из них можно было прочесть из памяти за минимальное количество канальных циклов. Это означает, что любой элемент данных (чтобы его не нужно было считывать по кускам) должен быть размещен по адресу, кратному длине элемента. Эти правила следует учитывать для более эффективного использования памяти.

Например, требуется объявить в структуре с именем *A* поля типа **char**, **double**, **bool**, **int**. Если программист не знает правил упаковки и объявляет поля структуры в произвольном порядке (листинг 9.10), компилятор VC по умолчанию запаковывает поля структуры так, как показано на рис. 9.2.

Листинг 9.10. Неэффективный порядок объявления полей в структуре

```
//файл A.h
struct A{
    char c;
    double d;
    bool b;
    int n;
};
//файл main.cpp
#include "A.h"
int main()
{
    size_t n = sizeof(A); //n=24
}
```

Упаковка полей структуры компилятором

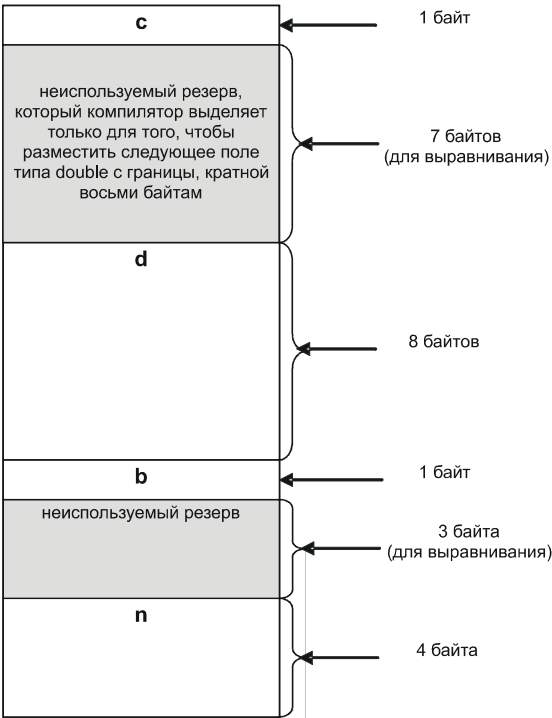


Рис. 9.2

А для того, чтобы эффективнее использовать память, достаточно было просто объявить поля структуры в другом порядке с учетом опций оптимизации компилятора VC (листинг 9.11, рис. 9.3).

Листинг 9.11. Как программист может учесть правила запаковки полей

```
//файл A.h
struct A{
    double d;
    int n;
    char c;
    bool b;
};

//файл main.cpp
#include "A.h"
int main()
{
    size_t n = sizeof(A); //в этом случае VC: n=16
}
```

Каким образом программист может «уплотнить» поля структуры

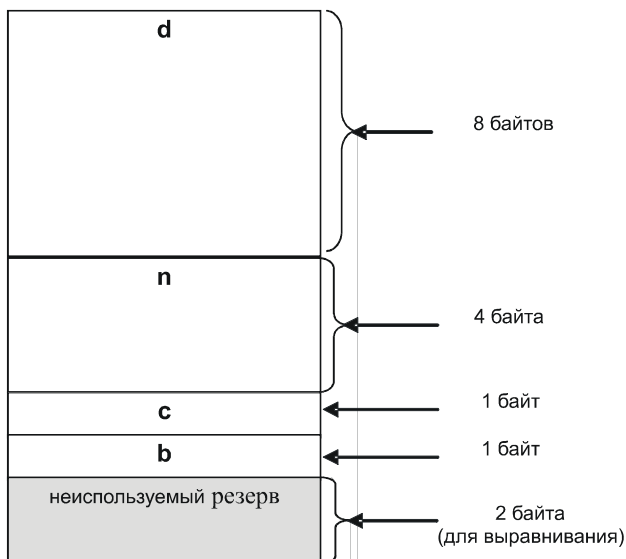


Рис. 9.3

9.12. Структуры и функции

Основным преимуществом структур является возможность манипулирования совокупностью данных как единым целым, в частности, это проявляется при передаче данных в функцию и при получении из функции результата.

9.12.1. Передача структуры в функцию в качестве параметра

Большое количество параметров, передаваемых в функцию, затрудняет ее использование. Например, требуется реализовать функцию, которая выводит на экран сведения о человеке. Без использования структур, функция выглядела бы примерно так:

```
void Print(const char* name, unsigned int age, SEX sex); //список
                                параметров можно расширить другими интересующими
                                нас данными о человеке
```

Структуры значительно упрощают передачу данных функциям, т. к. вместо трех отдельных параметров можно в данном случае использовать только один укрупненный — структуру `human`, содержащую три поля:

```
void Print(struct human any);
```

Данные, укрупненные посредством структуры, можно передавать в функцию как по значению, так и по адресу (аналогично параметрам базового типа — см. разд. 8.2).

Передача структуры в функцию по значению

Если структурная переменная передается по значению, то при вызове функции компилятор должен сформировать в стеке ее копию. Это вызывает дополнительные затраты:

- памяти (в стеке) для копии;
- времени на формирование копии.

А так как обычно посредством структур программист объединяет значительное количество полей, то передавать такие агрегатные данные по значению неэффективно (листинг 9.12).

Листинг 9.12. Передача структуры в качестве параметра по значению

```
#include <stdio.h> //прототип функции printf() для языка Си
void Print(struct human h) //функция принимает параметр структурного типа
                                по значению
```

```

{
    printf("%s age: %d sex: %s", h.name, h.age,
           ((h.sex==MALE)? "male" : "female"));
}
int main()
{
    struct human people[10];
    ...//Формирование полей элементов массива
    //Вывод информации о каждом человеке:
    for(int i=0; i<10; i++)
    {
        Print(people[i]); //на каждой итерации цикла компилятор
                           формирует в стеке копию очередного
                           элемента массива. Неэффективно!
    }
}

```

РЕКОМЕНДАЦИЯ

Передавать такие большие объекты по значению стоит только в тех случаях, когда программист намеренно предоставляет функции копию для модификации (чтобы изменения никак не затронули оригинал).

Передача структуры в функцию по адресу

Гораздо эффективнее передавать функции адрес структурной переменной посредством указателя или ссылки (листинги 9.13 и 9.14). Если при этом требуется защитить оригинал от модификации, то при передаче такого адреса следует использовать ключевое слово **const** (см. разд. 8.4.1).

Листинг 9.13. Передача адреса структурной переменной посредством указателя

```

void Print(const struct human* ph) //функция принимает адрес структурной
                                   переменной. При наличии ключевого
                                   слова const такой адрес допустимо
                                   использовать только для чтения

{
    printf("%s age: %d sex: %s", ph->name, ph->age,
           ((ph->sex==MALE)? "male" : "female"));
}
int main()
{

```

```
struct human people[10];
... //Формирование полей элементов массива
//Вывод информации о каждом человеке:
for(int i=0; i<n; i++)
{
    Print(&people[i]); //на каждой итерации цикла компилятор
                        формирует в стеке в качестве
                        передаваемого параметра адрес
                        экземпляра структуры human,
                        являющегося очередным элементом
                        массива people
}
```

Листинг 9.14. Передача адреса структурной переменной посредством ссылки (только C++)

[illegible]

Передача в функцию массива структур

Передача массива структур в функцию аналогична передаче массива базового типа, при этом компилятор передает в функцию адрес начала массива (см. разд. 8.2.3). Например, реализуем функцию, которая ищет в массиве с элементами типа `human` человека с указанным именем (листинг 9.15).

Листинг 9.15. Передача в функцию массива структур

```
struct human* find(const struct human* ar, const char* p, int n)
    //или const struct human ar[]
{
    for(int i=0; i<n; i++)
    {
        if(strcmp(ar[i].name,p) return ar+i; //или return &ar[i];
    }
    return 0; //человека с таким именем в массиве не оказалось
}

int main()
{
    struct human people[100] = {
        {"Вася", MALE, 30, ...},
        {"Маша", FEMALE, 20, ...},
        ...
    };
    struct human* p = find(people, "Вася", sizeof(people)/
        sizeof(struct human));
    if(p)... //такой человек в массиве есть
}
```

9.12.2. Возврат структуры по значению

Для того чтобы принять возвращаемое значение структурного типа, компилятор формирует вызов такой функции специфическим образом (листинг 9.16).

Листинг 9.16. Возврат структуры по значению

```
struct human f(void) //функция, которая возвращает объект типа human
    по значению
{
```

```
struct human tmp; //локальная переменная (компилятор отводит для
                  //ее память в стековом кадре текущей функции)
//формирование полей
return tmp; //для формирования возвращаемого значения компилятор
            копирует поля локальной переменной по тому адресу,
            который предоставила вызывающая функция
}
int main()
{
    human res;
    res = f(); //на первый взгляд эта функция не принимает никаких
              //параметров, но т. к. она возвращает объект
              //агрегатного типа по значению, то компилятор
              //формирует вызов следующим образом:
    а) компилятор резервирует в стеке место для
        возвращаемого значения и передает адрес
        зарезервированной области в функцию (VC передает
        этот адрес на регистре). Можно интерпретировать
        вызов такой функции следующим образом: компилятор
        вызывает функцию с невидимым параметром:
        f(адрес_для_возвращаемого_значения);
    б) после возврата управления компилятор копирует
        поля возвращаемого значения в поля переменной res
}
```

9.13. Что можно использовать в качестве поля структуры

Посредством структуры программист группирует данные разного типа. Поля структуры могут быть предназначены как для хранения той информации, которая в конечном итоге нужна пользователю, так и для формирования служебной (вспомогательной) информации, которая позволяет программисту группировать данные в сложные совокупности — списки, стеки, деревья и т. д. Поля структуры могут быть следующего типа:

❑ переменные базового типа. Например:

```
struct A{int a; double d;...};
```

❑ встроенные массивы базового типа. Например:

```
struct A{int a[10]; char c[20];...};
```

- ❑ указатели базовых типов. Например:
`struct A{int* a; char* c;...};`
- ❑ переменные пользовательского типа (структуры и объединения). Например:
`struct A{...};`
`struct B{A a;...};`
- ❑ встроенные массивы пользовательского типа. Например:
`struct A{...};`
`struct B{A a[10];...};`
- ❑ указатели другого или того же пользовательского типа (используются, в частности, при создании сложных структур данных — списков, деревьев и т. д.). Например:
`struct A{...};`
`struct B{A* pa; B* pB;...};`

9.14. Поля битов

Для некоторых элементов данных диапазон изменения значений мал (например, месяцев в году всего 12, дней в месяце не может быть больше 31 и т. д.). Иногда хочется для хранения таких данных отвести памяти не больше, чем это необходимо (для хранения месяца хватило бы 4 разряда, а для хранения дня в месяце хватило бы 5 разрядов). Но таких коротких встроенных типов данных в C/C++ нет (наименьшей адресуемой единицей является байт).

Пусть программа интенсивно оперирует датами, и требуется для хранения данных отвести как можно меньше памяти. Рассмотрим, каким образом можно минимизировать использование памяти.

Первый способ.

Очевидный вариант хранения даты.

Заводим для каждого из элементов данных свою переменную одного из базовых типов, стараясь занять минимум памяти (листинг 9.17):

- ❑ **unsigned short** — для года;
- ❑ **unsigned char** — для месяца;
- ❑ **unsigned char** — для дня
- ❑ и т. д.

Листинг 9.17. Использование подходящих коротких целых типов

```

//Файл date.h содержит объявление структуры Date и вспомогательные
//объявления:
enum WEEKDAY{MONDAY, TUESDAY, ...}; //для обозначения дня недели удобно
//ввести перечисление

struct Date{
//год
    unsigned short year; //а на самом деле диапазон изменения значений
                        //1-3000, поэтому достаточно 12 битов

//месяц
    unsigned char month; //а на самом деле диапазон изменения значений
                        //1-12, поэтому достаточно 4 битов

//день в месяце
    unsigned char day; //а на самом деле диапазон изменения значений
                        //1-31, поэтому достаточно 5 битов

//день недели
    WEEKDAY wday; //т. к. перечисление - это эквивалент int,
                //то компилятор выделит под такое поле sizeof(int)
                //байтов, а на самом деле диапазон изменения
                //значений 0-6, поэтому достаточно 3 бита

//признак праздник/будний день
    bool isHoliday; //встроенный тип C++ bool занимает 1 байт, а в
                    //языке Си BOOL является псевдонимом int
                    //(см. разд. 3.4.7) и занимает sizeof(int) байтов,
                    //а на самом деле для хранения признака
                    //достаточно 1 бита

//и, возможно, еще какую-нибудь специфическую информацию, относящуюся
//к конкретной дате
    ...
};

```

Размер экземпляра такой структуры равен суммарному размеру полей или больше (с учетом выравнивания полей структуры компилятором) (см. разд. 9.11). Поэтому даже несмотря на то что, зная эту особенность, мы постарались расположить поля оптимально, компилятор VC выделит для экземпляра такой структуры 12 байтов, из которых только 25 битов используются для хранения полезной информации:

```
size_t n = sizeof(Date); //12 байтов
```

Очевидными недостатками такого варианта хранения данных являются:

- неоптимальное использование памяти (при создании каждого экземпляра компилятор резервирует неиспользуемые биты, и даже байты);
- большие затраты при копировании экземпляров такой структуры (наряду с полезной информацией копируется также и неиспользуемый резерв).

Но у такого подхода есть также и неоспоримое достоинство — обработка отдельных компонент даты выглядит просто с точки зрения программиста и легко реализуется компилятором. Например:

```
Date d = {2007, 9, 1, SATURDAY, true};
d.year++; //компилятор модифицирует содержимое по вычисленному
          адресу, используя малое количество простых действий
```

Поэтому если в программе используются несколько экземпляров структуры `Date`, то, скорее всего, именно этим способом и стоит воспользоваться, т. к. потери памяти будут не существенны, и ими можно пренебречь ради простоты обработки.

А если таких наборов данных в программе миллионы? Тогда стоит подумать над альтернативным решением.

Второй способ.

Для хранения каждого элемента данных выделяем ровно столько разрядов, сколько требуется. Зная, что все наши данные помещаются в 25 разрядов, объявляем в структуре одно поле типа `int` или `unsigned int` (рис. 9.4):

```
//файл date.h
struct Date{
    unsigned int date; //в эту переменную запакуем те данные,
                      для каждого из которых в предыдущем варианте
                      отводилось отдельное поле, и сами распределим
                      в нем биты таким образом, как показано
                      на рис. 9.4.
};
```

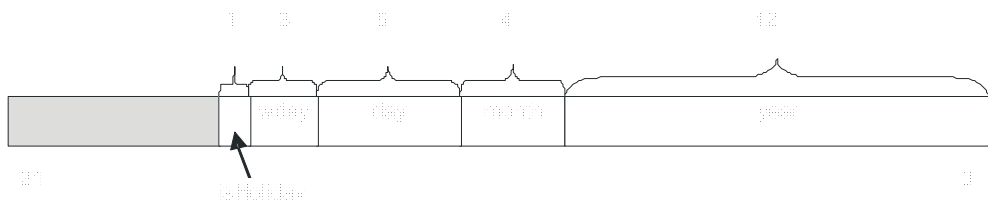


Рис. 9.4

Теперь каждый экземпляр структуры вместо 12 будет занимать 4 байта:

```
size_t n = sizeof(Date); //4 байта
```

Достоинствами этого варианта являются:

- ❑ плотная упаковка данных, в результате чего практически всегда получается выигрыш по занимаемой памяти;
- ❑ уменьшение затрат времени на манипулирование такой составной единицей данных как единым целым (например, при копировании или при одновременной переустановке всех компонентов даты).

Но появились существенные недостатки:

- ❑ программисту нужно все время помнить, какие биты для чего им отведены (компилятор об этом ничего не знает);
- ❑ резко усложняются действия с отдельными компонентами. Например, требуется задать новое или модифицировать текущее значение месяца. Это действие, очень просто реализуемое первым способом, в данном варианте требует выполнения трех шагов (листинг 9.18):
 - распаковка — преобразование из упакованного формата (соответствующих битов поля date) в естественный (число от 1 до 12) — иллюстрируется рис. 9.5;
 - выполнение действия со значением месяца (с проверкой того, находится ли результат действия в том же диапазоне);
 - упаковка — внедрение на свое место в поле date .

В результате получаем проигрыш по скорости, и увеличивается время выполнения.

Листинг 9.18. Ручная упаковка/распаковка значений (если бы такого понятия, как поля битов, не было)

```
#include "date.h"

int main()
{
    Date d;

    //запаковываем исходные значения (17.08.2007, пятница, не праздник)
    // в поле структуры посредством побитовых операторов
    d.date = 2007 | //год располагается в младших разрядах,
                  //поэтому значение никуда двигать не нужно
              (8<<12) | //а значение месяца должно располагаться
                  с 12 по 16 бит, поэтому прежде чем
```

```

        поразрядно складывать, ставим значение
        на свое место
        (17<<16) |    //день месяца аналогично
        (FRIDAY<<21); //день недели аналогично, а бит признака
                        останется в результате нулевым (false)

...
//Требуется увеличить номер месяца на единицу:
    unsigned int m = (d.date >> 12) & 0xf; //для этого значение нужно
                                            сначала распаковать
                                            (рис. 9.5)

m++; //произвести действие
    if(m <= 12) //проверить: находится ли новое значение в требуемом
                диапазоне
    { //и в этом случае запаковать новое значение обратно
        d.date = (d.date & 0xffff0fff) //обнулили предыдущее
                                         значение месяца в поле date
        | m; //и внедрили на это место новое значение
    }
}

```

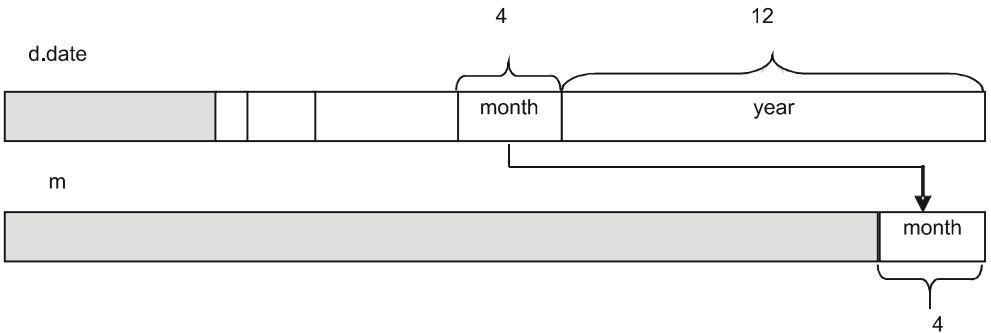


Рис. 9.5

Третий способ.

Использование битовых полей.

Альтернативой приведенным способам (первому и второму) является использование битовых полей (bitfields). Для компактной упаковки данных в памяти (с одной стороны) и для упрощения операций с такими короткими данными

(с другой), языки C/C++ предоставляют программисту особый тип данных, который можно использовать в качестве поля структуры или класса (только!).

Особенности использования битовых полей:

- ❑ под полем имеется в виду отдельно каждая короткая переменная;
- ❑ для каждой переменной отводится не больше битов, чем заказал программист (не больше, чем минимально необходимо);
- ❑ несколько таких логически разных переменных хранятся в одной физической переменной подходящей длины;
- ❑ компилятор за программиста, во-первых, выделяет нужное количество битов, а во-вторых — осуществляет побитовые операции и сдвиги, извлекая или запаковывая значения.

Использование таких полей (листинг 9.19) делает доступ к совокупностям битов удобным и легким для программиста. При этом очевидным преимуществом является экономия памяти.

Синтаксис объявления битового поля:

целый_тип [идентификатор] : размер_поля_в_битах;

Правила:

- ❑ целый тип может быть **signed** или **unsigned**. Таким образом вы сообщаете компилятору, как интерпретировать указанный диапазон. Знаковость влияет только на распаковку значения;
- ❑ в качестве целого типа могут быть использованы: **char**, **short**, **int** или **long**;
- ❑ в качестве целого типа может быть использован **enum** (интерпретируется компилятором как **signed int**);
- ❑ в C++ в качестве целого типа может быть также использован **bool**;
- ❑ если идентификатор опущен, то объявляется анонимное битовое поле, которое используется для выравнивания (**padding**).

Листинг 9.19. Использование битовых полей в структурах

```
//Файл date.h
enum WEEKDAY{MONDAY, TUESDAY,...}; //посредством перечисления будем хранить
                                     в экземпляре структуры день недели

struct Date
{
    unsigned int year:12; //12 битов
```

```
unsigned int month:4; //4 бита
unsigned int day:5;    //5 битов
WEEKDAY nWeekDay:4; //на первый взгляд кажется, что для представления
                     значения, диапазон изменения которого 0-7,
                     хватило бы и трех битов. Вспоминаем о том, что
                     enum – это эквивалент signed int. Это означает:
                     для того, чтобы при распаковке поля не получать
                     отрицательных значений, следует зарезервировать
                     один дополнительный бит, который всегда должен
                     содержать ноль
```

```
unsigned int isHoliday:1; //1 бит
```

```
};
//файл main.cpp
#include "date.h"
int main()
{
//Во всех приведенных ниже примерах мы пользуемся битовыми полями как
обычными полями структуры, а компилятор сам осуществляет
запаковку/распаковку значений с помощью побитовых операторов и
операторов сдвига
    Date d = {2007, 8, 17, FRIDAY}; //для инициализации битовых полей
                                     можно использовать список
                                     инициализаторов так же, как
                                     и для обычных полей структуры
    d.month = 9; //при работе с одним полем содержимое остальных полей
                 не модифицируется
    unsigned int m = d.month;
    d.nWeekDay=MONDAY;
    WEEKDAY wday = d.nWeekDay;
    ...
}
```

Рассмотрим специфику использования битовых полей.

- ❑ Разные компиляторы могут запаковывать битовые поля по-разному. Компилятор VC выделяет биты в порядке объявления, начиная с младшего, так же, как мы распределили их вручную вторым способом (см. рис. 9.4).
- ❑ Иногда для каких-либо специфических целей программист посредством анонимного битового поля может предписать компилятору просто заре-

зернировать указанное количество битов, т. е. следующее именованное битовое поле располагать после указанного резерва (рис. 9.6). Например:

```
struct A{
    short x:1;
    short y:2;
    short :1; //анонимное битовое поле - резерв
    short z:3;
    short :1; //анонимное битовое поле - резерв
    short w:4;
};
```

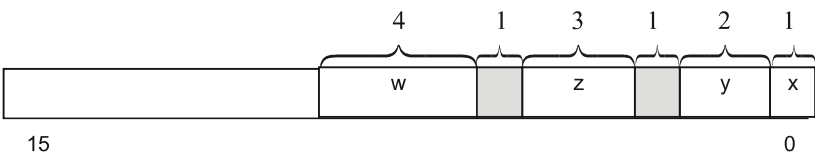


Рис. 9.6

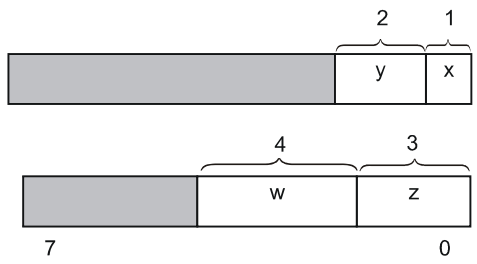


Рис. 9.7

□ Анонимное поле нулевой длины используется для указания компилятору — разместить следующую группу битовых полей, начиная с границы типа (рис. 9.7). В приведенном ниже примере нулевое анонимное поле предписывает компилятору размещать следующее битовое поле z, начиная с байтовой границы:

```
struct A{
    char x:1;
    char y:2;
```

```
char :0; //анонимное битовое поле нулевой длины
char z:3;
char w:4;
```

```
};
```

- Перед запаковкой битового поля компилятор обеспечивает требуемый диапазон путем обнуления всех разрядов, выходящих за пределы заданной программистом разрядности (листинг 9.20).

Листинг 9.20. Ограничение разрядности поля при запаковке

```
struct A{
    unsigned int x:3; //беззнаковое поле может принимать значения
                        от 0 до 7
};
int main()
{
    A a;
    int n=255; //это значение выходит за пределы диапазона изменения
                поля (шестнадцатеричное представление - 0xff)
    a.x = n; //при этом последовательность команд, которую генерирует
              компилятор, состоит из трех шагов (см. разд. П1.12):
    а)посредством побитовой конъюнкции обнулить все разряды
        операнда-источника за пределами разрядности битового
        поля - (n&7),
    б)обнулить битовое поле в операнде-приемнике, -
        (a.x&0xffffffff8),
    в)вставить операнд-источник в операнд-приемник
        посредством побитовой операции | -
        a.x = (n&7) | (a.x&0xffffffff8)
    //в результате присваивания поле a.x примет значение 7.
```

- Знаковость битового поля влияет только на процесс распаковки значения компилятором — компилятор генерирует разные последовательности низкоуровневых команд (листинг 9.21).

Листинг 9.21. Распаковка битовых полей с учетом знаковости

```
struct A{
    unsigned int x:3; //беззнаковое поле может принимать значения
                        от 0 до 7
```

```

};
struct B{
    int x:3; //знаковое поле может принимать значения от -4 до 3
};
int main()
{
    A a;
    B b;
    int n=255; //0xff
// запаковываются знаковые и беззнаковые поля одинаково:
    a.x = n; //в результате присваивания поле a.x примет значение 7
        (см. предыдущий пример), в двоичном виде 111
    b.x = n; //b.x=-1 (в двоичном представлении - 111).
//при распаковке знаковых и беззнаковых полей компилятор генерирует
разные низкоуровневые последовательности команд, т. к. при одном и том
же внутреннем представлении битового поля результат распаковки должен
получиться разным:
    int tmp1 = a.x; //tmp1=7 (двоичное представление - 000...0111)
        tmp1 = date&7;
    int tmp2 = b.x; //tmp1=-1 (двоичное представление - 111...1111)
        tmp2 = (date<<29)>>29;
}

```

- Если целый тип битовых полей отличается только знаковостью (**signed int/unsigned int**), то компилятор запаковывает их подряд (рис. 9.8), а знаковость влияет только на распаковку значений:

```

struct A{
    signed int x:1;
    unsigned int y:2;
    ...
};

```



Рис. 9.8

- Если у битовых полей целый тип разный (**int**, **char**), то компилятор начинает запаковывать следующее битовое поле с границы типа (рис. 9.9).

```
struct A{
    unsigned int x:1;
    unsigned char y:2;
    ...
};
```

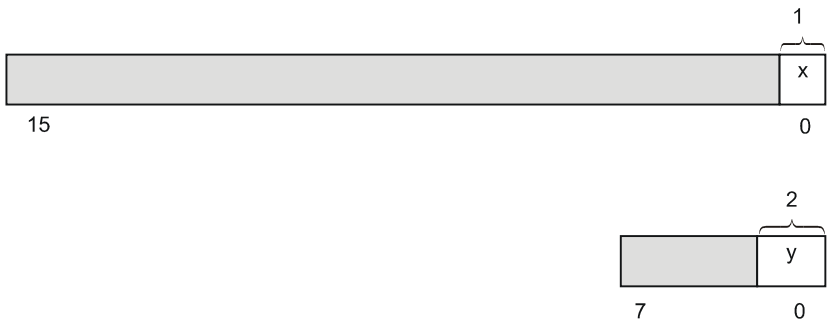


Рис. 9.9

- Для поля битов нельзя:
- получить адрес:
`int* p = &a.x; //ошибка;`
 - объявить ссылку и проинициализировать ее битовым полем:
`int& r = a.x; //ошибка;`
 - оператор **sizeof** с ними тоже не работает:
`size_t n = sizeof(a.x); //ошибка.`



Глава 10

Объединения (*union*)

Назначение объединений в языках Си и С++ одинаково, однако объединения С++ (так же, как структуры С++) обладают большими возможностями по сравнению с объединениями языка Си. В данной главе речь пойдет в основном о возможностях объединений языка Си.

10.1. Понятие объединения

Объединение (*union*) — это еще один агрегатный тип данных, который программист формирует из совокупности других типов — базовых или ранее определенных пользовательских. Основные действия с объединениями (объявление объединения, копирование экземпляров, передача в качестве параметра в функцию и т. д.) внешне выглядят так же, как аналогичные действия со структурами.

Например, объявление объединения выглядит следующим образом:

```
union B{ //ключевое слово union говорит компилятору о том, что далее
        следует объявление пользовательского типа данных,
        этот тип назван именем B
        список_полей_объединения
};
```

Назначение и внутренняя реализация объединений принципиально отличается от структур. При создании экземпляра структуры все перечисленные в объявлении поля одновременно присутствуют в памяти (компилятор под каждое поле отводит память). В экземпляре объединения компилятор располагает все поля, начиная с одного и того же адреса. Это означает, что, обращаясь к содержимому объединения посредством разных полей, можно одним и тем же содержимым пользоваться, как данными разного типа (см. примеры

использования в *разд. 10.2*). На рис. 10.1 приведены два варианта организации данных — посредством структуры и посредством объединения.

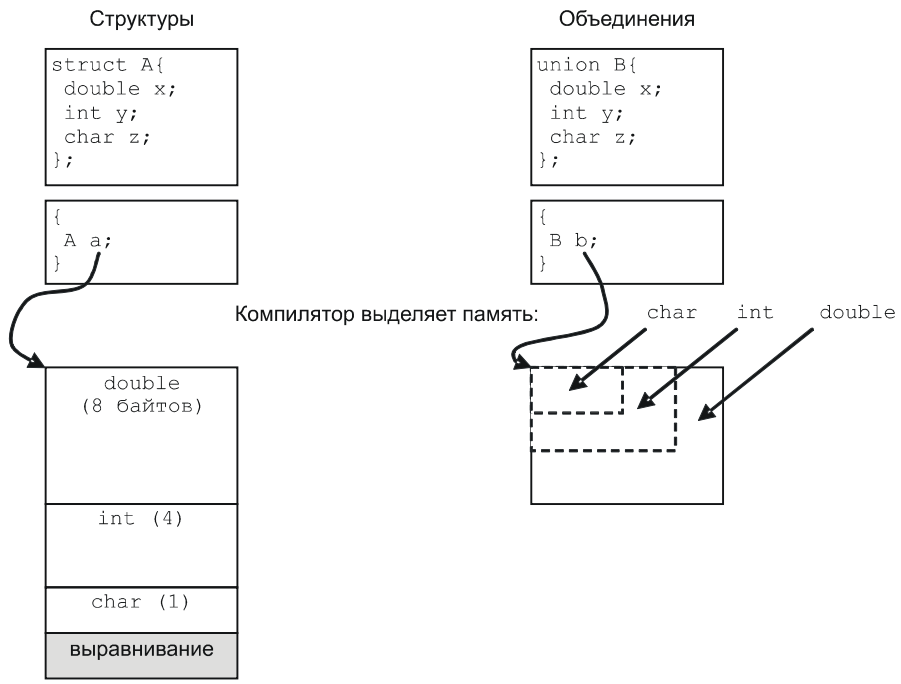


Рис. 10.1

10.2. Использование объединений

Рассмотрим, в каких случаях может понадобиться такой тип данных. Если программа работает в условиях дефицита памяти (в однокристальных микроЭВМ с очень маленькой памятью, используемых во встраиваемых применениях), то имеет смысл один и тот же участок памяти использовать на разных этапах исполнения программы для хранения разных переменных (листинг 10.1).

Листинг 10.1. Использование объединения для экономии памяти

```
//Файл Any.h  
  
union Any{ //объявление объединения  
    char c;
```

```
    int n;
    double d;
};
//Файл main.cpp
#include <stdio>
#include "Any.h"
int main()
{
//В языке Си
    union Any x; //создали экземпляр объединения
//В C++
    Any x;        //ключевое слово union не обязательно
    x.c = 'A';
                //используем выделенную компилятором область памяти
                как переменную типа char
    ...
                //байтовая переменная больше не нужна, можно
                использовать ту же область памяти для других целей,
                например, теперь нужна переменная типа int
    x.n = 1;
                //используем ту же область памяти для переменной
                типа int
    ...
                //а теперь для переменной типа double
    x.d = 2.2;
}
```

Объединение — это возможность интерпретировать одну и ту же область памяти (т. е. ее содержимое) по-разному. Этот прием удобно использовать, например, в ситуации, когда приложение получает данные от внешнего устройства как последовательность байтов, а смысл эти данные имеют только как четырехбайтовые целые, поэтому принимать их удобно в байтовый массив, а обрабатывать — как четырехбайтовые целые. Такую задачу можно было бы решить с помощью двух указателей разного типа: **char*** и **int*** (см. разд. 6.1.9), но более удобным может быть использование объединения вида, представленного в листинге 10.2.

Листинг 10.2. Использование объединения для интерпретации одной и той же области памяти по-разному

```
//Файл CharInt.h
union CharInt{ //объявление объединения
    char ar[4];
    int n;
};

//Файл main.cpp
#include <stdio>
#include "CharInt.h"
int main()
{
    union CharInt x; //создали экземпляр объединения
                      //получаем данные с внешнего устройства как
                      //последовательность байтов:
    for(int i=0; i<4; i++)
    {
        x.ar[i] = <значение_очередного_байта>
    }
                      //а используем полученные данные как int:
    printf("value= %d", x.n);
}
```

Объединения используют для унификации обмена данными разного типа между функциями или приложениями. В этом случае обычно "заворачивают" объединение, которое может содержать данное любого из перечисленных при объявлении типов, в структуру (листинг 10.3).

Листинг 10.3. Использование объединения для унификации передачи данных разного типа в функцию

```
//Файл uni.h содержит объявление унифицированной структуры
и вспомогательные объявления:
enum WHAT{CHAR, INT, DOUBLE}; //с помощью перечисления будем задавать
                                тип, передаваемого посредством
                                объединения данного

struct Uni{
    WHAT what;
    union { //имя можно опустить, т. к. в других местах программы
```

таким объединением пользоваться не будем

```
    char c;
    int i;
    double d;
} data;
};
//файл func.cpp
#include "uni.h"
void f(Uni* p)
{
    switch(p->what)
    {
        case CHAR:
            //пользуемся полем p->data.c
            break;
        case INT:
            //пользуемся полем p->data.n
            break;
        case DOUBLE:
            //пользуемся полем p->data.d
    }
}
```

10.3. Размер объединения

Размер объединения (сколько памяти компилятор выделит под переменную такого типа) определяется размером его наибольшего поля:

```
union B{
    double x;
    int y;
    char z;
};

size_t n = sizeof(B); //8, так как sizeof(double)=8
```

ЗАМЕЧАНИЕ

Оптимизирующие компиляторы могут увеличивать размер объединения для выравнивания адресов элементов данных по границам, равным степени двойки.

Например:

```
union C{
    double d;
    char ar[11];
};
size_t n = sizeof(C); //VC: 16 байтов
```

10.4. Инициализация объединений

При создании переменной-объединения можно применять список инициализаторов (листинг 10.4). При этом компилятор использует для присваивания значения первое из объявленных в объединении полей.

Листинг 10.4. Инициализация объединений

//Вариант 1.

```
union U{
    int n;
    char ch[4];
};
int main()
{
    union U ob = {0x44332211}; //для инициализации компилятор
                               использует поле n
}
```

//Вариант 2.

```
union U{
    char ch[4];
    int n;
};
int main()
{
    union U ob = {0x11, 0x22, 0x33, 0x 44}; // для инициализации
                                              компилятор использует массив ch
    //Содержимое памяти, отведенной под объединение U, в обоих случаях
    будет идентичным.
}
```

10.5. Анонимные объединения (специфика Microsoft)

Иногда просто требуется сообщить компилятору, что необходимо разместить несколько переменных по одному и тому же адресу. Для этого можно использовать анонимное объединение.

Специфика: обращение к членам объединения происходит не посредством экземпляра, а просто по имени поля объединения (листинг 10.5).

Листинг 10.5. Использование анонимных объединений

```
int main()
{
    ...
    {
        union{ //это определение, поэтому компилятор выделит
                память, исходя из наибольшего поля объединения
                int n;
                char ch[4];
        };
        //до конца блока можно пользоваться полями объединения просто по имени
        n=10;
        ch[3] = 33;
    }
    ...
}
```



ПРИЛОЖЕНИЯ



Приложение 1

Представление данных

Перед тем как использовать переменные в выражениях, программист должен описать их свойства, чтобы компилятор смог:

- ☐ выделить требуемый объем памяти для хранения значения каждой переменной;
- ☐ правильно выбрать команды процессора для действий с разными переменными.

Знание того, как представляются данные разных типов в памяти и в регистрах процессора, каковы особенности выполнения действий с этими данными, позволит программисту избежать некоторых ошибок, а также повысить эффективность разрабатываемой программы.

П1.1. О системах счисления и изображении количеств

У меня на днях был день рождения —
23 года исполнилось.
Торт был со свечками, с пятью, все горели кроме второй.
Вот так! 10111

Для кодирования количеств (чисел) люди в своей повседневной практике используют так называемые *системы счисления*. В любой системе счисления для изображения количеств служат комбинации цифр.

Любая система счисления должна позволять человеку (а также и техническому устройству):

- ☐ записывать количества, используя минимум места (на бумаге, в памяти компьютера);

- выполнять действия с количествами, придерживаясь максимально простого набора правил.

Как раз этими двумя свойствами и различаются системы счисления.

Система счисления (СС) — это способ изображения чисел. В ходе своего развития человечество изобрело весьма много различных систем счисления. Привычная для нас десятичная СС, а также СС, которые используются в цифровой технике — двоичная и шестнадцатеричная — это системы со следующими свойствами:

- *позиционные* — вес единицы разряда числа зависит от позиции разряда (пример непозиционной системы — римская);
- *с постоянным основанием* системы счисления — количество различных цифр одинаково для любого разряда числа (пример системы с переменным основанием — система исчисления времени);
- *с естественным порядком следования весов* — вес единицы следующего разряда на 1 больше максимального числа, представимого всеми предыдущими разрядами;
- *с естественным представлением цифр в разрядах* (не кодированным) — для каждого разряда используется количество цифр, равное основанию системы счисления (пример системы с кодированным представлением цифр — система исчисления времени, где для 60 различных значений секунд и минут используется не 60 разных знаков, а десятичное представление чисел от 0 до 59).

Важное преимущество СС, обладающих перечисленным набором свойств, заключается в простоте правил, по которым выполняются арифметические действия. Например, m -разрядное число (где d_i — это обозначение цифры в записи числа) в любой СС (с упомянутым набором свойств и с основанием системы счисления) связано количеством N , изображаемым этим числом вот такой простой формулой (1):

$$N = d_{m-1} * B^{m-1} + d_{m-2} * B^{m-2} + d_{m-3} * B^{m-3} + \dots + d_2 * B^2 + d_1 * B^1 + d_0 * B^0 \quad (1)$$

В этой формуле:

- B — основание системы счисления;
- $(d_{m-1} d_{m-2} \dots d_1 d_0)_{(B)}$ — цифры в записи числа (нижний индекс в скобках здесь и далее означает основание системы счисления).

Десятичная система счисления имеет основание 10:

$$(1056)_{(10)} = (1 * 10^3 + 0 * 10^2 + 5 * 10^1 + 6 * 10^0)_{(10)} \quad (1a)$$

Двоичная система счисления имеет основание 2 (значения цифр — 0 и 1):

$$(101011)_{(2)} = (1 * 2^5 + 0 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0)_{(10)} = 43_{(10)} \quad (16)$$

ЗАМЕЧАНИЕ

Обратите внимание, что формулу (1) можно использовать для перевода в десятичную систему из любой другой, все действия при этом переводе придется выполнять только в привычной десятичной СС.

П1.2. Перевод чисел из одной системы счисления в другую

Перевод в систему счисления с основанием B можно выполнить по схеме, которую легко получить, преобразовав формулу (1).

$$N = (((((..... + d_i)B + d_{i-1})B + ... + d_2)B + d_1)B + d_0 \quad (2)$$

Из формулы (2) видно, что при делении на основание системы счисления сначала исходного числа, а затем предыдущих частных, остатки от целочисленного деления будут давать значения цифр, начиная с младшей.

Пример 1.

Пусть необходимо перевести число 79 в двоичную систему счисления.

Начинаем действия справа (табл. П1.1).

Первый шаг: делим заданное число (79) на основание системы счисления, в которую переводим (2). Записываем частное (39) и остаток от деления (он может быть только 0 или 1, в данном случае остаток = 1).

Второй шаг: делим полученное частное (39) на 2, записываем частное (19) и остаток от деления (1).

Третий шаг: в результате деления 19 на 2 получаем частное (9) и остаток (1).

Следующие шаги выполняются по аналогичной схеме до тех пор, пока на очередном шаге (в нашем случае — на 7-ом) не получим частного, равного 0. Это признак окончания процесса.

Цепочка остатков (слева направо — 1001111) дает последовательность цифр числа для записи в требуемой системе счисления (в нашем примере — в двоичной).

Таблица П1.1. Перевод числа 79 в двоичную систему счисления

7	6	5	4	3	2	1	Номер шага
1 / 2	2 / 2	4 / 2	9 / 2	19 / 2	39 / 2	79 / 2	Действие
0	1	2	4	9	19	39	Частное от деления на 2 ←
<u>1</u>	<u>0</u>	<u>0</u>	<u>1</u>	<u>1</u>	<u>1</u>	<u>1</u>	Остаток ←

Проверим правильность результата:

$$(1001111)_{(2)} =$$

$$= (1 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0)_{(10)} =$$

$$= (64 + 0 + 0 + 8 + 4 + 2 + 1)_{(10)} = 79_{(10)}$$

ЗАМЕЧАНИЕ

Мы выполнили преобразование, оперируя с количеством, исходно записанным в десятичной системе. Для того чтобы преобразовать число из любой системы счисления, необходимо уметь делить в исходной СС. Поскольку люди уверенно выполняют действия лишь в десятичной системе счисления, то перевод из какой-нибудь СС в любую другую обычно выполняют в два этапа: сначала из исходной СС переводят в десятичную по формуле (1), а затем из десятичной — в требуемую.

Пример 2.

Выполним перевод того же самого числа (79) в троичную систему (табл. П1.2) и тоже проверим правильность результата:

$$(2221)_{(3)} =$$

$$= (2 \cdot 3^3 + 2 \cdot 3^2 + 2 \cdot 3^1 + 1 \cdot 3^0) =$$

$$= (54 + 18 + 6 + 1)_{(10)} = 79_{(10)}$$

Таблица П1.2. Перевод числа 79 в троичную систему счисления

4	3	2	1	Номер шага
2 / 3	8 / 3	26 / 3	79 / 3	Действие
0	2	8	26	Частное от деления на 3 ←
<u>2</u>	<u>2</u>	<u>2</u>	<u>1</u>	Остаток ←

Пример 3.

Рассмотрим более подробно *шестнадцатеричную* систему счисления.

Эта система имеет основание 16. Для изображения шестнадцати цифр используются:

- привычные цифры: 0,...9;
- буквы латинского алфавита: A(10), B(11), C(12), D(13), E(14), F(15).

Шестнадцатеричная (далее — hex) система используется в вычислительной технике в основном для компактного изображения двоичных чисел (например, при отображении содержимого внутренних элементов процессора в окне экранного отладчика — регистров и элементов памяти).

Популярность шестнадцатеричной системы объясняется тем, что одна цифра в шестнадцатеричной записи однозначно соответствует четырехразрядной комбинации цифр в двоичной записи того же числа.

Это несложно увидеть из формулы (3), где каждая скобка соответствует одной шестнадцатеричной цифре.

$$\begin{aligned}
 N &= \dots + 2^7 * b_7 + 2^6 * b_6 + 2^5 * b_5 + 2^4 * b_4 + 2^3 * b_3 + 2^2 * b_2 + 2^1 * b_1 + 2^0 * b_0 = \\
 &= \dots + (2^3 * b_7 + 2^2 * b_6 + 2^1 * b_5 + 2^0 * b_4) * 2^4 + (2^3 * b_3 + 2^2 * b_2 + 2^1 * b_1 + 2^0 * b_0) = \\
 &= \dots + (2^3 * b_7 + 2^2 * b_6 + 2^1 * b_5 + 2^0 * b_4) * 16^1 + (2^3 * b_3 + 2^2 * b_2 + 2^1 * b_1 + 2^0 * b_0) * 16^0 = \\
 &= \dots + h_1 * 16^1 + h_0 * 16^0
 \end{aligned} \tag{3}$$

Условные обозначения в формуле:

- N — изображаемое количество;
- b_i — двоичные цифры;
- h_i — шестнадцатеричные цифры.

В табл. П1.3 приведено соотношение между шестнадцатеричными, двоичными и десятичными числами от 0 до 16. Таблицу соответствия между шестнадцатеричными цифрами от 0 до 0F₍₁₆₎ и их двоичными и десятичными эквивалентами следует выучить наизусть.

Пример перевода числа из шестнадцатеричной системы в десятичную:

$$\begin{aligned}
 7EA_{(16)} &= (7 * 16^2 + 14 * 16^1 + 10 * 16^0) = \\
 &= 7 * 256 + 14 * 16 + 10 = \\
 &= 1792 + 224 + 10 = 2026
 \end{aligned}$$

Таблица П1.3. Соотношение между шестнадцатеричными, двоичными и десятичными числами

Hex (шестнадцатеричные)	Dec (десятичные)	Bin (двоичные)
00	00	0 0000
01	01	0 0001
02	02	0 0010
03	03	0 0011
04	04	0 0100
05	05	0 0101
06	06	0 0110
07	07	0 0111
08	08	0 1000
09	09	0 1001
0A	10	0 1010
0B	11	0 1011
0C	12	0 1100
0D	13	0 1101
0E	14	0 1110
0F	15	0 1111
10	16	1 0000

Теперь обсудим действия над многоразрядными числами. Перечень действий, которые могут потребоваться при выполнении вычислений, достаточно велик, но более сложные действия (такие, как умножение, деление, вычисление элементарных функций) можно свести к некоторой (может быть, длинной) последовательности более простых действий. Поэтому далее в основном будем обсуждать только сложение и вычитание.

Вспомним правило сложения (вычитания) многоразрядных чисел, которому учили в школе.

Правило

Для сложения/вычитания многоразрядных чисел A и B следует складывать/вычитать отдельно цифры в каждом из разрядов, пользуясь таблицей сложения/вычитания, с учетом возможного переноса (заема) из предыдущего (младшего) разряда.

Сложение одноразрядных чисел выполняют, используя таблицу сложения. Для десятичной СС ее называют *таблицей Архимеда* и учат наизусть в первом классе школы (она обычно изображена на задней стороне обложки тетрадей в клетку). Таблица для десятичной СС содержит 100 (с учетом симметрии — 55 различающихся) клеток для различных сочетаний одноразрядных слагаемых. Для каждой пары одноразрядных слагаемых в таблице указано значение суммы и значение переноса в следующий разряд.

Аналогично можно составить таблицу вычитания для одноразрядных чисел.

Правило и таблицу используют при разработке схемотехники арифметико-логического устройства в процессорах.

П1.3. Использование различных систем счисления при технической реализации средств цифровой вычислительной техники

В электронных цифровых устройствах разные цифры чаще всего изображают различными значениями электрического напряжения. В ходе развития электронных цифровых устройств инженеры быстро пришли к выводу, что в этих устройствах наиболее выгодно изображать числа в двоичной системе счисления.

Один из наиболее распространенных в настоящее время стандартов предписывает изображать:

- двоичную цифру 0 — напряжением в интервале от 0 В до +0,4 В;
- двоичную цифру 1 — напряжением в интервале от +2,4 В до 5,0 В.

Обратите внимание, что между этими диапазонами оставлен значительный промежуток в 2,0 В (от 0,4 В до 2,4 В). Это сделано для того, чтобы затруднить искажение логического сигнала под действием разнообразных помех. Если бы использовалась не двоичная, а десятичная система счисления, то тот же интервал напряжений от 0 до 5,0 В пришлось бы разбить на 10 несмежных диапазонов для изображения десяти цифр, при этом промежутки между диапазонами соседних цифр были бы гораздо меньше, и соответственно, искажения значений цифр происходили бы под действием помех гораздо меньшей величины.

Человек привык к десятичной системе счисления, в которой даже большие величины изображаются числами с обозримым количеством цифр (разрядов). Напротив, двоичная система счисления требует для изображения той же величины гораздо большего (приблизительно в три с половиной раза) количества разрядов. Например, текущий 2007 год в двоичной системе счисления изображается числом 11111010111, содержащим 11 разрядов.

Визуальное восприятие и быстрая оценка на глаз значений многоразрядных двоичных чисел для человека затруднительны. Быстрый перевод из двоичной системы счисления в десятичную и обратно человеку также затруднительно выполнять в уме.

По этой причине в цифровой вычислительной технике используют *шестнадцатеричную* и (реже) *восьмеричную* системы счисления. Числа в этих системах счисления обозримы (содержат небольшое, по сравнению с двоичной, количество цифр), и из этих систем счисления легко преобразовать числа в двоичную систему (и обратно).

П1.4. Особенности выполнения арифметических операций в ограниченной разрядной сетке

Арифметико-логическое устройство (АЛУ) в цифровом процессоре может выполнять действия только с числами определенной (ограниченной) разрядности. В зависимости от значения этой разрядности различают процессоры:

- ☐ 8-разрядные;
- ☐ 16-разрядные;
- ☐ 32-разрядные;
- ☐ 64-разрядные.

При сложении формируется значение суммы, разрядность которой равна разрядности слагаемых. Это означает, что даже при выполнении такого простого действия, как сложение, некоторые комбинации операндов могут дать результат, не уместящийся в данном количестве разрядов (перенос за границу разрядной сетки).

Некоторые процессоры способны выполнять действия с целыми числами различной разрядности. Например, представители семейства Intel x86 (модели i386 или старше) могут работать с операндами длиной в 1 байт, либо 2 байта, либо 4 байта. Другие процессоры всегда выполняют действия с целыми операндами одной разрядности (например, в RISC-процессорах с архитектурой ARM — только с четырехбайтовыми). Это не значит, что в таких процессорах не используются операнды с меньшей (или с большей) разрядностью. Просто перед выполнением действия операнды меньшей длины преобразуются к четырехбайтовым.

В табл. П1.4 приведены значения, которые может принимать число, изображаемое в ограниченной разрядной сетке. Обратите внимание, что каждое

последующее число можно получить из предыдущего в результате прибавления единицы. Однако если это проделать с максимальным числом (содержащим единицы во всех разрядах), то в n -разрядной сетке будет получено (неверное) значение: 000....000.

Таблица П1.4. Диапазон беззнаковых значений, представимых в n -битовой сетке

HEX	BIN	DEC	К следующему значению
00...01	000...001	?	
00...00	000...000	?	+1
...FFF	111...111	$2^n - 1$	+1
...FFE	111...110	$2^n - 2$	+1
...FFD	111...101	$2^n - 3$	+1
.....			
00...02	000...010	2	+1
00...01	000...001	1	+1
00...00	000...000	0	+1

Представление чисел оказывается циклически повторяющимся (в табл. П1.4 граница цикла обозначена двойной линией). Цикличность представления и ограниченность диапазона приводят к тому, что итогом операции может быть либо правильный результат, либо выход за границу диапазона представимых значений (этот выход называют переполнением разрядной сетки).

Факт возникновения переполнения никак не отмечается в сформированном результате — он просто оказывается неверным! Однако переполнение регистрируется схемотехникой процессора: в любом процессоре имеется специальный регистр, называемый регистром состояний или регистром флагов. В этом регистре при возникновении переполнения устанавливается в 1 один из разрядов — флаг переноса (беззнакового переполнения). По-английски его называют C-flag (от Carry — перенос).

П1.5. Изображение знакопеременных величин

Теперь рассмотрим, как изображают целые знакопеременные числа. Естественный способ представления отрицательных чисел можно получить, последовательно вычитая единицу из нуля. Такое представление (табл. П1.5)

называется *дополнительным кодом* (английский эквивалент — two’s complement).

Представление чисел со знаком в дополнительном коде также будет циклическим.

Таблица П1.5. Диапазон знакопеременных значений, представимых в *n*-битовой разрядной сетке

HEX	BIN	DEC	К предыдущему значению	Примечание
0...FFF	011...111	$2^{n-1}-1$		Наибольшее положительное число
			+1	
00...02	000...010	2	+1	
00...01	000...001	1	+1	
00...00	000...000	0	± 1	Ноль
F...FFF	111...111	-1	-1	
F...FFE	111...110	-2	-1	
...	...		-1	
80...00	100...000	-2^{n-1}		Наименьшее отрицательное число

Из таблицы видно, что положительные числа получаются последовательным прибавлением единицы к предыдущему (меньшему) значению (начиная с нуля), а отрицательные — вычитанием единицы из предыдущего (большого) значения, также начиная с нуля.

Обратите внимание на то, что в представлении знакопеременных чисел кодовые комбинации от 100...000 до 111...111 переехали вниз таблицы, и теперь используются для изображения отрицательных количеств. Максимальное положительное число, которое можно представить, используя *n*-разрядный дополнительный код, приблизительно вдвое меньше того, которое было в представлении знакопостоянных величин в той же *n*-разрядной сетке.

Таблица П1.6 демонстрирует, как одни и те же кодовые комбинации используются для изображения различных диапазонов значений. Средние столбцы содержат граничные значения величин в общем виде и для байтовой разрядной сетки, т. е. для *n* = 8.

Таблица П1.6. Дополнительный код: соотношение между представлением беззнаковых и знакопеременных величин

Беззнаковые величины	Граничные значения	Для байтовой разрядной сетки	Знакопеременные
Двоичные комбинации			Двоичные комбинации
<u>111...111</u>	$2^n - 1$	<u>+255</u>	Не представимы
<u>100...001</u>		<u>+129</u>	
100...000		+128	
011...111	$2^{n-1} - 1$	+127	011...111
...	...		...
000...000	0	0	000...000
Не представимы	-1	<u>-1</u>	<u>111...111</u>
	$-2^{n-1} - 1$	<u>-127</u>	<u>100...001</u>
	-2^{n-1}	-128	100...000

Обратите внимание на то, что одна и та же кодовая комбинация может обозначать различные количества, в зависимости от того, как ее интерпретировать — как знаковую либо как беззнаковую.

Например, комбинация 10000001 означает:

- ☐ +129 — при беззнаковой интерпретации;
- ☐ -127 — при знаковой интерпретации.

Комбинация 11111111 это:

- ☐ -1 — при знаковой интерпретации;
- ☐ +255 — при беззнаковой интерпретации (в таблице соответствующие значения подчеркнуты).

Интерпретацию кодов определяет программист и сообщает об этом компилятору, когда указывает в описаниях тип переменных. Компилятор же при трансляции использует соответствующие (различные) процессорные команды для обработки переменных разных типов.

Примеры таких ситуаций:

- знаковое и беззнаковое целочисленное умножение;
- команды преобразования коротких целочисленных форматов в длинные;
- деление целочисленных переменных на степень двойки путем сдвига (см. разд. П1.9, П1.10, П1.12).

Отметим некоторые особенности кодирования знакопеременных количеств дополнительным кодом:

- все неотрицательные числа содержат подряд один или более нулей в старшей (левой) части двоичного кода. При этом: чем меньше модуль числа, тем больше нулей там будет (в частности, число +1 содержит в n -битовом коде $(n-1)$ старших нулей);
- все отрицательные числа содержат подряд одну или более единиц в старшей (левой) части двоичного кода. При этом: чем меньше модуль числа, тем больше единиц там будет (в частности, число -2 содержит в n -битовом коде $(n-1)$ старших единиц, а число -1 содержит n старших единиц, т. е. просто все до единой);
- старший бит знакопеременного целочисленного операнда можно рассматривать как знаковый (0 означает плюс, а 1 — минус);
- одни и те же кодовые комбинации в интервале от 100...000 до 111...111 при кодировании беззнаковых величин и при кодировании величин со знаком представляют различные количества, в то время как кодовые комбинации от 000...000 до 011...111 — одни и те же количества, независимо от типа величины;
- для представления чисел используются все кодовые комбинации, возможные в данной разрядной сетке, т. е. любой последовательности единиц и нулей в коде соответствует какое-то числовое значение (при представлении чисел в форматах плавающей точки это будет не так!);
- замечательное свойство дополнительного кода для изображения знакопеременных чисел состоит в том, что действия сложения и вычитания с дополнительным кодом можно выполнять по правилу "в столбик", безотносительно к знакам операндов. При этом лишь следует учитывать возможность того, что результат действия (иногда) может выйти за границы представимого диапазона значений.

Дополнительный код используется для изображения знакопеременных величин во всех современных процессорах. Однако встречаются случаи, когда в отдельных специальных случаях знакопеременные величины кодируются двоичными словами по-иному (пример см. в разд. П1.11.2).

П1.6. Выявление переполнений при выполнении сложения и вычитания

Цикличность представления и ограниченность диапазона приводят к тому, что итогом операции может быть либо правильный результат, либо выход за границу диапазона представимых значений, т. е. переполнение разрядной сетки.

Цикличность представления целых чисел в ограниченной разрядной сетке можно изобразить с помощью круговой диаграммы (рис. П1.1 и П1.2). Эта диаграмма наглядно иллюстрирует переход к следующему числу путем прибавления единицы, переход к предыдущему — путем вычитания единицы, сложение чисел — как сложение эквивалентных углов, а также хорошо видны явления переполнения для беззнаковых чисел и чисел со знаком.

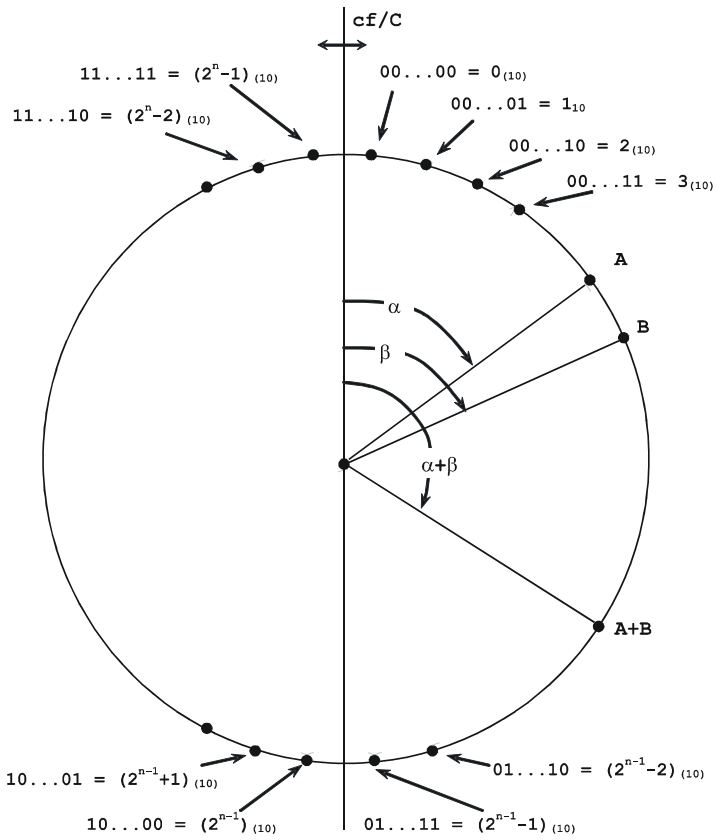


Рис. П1.1

Для выявления выхода за границы представимого диапазона используется флаг — бит переноса в регистре состояний процессора (обозначается *cf* или *c* — от Carry). При выполнении действий АЛУ устанавливает этот бит в 1, если при интерпретации операндов как беззнаковых, результат операции выйдет за границы представимого диапазона.

Другая диаграмма (рис. П1.2) иллюстрирует кодирование знакопеременных величин.

Для выявления выхода результата операции за границы представимого диапазона в регистре состояний имеется еще один флаг — бит арифметического переполнения (обозначается *of* или *v* — от oVerflow). При выполнении действий АЛУ устанавливает этот бит в 1, если при интерпретации операндов как знакопеременных, результат операции выйдет за границы представимого диапазона.

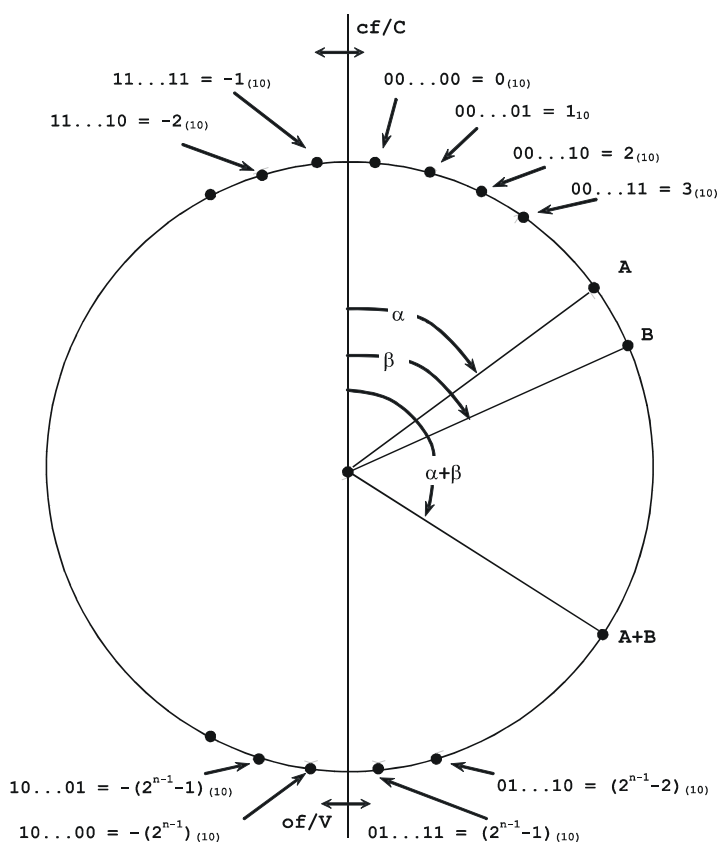


Рис. П1.2

ЗАПОМНИТЕ!

По внешнему виду операндов невозможно заключить, изображают ли кодовые комбинации положительные (беззнаковые) числа или знакопеременные. Об этом знает только программист, когда он объявляет типы переменных. Получив объявления переменных, о них узнает компилятор и учтет при генерации машинных команд.

Одна и та же пара кодов при выполнении действия (например, сложения) может дать в результате переполнение, если эти коды изображают беззнаковые числа, и напротив — дать верный результат, если коды изображают числа со знаком. О том, произошло ли переполнение, можно узнать, проверив значение соответствующего флага после выполнения операции.

Пример.

Допустим, в байтовой разрядной сетке выполняется сложение кодов:

1111 1101 и 0000 0101.

Сложение столбиком:

$$\begin{array}{r} 1111\ 1101 \\ 0000\ 0101 \\ \hline \end{array}$$

0000 0010 — результат (при этом возникает перенос за границу разрядной сетки)

При выполнении этого действия схемотехника АЛУ устанавливает флаг *cf* в 1 и сбрасывает флаг *of* в 0.

Если слагаемые интерпретируются как беззнаковые (*unsigned char*), то они изображают количества 253 и 5, а их сумма равна 258, что превышает максимальное значение 255, которое можно представить в байтовой разрядной сетке. Это означает, что произошло (беззнаковое) переполнение, о чем сигнализирует установившийся флаг *cf* (флаг беззнакового переполнения).

Если же коды изображают знакопеременные операнды, то первому коду соответствует значение -3 , а второму — по-прежнему 5. Сумма этих двух значений равна $+2$, выход за границу представимого диапазона значений ($-128\dots+127$) отсутствует, о чем свидетельствует сброшенный флаг *of* (флаг знакового переполнения).

Для учета состояния флагов в системах команд процессоров всегда имеются команды *с условным исполнением*. Такая команда выполняет действие, если имеет место указанная в ней комбинация флагов, или же она ведет себя как пустая команда, если состояния флагов не соответствуют заданным.

В большинстве процессоров условное исполнение реализовано только для команд перехода (команды условного перехода). В семействе *x86* имеется

также группа команд условного копирования CMOVcc (Conditional MOVe). Однако в некоторых процессорах команд с условным исполнением может быть гораздо больше (например, в процессорах с архитектурой ARM большинство команд являются командами с условным исполнением).

П1.7. Смена знака целого знакопеременного числа

Как поменять знак числа, изображенного в дополнительном коде? Для получения правила (алгоритма) продelaем простые преобразования, взяв за исходное некоторое число k :

$$-k = (1 - 1) - k = (-1 - k) + 1$$

Теперь учтем, что -1 изображается кодом $111\dots11$.

Действие $(-1 - k)$ всегда возможно, и при его выполнении ни в одном из разрядов не происходит заема. Его можно реализовать инвертированием всех битов кода k . Это легко увидеть, выполнив вычитание столбиком:

$$\begin{array}{r} \underline{1\ 1\ 1\ 1\ \dots\ 1\ 1} \text{ — код числа } (-1) \\ 1\ 0\ 0\ 1\ \dots\ 0\ 1 \text{ — код вычитаемого (число } k) \\ \hline 0\ 1\ 1\ 0\ \dots\ 1\ 0 \text{ — результат (инверсия вычитаемого)} \end{array}$$

Таким образом, правило получения кода для числа с противоположным знаком выглядит так: проинвертировать (заменить противоположными значениями) все биты исходного двоичного числа, а затем прибавить к результату единицу (по правилу в столбик).

В наборе процессорных команд обычно имеется специальная команда для выполнения этого действия. Отметим, что если процессор умеет выполнять вычитание, то смену знака можно выполнить вычитанием из нуля.

Обратите внимание на то, что смена знака наименьшего отрицательного числа $100\dots00$ даст неверный результат. Это следствие того факта, что диапазон представимых значений в дополнительном коде несимметричен относительно нуля (отрицательных чисел на одно больше, нежели положительных).

П1.8. Действия с повышенной разрядностью

Стандарт языка Си предусматривает четыре разновидности длин целочисленных операндов:

- ☐ один байт (char);
- ☐ два байта (short или int);

- четыре байта (int или long);
- восемь байтов (long long).

Исторически так сложилось, что тип int был привязан к разрядности процессора, т. е. имел длину, совпадающую с длиной машинного слова (операнда, с которым АЛУ данного процессора могло справиться в рамках одной команды).

В то же время требуется, чтобы можно было проводить вычисления с переменными любого допустимого стандартом целого типа на любых процессорах (в т. ч. и на малоразрядных). Это оказывается всегда возможным, поскольку действия с длинными переменными можно выполнять, используя несколько коротких процессорных команд, которые формируют значение результата по частям. Например, для сложения длинных операндов можно использовать несколько процессорных команд сложения, причем первая команда складывает младшие части операндов, а все остальные команды сложения должны учитывать и возможные переносы, возникшие при сложении младших частей.

Такая возможность (и в частности, учет переносов) поддерживается во всех процессорах на аппаратном уровне, а соответствующая техника программирования носит название — операции с повышенной разрядностью.

П1.9. Особенности умножения и деления целых двоичных чисел

При умножении разрядность получаемого результата может превысить (одинаковые) разрядности сомножителей вдвое. Это означает, что (возможно) придется для результата использовать переменную удвоенной длины. Обычно команды умножения в процессоре реализованы именно по такой схеме, когда на выходе блока умножения формируется результат удвоенной (по сравнению с сомножителями) длины. К сожалению, при использовании дополнительного кода замечательное свойство инвариантности результата операции к знаковости операндов имеет место только для действий сложения и вычитания. Умножение одних и тех же двоичных кодов сомножителей дает различные результаты в зависимости от того, являются сомножители беззнаковыми либо знакопеременными.

Пример.

Пусть умножаются два байтовых сомножителя: 1111 1010 и 0000 0011.

Если эти коды изображают беззнаковые величины, то они равны соответственно 250 и 3, а их произведение равно 750 (в двоичном виде 0000 0010 1110 1110).

Если те же коды изображают знакопеременные величины, то они равны -6 и 3 , а их произведение равно -18 (в двоичном виде $1111\ 1111\ 1110\ 1110$).

Как видим, результат умножения изображается различными кодовыми комбинациями в зависимости от знаковости операндов, хотя коды сомножителей одинаковы. Этим фактом объясняется наличие в процессорах двух вариантов команды целочисленного умножения (знакового и беззнакового). Одновременно отметим тот факт, что если результат умножения имеет удвоенную длину, то в ней переполнение не может возникнуть ни при каких комбинациях значений сомножителей.

Однако при использовании такого умножения только программист может решить, что делать с результатом удвоенной длины (ведь нельзя же каждый раз при выполнении умножения удваивать длину используемых операндов).

Если сравнить результаты знакового и беззнакового умножения в приведенном примере, то можно отметить, что они различаются только в старших частях, в то время как младшие — идентичны. Это не случайное совпадение для приведенного примера, а общая закономерность выполнения умножения при использовании дополнительного кода. Этим объясняется тот факт, что наряду с командами умножения, формирующими результат удвоенной длины, в некоторых процессорах имеются и команды, которые дают результат умножения с той же длиной, что и сомножители. При использовании таких команд гораздо выше шансы на возникновение переполнения, но зато можно применять одну и ту же команду, как для знаковых, так и для беззнаковых операндов.

И снова подчеркнем, что контроль переполнения при любых арифметических действиях является исключительно заботой программиста.

П1.10. Приведение типов данных

В языке Си (как, впрочем, и в других) используется несколько целых типов данных, различающихся размером:

- `char` — один байт;
- `short` — два байта;
- `int` — два либо четыре байта (зависит от разрядности процессора);
- `long` — четыре байта;
- `long long` — восемь байтов.

Рассмотрим на примерах, как будет выглядеть двоичный код одного и того же числа в различных типах данных.

Пример 1.

Число +6 будет выглядеть:

- в байтовой разрядной сетке (signed или unsigned char) — 0000 0110;
- в двухбайтовой (signed или unsigned short) — 0000 0000 0000 0110;
- в четырехбайтовой — 0000 0000 0000 0000 0000 0000 0000 0110.

Пример 2.

Число -3 будет выглядеть:

- в байтовой разрядной сетке (signed или unsigned char) — 1111 1101;
- в двухбайтовой (signed или unsigned short) — 1111 1111 1111 1101;
- в четырехбайтовой — 1111 1111 1111 1111 1111 1111 1111 1101.

Вы можете легко убедиться в правильности последних трех строк, если прибавите (в столбик, со всеми переносами!) к любому из приведенных чисел число +3 (его код —0011).

Во всех случаях у вас (в той же разрядной сетке) получится 0 (так и должно быть, поскольку $(-3) + (+3) = 0$).

Из приведенных примеров нетрудно по индукции вывести правило приведения более короткого типа к более длинному:

- для беззнаковых (unsigned) типов надо заполнить левую часть более длинного типа нулями;
- для знакопеременных надо заполнить левую часть более длинного типа значением старшего (знакового) бита исходного, более короткого типа. Именно так работает оператор преобразования типа языка Си (cast operator).

Например:

- (short) 0001 0111 = 0000 0000 0001 0111 (результат — со знаком);
- (int) 1100 1101 = 1111 1111 1111 1111 1111 1111 1100 1101 (результат — со знаком);
- (unsigned int) 1100 1101 = 0000 0000 0000 0000 0000 0000 1100 1101 (результат — без знака).

Правоммерно задать обратный вопрос — а что, если начнем приводить более длинный тип к более короткому?

Компилятор просто усечет (отбросит) старшую часть более длинного операнда, но при этом программист должен быть уверен в том, что во-первых — отбрасыва-

ваемая часть содержит только одинаковые цифры (все нули либо все единицы), и во-вторых — что после усечения знакового операнда знак числа не изменился, т. е. в результирующем коротком типе старший бит имеет то же значение, что и биты в отброшенной левой (старшей) части. Если это не так, то значит, что значение исходной переменной не представимо (не помещается) в короткой разрядной сетке — приведение выполнить невозможно!

Приведем фрагмент кода, в котором такая проверка делается. Производится сравнение длинного операнда с границами, в которых тот должен находиться, чтобы была возможность привести к короткому типу.

```
int x = значение;
```

```
char c;
```

```
if(x>=-128 && x<=127) c=x; //можно присвоить без возникновения ошибки
```

Еще одно желание, которое может возникнуть — привести беззнаковый тип к знаковому (или наоборот) без изменения разрядности. Такое приведение тоже не всегда будет выполнено компилятором верно (*см. разд. П1.12*).

Прежде всего, отметим, что приведение, безусловно, возможно лишь в том случае, если диапазон значений приводимого типа целиком содержится в (большем) диапазоне типа, к которому выполняется приведение. Это имеет место лишь в случае, когда более короткий беззнаковый тип приводится к более длинному знакопеременному. Во всех остальных случаях возможно возникновение ошибки.

Например, приведение знакового типа к (сколь угодно длинному) беззнаковому невозможно, если исходная переменная отрицательна. Точно так же, приведение беззнакового типа к знакопеременному той же разрядности приведет к ошибке в половине случаев, т. к. диапазон положительных значений для знакового типа вдвое уже, чем у беззнакового той же длины.

Следует помнить, что компилятор *не контролирует* возможность возникновения ошибок при приведении типов — это целиком обязанность программиста.

Общая рекомендация может состоять в том, что не следует ни в каких операциях смешивать беззнаковые и знакопеременные операнды, если нет твердой уверенности в том, что компилятор верно выберет эквивалентные процессорные команды при трансляции. А твердая уверенность может быть лишь тогда, когда программист отчетливо понимает возможности и ограничения уровня процессорных команд.

П1.11. Числа с плавающей точкой

Еще раз отметим некоторые свойства целочисленного формата представления величин:

- разрешающая способность (минимальное изменение величины, которое можно отобразить), равна 1 (при любом числе разрядов n);
- диапазон представимых значений однозначно определяется количеством разрядов n в разрядной сетке — 2^n , т. е. разрешающая способность и диапазон связаны однозначно при данном количестве битов в представлении числа.

Целочисленное представление хорошо подходит для таких задач, в которых величины подсчитываются в штуках (хороший пример — задача учета персонала или посетителей в учреждении).

Во многих других случаях необходимо иметь разрешающую способность, отличную от единицы. Например, контроль координаты, когда разрешающая способность определяется необходимой точностью контроля (скажем — 0,01 мм в металлорежущем станке). В таких случаях часто используют прием, называемый масштабированием, когда единице двоичного кода сопоставляют величину, соответствующую требуемой разрешающей способности (в нашем примере — 0,01 мм, при 16-битовой разрядной сетке диапазон представимых значений от 0 до $65535 * 0,01 = 625,35$ мм). Масштабирование в данном случае — это действие, которое выполняется только в голове у программиста.

Почти эквивалентный прием: можно сместить положение разделителя целой и дробной части так, что часть двоичного слова (левее разделителя) будет представлять целую часть числа, а другая часть справа от разделителя — дробную. Для такого представления можно расширить уже приведенное разложение количества (см. формулу 1б) по степеням основания системы счисления:

$$N = d_{m-1} * 2^{m-1} + d_{m-2} * 2^{m-2} + d_{m-3} * 2^{m-3} + \dots + d_2 * 2^2 + d_1 * 2^1 + d_0 * 2^0 + d_{-1} * 2^{-1} + d_{-2} * 2^{-2} + \dots$$

Здесь: $d_{-1}, d_{-2}, \dots$ — цифры в дробной части числа.

В этом случае вес единицы младшего разряда кода составит отрицательную степень двойки: 2^{-k} .

зн						...	...	d_1	d_0	d_{-1}	d_{-2}	d_{-3}	...	d_{-k}
----	--	--	--	--	--	-----	-----	-------	-------	----------	----------	----------	-----	----------

Этот прием можно реализовать не только в голове, но и в аппаратуре (в частности, в АЛУ). Для действий умножения АЛУ можно спроектировать так,

что при формировании результата положение разделителя в нем будет оставаться на принятом фиксированном месте. Отметим, что действия сложения и вычитания чисел в дополнительном коде будут выполняться правильно для произвольного положения разделителя.

Пока еще точность и диапазон по-прежнему связаны.

Такое представление дробных чисел (его называют форматом с фиксированной точкой) подходит только для узкого класса задач, где требуется представление величин с данной определенной разрешающей способностью.

Следующий шаг в направлении повышения универсальности формата представления чисел — сделать положение разделителя переменным и где-то это положение указывать (необходимо для каждого числа отдельно). Часть двоичного слова, изображающую значащие цифры, принято называть *мантиссой*, а другую часть, задающую положение разделителя — *порядком*. Точность и диапазон после этого становятся при этом не зависимыми друг от друга.

Значение числа можно представить следующим образом:

Знак	Мантисса	Порядок
------	----------	---------

Точность — задается количеством знаков (битов) в поле мантиссы.

Диапазон — задается количеством битов в поле порядка (оно определяет, насколько можем сдвигать разделитель).

Действия над такими числами выполнять сложнее.

П1.11.1. Неоднозначность представления и нормализованная форма

Число в формате плавающей точки (ПТ) можно представить так:

$\pm \text{Мантисса} * \text{Основание}^{\pm \text{Порядок}}$

Однако такое представление неоднозначно:

$1.2345 * 10^2 = 12.345 * 10^1 = 0.12345 * 10^0$

Выбирают одно из представлений, как стандартное — такое представление называют *нормализованным*.

Например, во многих инженерных калькуляторах в качестве нормализованного выбирается представление, у которого мантисса содержит ровно один ненулевой десятичный разряд в целой части мантиссы:

$\pm x.xxxxxxx \pm xx$. — так это выглядит на дисплее калькулятора
мантисса порядок

Двоичное представление чисел в формате ПТ. Нормализация

Если определенным образом выбрано нормализованное представление, то при записи в двоичном виде старший разряд мантиссы — всегда двоичная 1 (ее можно не запоминать — сэкономим бит).

Процедура нормализации состоит в следующем: сдвигаем мантиссу так, чтобы получить нормализованное представление, и при этом корректируем порядок, чтобы сохранить величину числа неизменной.

Пример выполнения нормализации:

Исходное число:

$101101.110001000 * 10^{1001}$ (в этой записи все числа двоичные).

Пусть выбрано нормализованное представление, в котором мантисса содержит один значащий разряд в целой части (его значение всегда равно 1).

При нормализации сдвигаем мантиссу вправо на 5 разрядов (это эквивалентно делению мантиссы на $2^5 = 32$) и корректируем порядок (увеличиваем его на $5_{(10)} = 101_{(2)}$).

Нормализованное представление:

$1.011011100010... * 10^{1110}$ (все числа двоичные).

Опуская неявный бит, получим мантиссу:

011011100010...

и порядок:

..01110.

Уточним некоторые понятия.

Нормализованное число содержит в мантиссе ровно один разряд в целой части, он отличен от нуля, а порядок находится в допустимых пределах, определяемых (ограниченной) разрядностью поля порядка.

Ненормализованное число может содержать в целой части мантиссы более одного (или ни одного) разряда, отличного от нуля. Если выполнить операцию нормализации, то оно, возможно, превратится в нормализованное. Такое число часто (но не всегда) получается как результат выполнения операций над числами ПТ.

Денормализованное число — это число, которое в данных разрядностях мантиссы и порядка невозможно нормализовать (поскольку величина числа слишком мала).

ВАЖНОЕ ЗАМЕЧАНИЕ 1

При нормализации, когда приходится сдвигать мантиссу вправо, возможна потеря точности при выходе младших битов мантиссы вправо за границу ее разрядной сетки.

ВАЖНОЕ ЗАМЕЧАНИЕ 2

Нормализация не всегда возможна, если при коррекции порядка происходит выход за границу его разрядной сетки. Если число слишком велико, его просто нельзя представить в выбранном формате ПТ (не хватает поля порядка). Если число слишком мало, то его можно представить, но лишь в денормализованной форме.

НЕ ОЧЕНЬ ВАЖНОЕ ЗАМЕЧАНИЕ

Кодовые комбинации, соответствующие ненормализованным числам, не используются для внешнего представления результатов действий (но получаются в ходе выполнения действий).

Нормализация нужна для представления мантиссы с максимально возможной в выбранной разрядной сетке точностью. Неявный бит возможен только в нормализованном представлении.

Действия с числами ПТ

Рассмотрим особенности выполнения действий в формате ПТ, вызванные ограниченностью разрядной сетки для представления мантиссы и порядка. Последующие примеры выполняются в десятичной системе счисления. Для обозримости чисел в примерах использована разрядная сетка, в которой мантисса имеет четыре десятичных разряда, а для порядка отведен лишь один десятичный разряд.

Для сложения/вычитания чисел ПТ следует:

- 1) Выводить порядки.
- 2) Сложить/вычесть мантиссы.
- 3) Нормализовать результат (если необходимо).

Пример выполнения операции сложения в формате ПТ представлен на рис. П1.3. При выравнивании порядков и при нормализации результата произошла потеря точности из-за того, что некоторые значащие цифры не помещаются в выбранную разрядную сетку. При выполнении сложения мантисс результат выходит за выбранную разрядную сетку влево. Чтобы можно было выполнить это действие без потери левой значащей цифры, вычислитель ПТ должен иметь более широкую внутреннюю разрядность. Кроме того, это вызывает необходимость последующей нормализации результата.

9.876*10² + 5.432*10¹

=

1.04192

9.876

+

0.5432

2

*

10²

*

10²

+

2

*

10²

=

1.04192

=

1.04192

=

1.042

← выравнивание порядков – денормализация произошла потеря точности

← результат ненормализованный

← результат нормализован, снова потеря точности

← результат после округления

← границы выбранной разрядной сетки

Рис. П1.3

Пример выполнения вычитания приведен на рис. П1.4. При выравнивании порядков и здесь происходит потеря точности — значащая цифра выходит за границу выбранной разрядной сетки. Однако если вычислитель ПТ имеет более широкую внутреннюю разрядность, то младшие разряды при нормализации могут "вернуться".

1.003*10⁻⁸ - 9.982*10⁻⁹

=

0.0048

1.003

-

0.9982

2

*

10⁻⁸

*

10⁻⁸

-

8

*

10⁻⁸

=

0.0048

=

0.048

← выравнивание порядков – денормализация произошла потеря точности

← результат ненормализованный

← результат не удастся нормализовать, т. к. порядок достиг предельного значения в выбранной разрядной сетке

← границы выбранной разрядной сетки мантиссы

Рис. П1.4

Для умножения (деления) чисел в формате ПТ следует:

1. Перемножить (поделить) мантиссы.

2. Сложить (вычесть) порядки.

Пример выполнения умножения чисел ПТ приведен на рис. П1.5. В данном примере сомножители имеют такие (большие) значения, что результат не удастся нормализовать, этому препятствует достигший своего наибольшего значения порядок. Это означает, что произошло арифметическое переполнение, результат невозможно выразить в выбранной разрядной сетке.

И наконец, рассмотрим два примера деления чисел ПТ (рис. П1.6).

В случае а) порядок ненормализованного результата деления находится в допустимых пределах, но его невозможно нормализовать, поскольку значение порядка — минимально возможное в выбранной разрядной сетке.

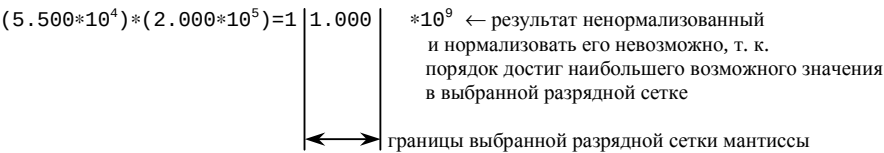


Рис. П1.5

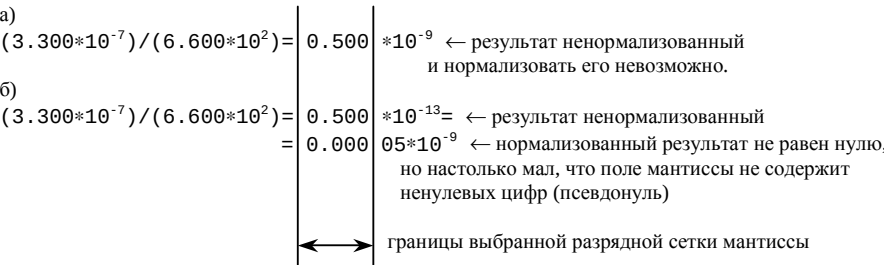


Рис. П1.6

В случае б) результат деления также получается ненормализованным, при этом значение порядка имеет недопустимое в данной разрядной сетке значение -13 . Результат деления можно нормализовать, но в ходе нормализации сдвиг мантиссы вправо приводит к выходу всех ее ненулевых разрядов за границу разрядной сетки. Такой (ненулевой) результат невозможно изобразить в выбранной разрядной сетке как отличный от нуля. Подобный результат операции ПТ в ограниченной разрядной сетке называют термином *псевдонуль*.

П1.11.2. Форматы представления чисел ПТ двоичным кодом

Знак числа, поле порядка и поле мантиссы располагаются в двоичном коде числа следующим образом:

Знак	Порядок в формате со смещением	Мантисса всегда в прямом коде (с опущенным неявным битом)
------	--------------------------------------	---

Знак относится ко всему числу.
Поле порядка содержит сумму истинного порядка и константы, называемой смещением.

Поле мантиссы содержит ее значение в прямом (не в дополнительном) коде с опущенным неявным битом.

До 1985 года на разных вычислительных платформах использовали форматы плавающей точки с разными размерами полей. Основной недостаток такой ситуации состоял в том, что при вычислениях на разных компьютерах степень потери точности была различной. Как следствие, одна и та же программа на одном компьютере (с большой длиной полей чисел ПТ) давала результат с малой (допустимой) ошибкой. На других машинах (где длина полей чисел ПТ была меньше) ошибка была недопустимо большой. Наконец, на третьих ЭВМ (длины полей еще короче) программа могла аварийно остановиться, в частности — из-за возникновения нуля в знаменателе выражения, который вычислялся как малая разность больших чисел (например, при решении систем линейных уравнений).

Такая неодинаковость результатов была совершенно недопустимой, и в 1985 году в США был принят стандарт на форматы чисел с плавающей точкой, которому с тех пор следуют все разработчики вычислителей ПТ (реализованных как аппаратно, так и программно).

При выборе формата плавающей точки разработчики стандарта выбирали:

- ☐ разрядность мантиссы и порядка;
- ☐ формат (с явным или неявным битом);
- ☐ вид нормализованного представления;
- ☐ величину смещения при изображении порядка;
- ☐ расположение полей в коде числа с плавающей точкой (оно выбрано таким, как указано в *разд. П1.11.2*).

П1.11.3. Стандарт на числа ПТ ANSI/IEEE 754-1985

Стандарт предусматривает три формата чисел ПТ (табл. П1.7):

- ☐ короткое вещественное (KB) — соответствует типу `float` языка Си;
- ☐ длинное вещественное (ДВ) — соответствует типу `double` языка Си;
- ☐ временное вещественное (ВВ).

Формат временного вещественного (ВВ) стандарт предписывает использовать только внутри вычислителей с плавающей точкой (реализованных как аппаратно, так и программно). Параметры временного вещественного выбраны так, чтобы при действиях в этом формате величина ошибок округления, возникающих при нормализации результатов операций, не превышала разрешающей способности формата длинного вещественного.

Таблица П1.7. Форматы чисел ПТ в стандарте ANSI/IEEE 754-1985

Русское наименование формата	Короткое вещественное (KB)	Длинное вещественное (ДВ)	Временное вещественное (ВВ)
Английское наименование	Short Real (SR)	Long Real (LR)	Temporary Real (TR)
Всего битов (байтов)	32 (4)	64 (8)	80 (10)
Поле порядка (битов)	8	11	15
Смещение порядка	$2^7 - 1 = 127$	$2^{10} - 1 = 1023$	$2^{14} - 1 = 16383$
Диапазон десятичных порядков	$-37 \div +38$	$-307 \div +308$	$-4931 \div +4932$
Поле мантиссы (битов)	23	52	64
Точных значащих десятичных цифр	6	15	19
Неявный бит	Опущен	Опущен	Изображается

После того как выражение вычислено, перед присваиванием его переменной (в зависимости от ее типа) результат вычисления преобразуется в `float` либо в `double`. При этом преобразовании всегда происходит потеря точности из-за округления.

В соответствии со стандартом ANSI/IEEE 754-1985 предусмотрено четыре типа (правила) округления:

- ☐ к ближайшему значению (по школьным правилам округления);
- ☐ по направлению к $+\infty$ (с избытком);
- ☐ по направлению к $-\infty$ (с недостатком);
- ☐ по направлению к 0 (усечение, т. е. просто отбрасывание лишних разрядов — выполняется легко и быстро).

Программист уровня Ассемблера имеет возможность выбрать (включить) нужный режим округления путем изменения двухбитового поля в одном из

управляющих регистров сопроцессора. Выбор подходящего режима округления позволяет в некоторых случаях минимизировать ошибки, возникающие при программировании численных методов в вычислительной математике.

Обратите внимание на количество десятичных разрядов, с которым представляет результаты программа "Калькулятор" ОС MS Windows. Оно равно 32 (а не 15, как можно было бы заключить из данных таблицы). Этот пример демонстрирует возможность программной организации вычислений с повышенной разрядностью и для форматов плавающей точки.

ВАЖНЕЙШЕЕ ЗАМЕЧАНИЕ

После того как программа оттранслирована (и, возможно, загружена в память), нет никакой возможности достоверно определить, глядя в определенные адреса памяти, что расположено по этим адресам: машинная команда, целое число, знаковое или беззнаковое, число ПТ, текстовая строка или что-то еще. Выбрав соответствующие опции трансляции, можно включить в исполняемый файл отладочную информацию (целесообразно делать на этапе отладки). Однако после того как программа отлажена, такую информацию обычно удаляют из исполняемого файла. Почему же процессор догадывается, как следует правильно выполнять программу? А потому что компилятор (на этапе трансляции, когда у него были описания для каждого элемента данных) индивидуально в каждой команде установил правильное соответствие между кодом операции и местоположением операндов — только эта информация и содержится в командах.

В заключение рассмотрим пример представления числа в формате с плавающей точкой.

Представим величину 13,375 в формате короткого вещественного (float).

$$(13,375 = 13 + 3/8 = 8 + 4 + 1 + 1/4 + 1/8 = 2^3 + 2^2 + 2^0 + 2^{-2} + 2^{-3})_{(10)} = \\ = (01101.0110 = 1.101011 * 10^{011})_{(2)}$$

Исходное число представлено суммой степеней двойки, а затем переведено в двоичную систему и нормализовано.

Значение поля знака (один бит) равно 0 (т. е. знак плюс).

Значение поля порядка (восемь битов) равно сумме истинного порядка + смещение:

$$011 + 1111111 = 10000010$$

Мантисса равна 1.101011

Старший (скрытый) бит в мантиссе не изображается, поле мантиссы (23 бита) равно:

$$1010110\ 00000000\ 00000000$$

Таким образом, код величины 13.375 в формате float имеет вид (длина 4 байта):

$$0\ 10000010\ 1010110\ 000000000000000000000000_{(2)} = 0x41560000_{(16)}$$

Поля знака, порядка и мантиссы в предыдущей строке разделены пробелами, байты выделены подчеркиванием.

П1.12. О понятии старшинства арифметических типов данных

В разных частях этой книги используется словосочетание "привести к старшему типу" применительно к арифметическим типам. В понятие старшинства вкладывается следующий смысл: если диапазон значений, которые может принимать переменная одного типа, целиком содержится в диапазоне значений, которые может принимать переменная второго типа, то второй тип считается старшим по отношению к первому. Иными словами, это означает, что любое значение переменной *младшего* типа может быть представлено с помощью переменной *старшего* типа.

Отметим некоторые особенности этого соотношения:

- любой знаковый тип не может быть младше любого беззнакового типа, т. к. в беззнаковом типе невозможно представить отрицательные значения;
- знаковые и беззнаковые типы одной длины не могут находиться в отношении "младший → старший";
- любой плавающий тип всегда старше любого целого типа. Однако значения целого типа могут представляться плавающим типом как совершенно точно, так и с небольшой ошибкой округления. Последнее происходит, если длина значащей части целого типа (без учета знакового бита) больше длины мантиссы плавающего типа (с учетом неявного бита).

Если старшинство определено таким образом, то все преобразования арифметических типов можно разбить на три группы:

- преобразование типа возможно выполнить без ошибки при любом значении переменной исходного типа;
- при выполнении преобразования типа результирующее значение с плавающей точкой формируется с (незначительной) ошибкой округления;
- преобразование типа может дать грубую ошибку для части значений переменной исходного типа, поэтому оно не всегда допустимо и требует от программиста дополнительной проверки на возможность преобразования.

На рис. П1.7 изображены отношения старшинства между различными арифметическими типами.

Буквой *u* обозначены беззнаковые целые типы, буквой *s* — целые типы со знаком. Число после буквы обозначает количество значащих двоичных цифр в числе (без учета знакового бита).

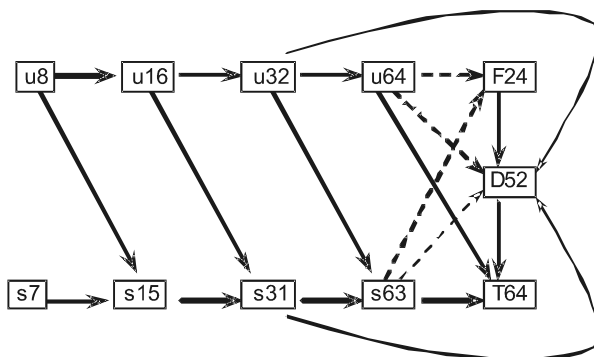


Рис. П1.7

Таким образом:

u8 — unsigned char,

u16 — unsigned short,

u32 — unsigned long,

u64 — unsigned long long,

s7 — signed char,

s15 — signed short,

s31 — signed long,

s63 — signed long long.

Буквы F, D, T обозначают соответственно плавающие типы float (4 байта), double (8 байтов) и внутреннее представление чисел ПТ Temporary Real (10 байтов). Числа в этих обозначениях показывают количество эффективных значащих битов в мантиссе.

Отношения "младший → старший" показаны на рисунке стрелками, причем сплошные стрелки отмечают преобразования типов, которые выполняются без ошибки (группа 1), а пунктирные стрелки соответствуют преобразованиям группы 2, при которых возможно возникновение ошибок округления.

П1.13. Битовые поля и операции над ними

Как уже упоминалось, арифметико-логическое устройство многих процессоров способно выполнять действия с операндами различной длины. В большинстве процессоров эти длины делают равными степени двойки: один, либо

два, либо четыре, либо восемь байтов. Однако в практике программирования нередки ситуации, когда длина информационного элемента не равна (не кратна) одному байту. Приведем несколько примеров.

Пример 1.

Рассмотрим представление элемента растрового изображения (пиксела). В одном из распространенных форматов кодирования пиксела (HighColor) для представления интенсивностей трех цветовых компонент: красной (Red), зеленой (Green) и синей (Blue) используются пятибитовые значения, представляющие эти интенсивности в диапазоне от 0 до $2^5 - 1 = 31$. Можно было бы для хранения каждой из компонент занять один байт, и для модификации отдельных цветовых компонент использовать команды процессора, оперирующие с байтом. Однако для экономии памяти, в которой хранится изображение, три цветовые компоненты *R*, *G*, *B* упаковываются в двухбайтовую переменную вот таким образом:

15	14	10	9 8	7 6 5	4	0	Номер бита
0	b b b b b	g g	g g g	r r r r r			Содержимое

При этом объем памяти для хранения изображения уменьшается в полтора раза по сравнению с вариантом "один цвет в одном байте".

Такой вид данных, когда отдельные информационные элементы имеют размер, не кратный одному байту и/или упакованы так, что границы этих элементов не совпадают в памяти с границами байтов, принято называть *битовыми полями* (Bit Fields). Для манипуляций с битовыми полями необходимо выполнять некоторые специфические операции, о которых поговорим чуть позже при рассмотрении примера модификации пиксела изображения HighColor.

Пример 2.

Рассмотрим команду процессора.

В различных процессорах двоичные последовательности, кодирующие команды, могут иметь различную длину и внутреннюю структуру, но во всех системах в процессорных командах может быть закодировано:

- ☐ операция, которую следует выполнить;
- ☐ длина операндов;
- ☐ месторасположение операндов (в памяти или в регистрах процессора);
- ☐ регистры;
- ☐ многие другие свойства и особенности команды.

Для кодирования каждого из этих свойств в коде команды используется ровно столько битов, сколько требуется для того, чтобы длина команды была минимальной. Например, в процессоре, содержащем шестнадцать регистров общего назначения, для указания регистра достаточно четырех битов.

Последовательность процессорных команд может быть результатом работы транслятора, т. е. теми выходными данными, которые генерирует транслятор. Для того чтобы сформировать процессорную команду, транслятору требуется определить значения отдельных полей команды, а затем скомпоновать эти поля в один (возможно, многобайтовый) элемент.

Пример 3.

Рассмотрим содержимое управляющего регистра периферийного устройства.

Для управления периферийным устройством в состав его регистров часто входит такой регистр, в котором отдельные комбинации битов позволяют программисту независимо задавать различные режимы работы устройства. Например, в состав любого персонального компьютера входит трехканальный интервальный программируемый таймер, его регистр управления имеет длину в один байт, в котором содержатся следующие битовые поля:

- бит 0 — режим счета (двоичный либо двоично-десятичный);
- биты 1...3 — режим работы (три бита используются для указания одного из шести режимов);
- биты 4...5 — выбор операции (два бита кодируют одну из четырех операций);
- биты 6...7 — выбор канала таймера (одного из трех).

Для управления работой таймера иногда может потребоваться изменить только некоторые поля регистра управления, оставив прочие биты неизменными. Конечно, можно при каждом управляющем воздействии записывать целиком весь байт, но в этом случае придется хранить в дополнительной переменной копию его содержимого.

Пример 4.

Теперь рассмотрим, какова может быть последовательность действий, позволяющая изменить некоторые части (битовые поля) в операнде, гарантированно оставив прочие биты операнда неизменными. Сделаем это на примере редактирования пиксела изображения HighColor, формат которого только что был рассмотрен.

Предположим, что требуется изменить значение одной из цветовых компонент (например, увеличить яркость зеленой составляющей пиксела G на за-

данную величину d), для этого нужно выполнить последовательность действий:

1. *Вырезать* из двухбайтового пиксела участок, соответствующий зеленому цвету (биты с 5 по 9):

0bbbbbgggggrrrr — исходный пиксел,

000000ggggg00000 — после вырезания.

2. *Переместить* вырезанное значение в разрядной сетке так, чтобы оно оказалось в младших позициях разрядной сетки, и его величина находилась бы в разрешенном диапазоне значений от 0 до 31:

00000000000ggggg — после сдвига.

3. Теперь можно выполнить модификацию значения цвета, т. е. сложение $(G + d)$, при этом сложении возможно переполнение и выход результата за пределы отведенной для цветовой компоненты пятибитовой разрядной сетки. Если это произойдет, то целесообразно в качестве результата взять максимальное разрешенное для цветовой компоненты значение, т. е. 31:

00000000000ggggg — после сложения.

4. *Переместить* модифицированное значение цветовой компоненты, чтобы она заняла отведенное ей место в двухбайтовой разрядной сетке:

000000ggggg00000 — после перемещения.

5. *Подготовить* (обнулить) место для компоненты в двухбайтовом пикселе.

6. *Вставить* модифицированное значение цвета в двухбайтовый пиксел так, чтобы остальные его биты остались неизменными:

0bbbbbgggggrrrr — после вставки получили модифицированное значение пиксела изображения, в котором цветовая компонента G изменена, а компоненты R и B оставлены неизменными.

Для того чтобы стало возможным выполнять действия, выделенные курсивом в пунктах 1, 2, 4, 5, в состав системы команд процессора вводят команды, которые позволяют выполнять следующие действия с операндом:

- а) очистка заданных программистом битов в операнде, что позволит в п. 1 очистить те биты, которые соответствуют цветовым компонентам R и B , а в п. 5 — биты, соответствующие модифицированной компоненте G ;
- б) установка заданных программистом битов операнда в 1;
- в) инвертирование указанных битов в операнде;
- г) сдвиг битов операнда в разрядной сетке на заданное количество позиций.

Для выполнения пп. 1 и 3 в составе системы команд процессора имеются двухоперандные команды, действие которых состоит в том, что каждый бит операнда-приемника (результата операции) формируется из двух соответствующих битов двух операндов-источников в соответствии с одной из трех логических функций:

- AND (и);
- OR (или);
- XOR (ИСКЛЮЧАЮЩЕЕ ИЛИ).

Для двух однобитовых аргументов возможны четыре комбинации: 00, 01, 10, 11.

Результат операции AND двух однобитовых аргументов равен единице только если оба аргумента равны единице (один и второй). В остальных трех случаях операция AND дает результат 0.

Побитовая операция AND предоставляет возможность очистить (установить в 0) любую комбинацию битов в операнде. Для этого программист должен сам сконструировать второй операнд (называемый обычно маской). Маска должна содержать 1 в тех позициях, которые в первом операнде должны остаться неизменными, и 0 — в тех позициях, которые должны быть очищены. В рассмотренном только что примере действие а) можно выполнить с помощью побитовой операции AND.

0bbbbbgggggrrrrr — исходный пиксел, выполняем побитовую операцию AND;

0000001111100000 — маска;

000000ggggg00000 — это результат побитовой операции AND.

Результат операции OR двух однобитовых аргументов равен:

- 0 — только в случае, когда оба аргумента равны 0;
- 1 — если хотя бы один из аргументов равен 1.

Побитовая операция OR позволяет установить произвольный набор битов в операнде в 1. Для этого надо использовать второй операнд (маску), содержащий 1 в тех позициях, которые хочется в первом операнде установить, и 0 — в позициях, которые в первом операнде должны остаться неизменными.

Действие 5 в рассмотренном примере с модификацией цветовой компоненты в пикселе можно выполнить в два шага: на первом очистить место под компоненту G с помощью побитового AND, а затем вставить модифицированную компоненту с помощью побитового OR:

0bbbbbgggggrrrrr — исходный пиксел.

AND

0111110000011111 — маска

 0b00000000000000000000000000000000 — это результат побитовой операции AND, освободили место для компоненты G;

OR

00000000000000000000000000000000 — модифицированный пиксел, полученный на шаге (4)

 0b00000000000000000000000000000000 — и вот он, искомый результат.

Результат операции XOR равен 1, если значения двух битов-аргументов различны. Несложно убедиться в том, что с помощью XOR можно проинвертировать те биты операнда, которым соответствуют единицы во втором аргументе-маске.

В наборе команд процессора обычно имеется также однооперандная команда NOT, действие которой состоит в инвертировании (замене на противоположное значение) всех битов операнда.

П1.13.1. Подробнее об операциях сдвига

Для перемещения положения битов в разрядной сетке, в состав системы команд процессора вводят *команды сдвига*. При сдвиге операнда крайние биты выдвигаются за границу разрядной сетки и одновременно с другого конца разрядной сетки освобождаются позиции. Существуют несколько разновидностей операций сдвига, которые различаются судьбой выдвигаемых битов и освобождающихся позиций.

Логический сдвиг — самая простая разновидность (выдвигаемые за границу разрядной сетки биты теряются безвозвратно, а позиции, освобождающиеся с другого конца разрядной сетки, заполняются нулями).

Действие сдвига целочисленного операнда можно использовать для быстрого умножения (при сдвиге влево) или для быстрого деления (при сдвиге вправо) целочисленного операнда на степень основания системы счисления, т. е. на 2^n , где n — количество позиций, на которое осуществляется сдвиг операнда. Смысл замены настоящего умножения сдвигом заключается в том, что в большинстве процессоров сдвиг операнда выполняется гораздо быстрее (за меньшее количество тактов), нежели целочисленное умножение.

При сдвиге влево (т. е. при умножении на степень двойки) может использоваться операция логического сдвига влево, независимо от того, является ли операнд знаковым (представленным в дополнительном коде) или беззнаковым.

При использовании сдвига для быстрого деления (сдвиг вправо) операции сдвига для знаковых и для беззнаковых целых различаются. При сдвиге вправо целого со знаком, представленного дополнительным кодом, отрица-

тельное число при делении на 2 формально должно остаться отрицательным, т. е. знаковый (старший) бит должен сохранить свое значение. Такая разновидность сдвига вправо, при которой знаковый бит сохраняет свое значение, носит название *арифметического сдвига*. В системах команд почти всех процессоров реализована такая команда.

ЗАМЕЧАНИЕ

Можно говорить об операции арифметического сдвига влево, при которой старший (знаковый) бит после сдвига также сохраняет свое значение. Однако этого не делают, поскольку в отрицательном числе, представленном в дополнительном коде, старший знаковый бит равен единице, и при этом еще один или несколько битов, смежных со старшим, также могут быть равны единице (для чисел, не очень больших по модулю). При логическом сдвиге влево на один разряд, модуль отрицательного числа удваивается. Если при очередном сдвиге в знаковый разряд попадает значение 0, то формально результат поменяет знак. Это всего лишь означает, что этот очередной сдвиг (удвоение числа) дал значение, слишком большое для представления в данной разрядной сетке, т. е. произошло переполнение.

Если же сдвигаемый операнд беззнаковый, то освобождающиеся при сдвиге слева позиции следует заполнять нулями, т. е. для деления сдвигом беззнакового операнда следует использовать *логический сдвиг* вправо.

В языке Си имеются операторы сдвига операнда вправо и влево. Компилятор, в зависимости от типа целочисленного операнда (со знаком или без знака), использует подходящую команду сдвига — арифметический либо логический сдвиг.

В системах команд процессоров существует еще одна разновидность сдвига — *циклический сдвиг*. При выполнении этого действия биты, выдвигаемые с одного конца разрядной сетки, заполняют позиции, освобождающиеся на другом конце разрядной сетки, т. е. при циклическом сдвиге никакой потери битов не происходит. При программировании на языке Си использование процессорных команд циклического сдвига обычно невозможно.



Приложение 2

Язык Си и низкоуровневое программирование

Что означает термин "низкоуровневое"? В данном и последующих контекстах этот термин почти всегда будет пониматься как "аппаратно-зависимое".

Классическая манера обучения программированию на языке Си (да и на других языках высокого уровня) предполагает, что написанные фрагменты кода будут одинаково работать на различных вычислительных системах (в том смысле, что одинаковыми будут полученные результаты). Именно этот факт имеют в виду, когда используют словосочетание *язык высокого уровня*. При программировании вычислительных алгоритмов и обработки символьных данных большей частью, так оно и получается.

Где встречается низкоуровневое программирование?

При разработке реальных (а не учебных) программ, в особенности для задач управления объектами, разработчику весьма трудно удержаться в рамках конструкций высокоуровневого языка и переносимого кода. Отдельные фрагменты разрабатываемой программы оказываются аппаратно-зависимыми, т. е. должны быть изменены при переносе такой программы на другую ВС.

Обычное решение этой проблемы состоит в том, что все аппаратно-зависимые компоненты разрабатываются отдельно и заранее, часто входят в состав используемого на данной ВС системного ПО (программного обеспечения).

Однако в ряде случаев то, что требуется в конкретной задаче, либо реализовано в системном ПО, слишком универсально и, как следствие, недостаточно эффективно либо вовсе отсутствует.

При разработке управляющих программ для микроконтроллеров (управляющих микроЭВМ) нередки случаи, когда ничего похожего на операционную систему не используется, а программист сам пишет весь код от начала и до конца. Тогда ему приходится самому разрабатывать и отлаживать

и все аппаратно-зависимые компоненты программы. Для этих целей обычно использовался язык Ассемблера.

Но в последние десять лет место Ассемблера все чаще стали занимать языки высокого уровня (и в частности, язык Си). Широкому распространению языка Си способствует тот факт, что разработаны качественные компиляторы Си для многих и многих типов управляющих микросистем.

Сейчас было рассказано о причинах, заставляющих программистов иногда использовать возможности аппаратного уровня для повышения эффективности разрабатываемой программы. В последующих разделах будут обсуждаться ситуации, в которых приходится использовать низкоуровневые возможности и соответствующие приемы. Для чтения дальнейшего текста необходимо понимать значение ряда терминов, относящихся к аппаратному уровню организации процессоров. Однако подавляющее большинство начинающих изучать программирование на языке Си имеют весьма смутное понятие об устройстве аппаратного уровня цифровых процессоров. Да и многие профессиональные программисты также представляют себе это устройство не в полной мере. Поэтому автор счел необходимым объяснить значения используемых терминов именно здесь, в основном тексте. Читателю настоятельно рекомендуется хотя бы бегло просмотреть эти объяснения вне зависимости от его уровня подготовки.

Глоссарий.

1. **Адрес ячейки в памяти** — индивидуальный признак ячейки памяти, целое число, которое используется при организации доступа к информационному элементу, расположенному в памяти. Если элемент занимает в памяти более одной ячейки, то его адресом считается наименьший из адресов, занимаемых этим информационным элементом.
2. **Адресное пространство** — одно из основных логических понятий архитектуры цифровых процессоров. Предполагается, что множество элементарных ячеек памяти (видимых) включено в адресное пространство, каждая ячейка имеет индивидуальный идентификатор, представляющий собой целое число и называемое адресом этой ячейки в данном адресном пространстве.
3. **Арифметико-логическое устройство (АЛУ)** — узел процессора, который выполняет с операндами действия, меняющие их значение: сложение, сдвиг и т. п. Обычно предполагается, что АЛУ не содержит в своем составе элементов памяти, т. е. является комбинационным логическим узлом.
4. **Биты признаков** — биты, содержащиеся в регистре состояния процессора (то же самое, что *флаги*). АЛУ при выполнении процессорной команды

может изменять значения отдельных битов признаков в зависимости от свойств получившегося результата. Стандартный набор включает четыре бита признаков:

- признак нулевого результата (Z , zf — Zero);
- признак отрицательного результата (N — Negative, sf — sign flag);
- признак переноса или беззнакового переполнения (C — carry, cf — carry flag);
- признак знакового переполнения (V или of — OVerflow).

В зависимости от значений битов признаков, команды с условным исполнением либо выполняют действие, либо ведут себя как пустая команда.

5. **Блок адресной арифметики** — узел процессора, выполняющий вычисление адреса операнда, алгоритм этого вычисления зависит от используемого в этой команде способа адресации.
6. **Дешифратор** — узел процессора, выполняющий дешифрацию (декодирование) кода процессорных команд и формирующий управляющие сигналы для АЛУ и для блока адресной арифметики.
7. **Косвенно-регистровая адресация** — способ адресации, при котором адрес операнда в памяти указывается содержимым одного из регистров процессора. Модифицируя содержимое этого регистра другими командами, можно добиться того, что одна и та же команда (повторяемая в цикле) будет выполнять действия с различными операндами в памяти.
8. **Кэш-память** — память малого объема с быстроедействием, равным или близким к быстроедействию процессора.
9. **Логический адрес** — целое число, определяющее положение информационного элемента в адресном пространстве. Логические адреса формируются в ходе выполнения процессорных команд. Для обращения к ячейкам реальной (физической) памяти логические адреса преобразуются в физические в ходе трансляции адресов.
10. **Машинное слово** — максимальный размер операнда, с которым способны выполнять действия большинство процессорных команд. Чаще всего этот размер совпадает с размером регистров процессора.
11. **Минимальная адресуемая единица (МАЕ)** — размер элементарной ячейки адресуемой памяти, имеющей индивидуальный адрес. Для большинства процессоров МАЕ равно 8 битам (одному байту).
12. **Многокомпонентная адресация** — способ адресации, при котором блок адресной арифметики (по окончании декодирования команды) вычисляет

адрес операнда в основной памяти, с помощью действий над несколькими компонентами. Компоненты могут быть расположены либо в регистрах (как при регистровой адресации), либо в коде команды (как при прямой адресации). Чаще всего адрес вычисляется как сумма этих компонент. Использование многокомпонентной адресации усложняет схемотехнику декодирования и адресной арифметики, но позволяет уменьшить количество команд, которые предназначены для модификации адресов операндов.

13. **Непосредственная адресация** — способ адресации, при котором операнд помещается прямо в код процессорной команды. Способ удобен, если значение операнда фиксировано и известно уже на этапе трансляции (например, количество повторений в операторе цикла).
14. **Основная память** или **оперативное запоминающее устройство (ОЗУ)** — блок, который наряду с процессором имеется в любом универсальном цифровом вычислителе с хранимой программой. Основная память имеет адресную организацию.
15. **Обработчик (подпрограмма обработки) прерывания** — подпрограмма, которая автоматически вызывается при возникновении сигнала запроса на входе запроса прерывания процессора. При входе в обработчик происходит автоматическое запоминание содержимого некоторых регистров процессора (обязательно счетчика команд, т. е. адреса возврата, и регистра состояний, который содержит биты признаков и некоторые управляющие биты). При возврате из обработчика в эти регистры автоматически восстанавливаются значения, бывшие там перед входом в обработчик.
16. **Прямая адресация** — способ адресации, при котором адрес операнда входит в состав кода команды. Способ удобен, если адрес известен уже на этапе трансляции, а в ходе выполнения программы его не требуется изменять (например, при обращении к регистрам периферийных устройств, каждый из них расположен по фиксированному и известному адресу).
17. **Регистр** — узел процессора, предназначенный для хранения двоичного слова. Регистр характеризуется количеством двоичных разрядов. Длина регистров в цифровом процессоре связана с другими характеристиками процессора: длиной машинного слова (АЛУ и регистры данных), размером адресного пространства (разрядность адресных регистров и схемы формирования физического адреса).
18. **Регистр состояния** — регистр, который содержит флаги, сигнализирующие о состоянии устройства. Регистр состояния процессора обычно содержит биты признаков, биты управления прерываниями процессора, а также (может быть) и другие управляющие биты. Регистр состояния

периферийного устройства может содержать биты или битовые поля, которые показывают состояние этого ПУ (например, готовность к обмену с другим устройством).

19. **Регистровая модель устройства** — перечень регистров, которые доступны программисту для анализа (при доступе по чтению) и/или для изменения (при доступе по записи). При низкоуровневом программировании необходимо иметь полное представление о регистровой модели устройства. Регистровая модель процессора минимально включает в свой состав счетчик команд, указатель стека, регистр состояния, регистры операндов (как минимум, один), адресные регистры (как минимум, один). Во многих (но не во всех) процессорах регистры операндов могут быть универсальными, т. е. их разрешается использовать и для адресов. В этом случае для них может быть использовано название "регистры общего назначения". Регистровая модель периферийного устройства в минимальном варианте содержит регистр данных и регистр состояния.
20. **Способ адресации** — схема (алгоритм), используемая в процессорной команде с обращением к памяти для определения адреса операнда в основной памяти. Различные процессоры могут использовать от двух/трех до нескольких десятков различных способов адресации.
21. **Стек** — участок памяти, к которому организуется стековый доступ. Это может быть сделано как чисто программно (используя косвенно-регистровую адресацию с автоматической модификацией адресного регистра), так и аппаратно, когда для формирования адресов используется специальный регистр — указатель стека, а для их автоматической модификации — специальная схемотехника, изменяющая содержимое указателя стека при каждом обращении.
22. **Стековый доступ** к памяти — способ организации записи или считывания последовательности информационных элементов, при котором обращение происходит подряд по порядку следования адресов. При этом обычно все информационные элементы имеют одинаковую длину (элемента стека). Если запись осуществляется по порядку убывания адресов, то считывание происходит в обратную сторону — по возрастанию адресов. При такой организации реализуется правило: последним записан — первым прочитан. Как правило, стековый доступ к памяти поддерживается в процессоре аппаратно. Для формирования адресов, по которым выполняются обращения, используется специальный регистр — указатель стека, причем после каждого обращения содержимое этого регистра автоматически (аппаратно) изменяется на длину элемента стека. Длина элемента стека в процессорах обычно равна длине машинного слова. По-

рядок обращения к стеку в разных процессорах может отличаться: а) направлением роста стека, т. е. направлением перебора адресов при записи/чтении; б) порядком действий "запись в стек" — автомодификация указателя стека. Наиболее распространен вариант, когда: а) стек растет в сторону уменьшения адресов, т. е. запись по убыванию адресов, а чтение — по возрастанию; б) сначала декремент указателя стека, а затем запись, т. е. указатель стека всегда содержит адрес последнего занятого элемента. При считывании порядок обязательно обратный: сначала считывание, а затем инкремент указателя стека.

23. **Счетчик (указатель) команд** — регистр процессора, который содержит (почти в любой момент времени) адрес памяти, с которого расположена следующая, еще не начавшая выполнение команда. После окончания выборки команды схемотехника процессора автоматически модифицирует содержимое указателя команд так, чтобы он указывал на следующую команду.
24. **Трансляция адресов** — процедура отображения логических (виртуальных) адресов, формируемых в ходе выполнения команд конкретной программы в физические адреса. Трансляция адреса выполняется блоком формирования адресов, входящим в состав процессора. В простых процессорах трансляция адреса может быть вырожденной в простое тождество (физический адрес совпадает с логическим).
25. **Указатель стека** — специализированный регистр процессора, используемый для организации стекового доступа к участку основной памяти, используемому в качестве стека.
26. **Управляющие биты** в регистре состояния — позволяют выполнять переключения режимов. Для процессора — в первую очередь разрешать либо запрещать реагировать на запросы аппаратных прерываний.
27. **Физический адрес** — номер ячейки памяти в физическом адресном пространстве.
28. **Физическое адресное пространство** — система нумерации ячеек в реальных модулях памяти, которые могут быть установлены в вычислительной системе. В конкретной реализации ВС объем реально установленной памяти не может превышать размера физического адресного пространства.
29. **Флаги** — отдельные биты в регистре состояния, сигнализирующие о состоянии устройства. В регистре состояния процессора флаги — это то же, что и биты признаков, они сигнализируют о свойствах результата одной из недавно выполненных процессором команд.

П2.1. Низкоуровневая (регистровая) модель вычислительного ядра

Цифровой универсальный процессор содержит в своем составе некоторое количество регистров, каждый из которых способен хранить двоичное слово. Разрядности этих регистров чаще всего одинаковы, именно их имеют в виду, когда говорят о разрядности процессора. Когда употребляют термин "регистровая модель процессора", то имеют в виду все те регистры, которые программно доступны при использовании процессорных команд (т. е. для программиста на Ассемблере).

Минимальный состав регистровой модели включает в себя:

- ☐ счетчик команд;
- ☐ регистр состояний;
- ☐ указатель стека;
- ☐ регистр адресный (указатель);
- ☐ регистр данных (аккумулятор).

В более сложных процессорах регистров адресных и регистров данных может быть более одного. Так, например, в процессорах семейства Intel x86 регистровая модель содержит счетчик команд, регистр состояний, указатель стека и еще семь регистров общего назначения РОН (в каждом из РОН может храниться операнд или адрес элемента в памяти — указатель). Все перечисленные регистры имеют длину 4 байта (32 бита).

В любом процессоре, поддерживающем механизм аппаратных прерываний, программисту необходимо разрешать/запрещать процессору реагировать на запрос аппаратного прерывания. Обычно для этого служит один или несколько битов (флагов, масок) в регистре состояния процессора. Помещение масок прерываний в регистр состояния имеет глубокий смысл: при входе в подпрограмму обработки прерывания в большинстве процессоров автоматически сохраняется не только адрес возврата (содержимое счетчика команд), но и содержимое регистра состояния (включающее биты признаков и значения масок прерываний). Последние отражают состояние процессора (какие прерывания были разрешены процессору перед входом в прерывание).

После завершения подпрограммы обработки прерывания сохраненные значения автоматически возвращаются в регистр состояния, восстанавливая тот режим, который был перед входом в обработчик.

В более сложных процессорах возможно (и нередко необходимо) выполнять еще ряд управляющих (настроечных) операций. Если таких операций немного,

то управление ими может быть организовано через дополнительные битовые поля в регистре состояний. При большом количестве настроек управление может осуществляться через один или несколько системных управляющих регистров, запись в которые определенных кодовых комбинаций позволяет выполнять требуемые настройки режимов процессора. Например, в процессорах семейства x86 имеется более десятка системных регистров, для доступа к которым есть специальные команды.

Регистр состояния в некоторых архитектурах полностью доступен для чтения и записи, как любой другой регистр, но в других процессорах доступ к регистру состояния может быть в той или иной степени ограничен. Например, в системе команд могут быть специальные команды, позволяющие изменять содержимое некоторых полей (битов) в регистре состояния, в то время как другие его поля будут недоступны для непосредственной записи/чтения.

В процессорах x86 счетчик команд является программно недоступным в том смысле, что его нельзя использовать в ассемблерной команде в качестве операнда. Однако любая команда перехода (нарушающая естественный порядок следования команд по адресам подряд) фактически выполняет засылку в счетчик команд нового содержимого — адреса перехода. Чтение содержимого счетчика команд в процессорах x86 напрямую невозможно, но может быть выполнено последовательностью из нескольких команд.

П2.1.1. Оптимизация фрагмента кода по скорости

Обращение к регистрам в персональном компьютере обычно занимает на порядок меньше времени, нежели обращение к элементам в адресуемой памяти. Один из путей оптимизации программы может состоять в том, что часто (интенсивно) используемые переменные лучше располагать в регистрах.

ЗАМЕЧАНИЕ

В однокристальных микроконтроллерах для управляющих (встраиваемых) применений, внутрикристалльная память с адресной организацией выполняется по той же технологии, что и процессор, и имеет то же быстродействие. Поэтому перенос переменных в регистры может и не дать выигрыша.

Если программируем на языке высокого уровня, то использование тех или иных регистров для различных целей находится в ведении компилятора (и он может это делать с большей либо с меньшей эффективностью).

Иногда программисту хочется принудительно поместить переменную в регистр.

В стандарте языка Си имеется ключевое слово **register**, применив которое в объявлении переменной (обычно локальной), программист может предла-

гать компилятору использовать один из регистров для хранения этой переменной.

В некоторых компиляторах возможность указания использовать определенный регистр отсутствует (например, среда MS Visual Studio). В этом случае компилятор сам решает, какой из регистров подойдет для переменной. При этом у компилятора может и не оказаться такой возможности (нет свободных регистров).

В некоторых средах разработки компилятор понимает предопределенные имена т. н. псевдопеременных, соответствующих регистрам процессора. Например, в среде Borland C 3.1 регистры 16-разрядной архитектуры 8086 имеют имена: `_AX`, `_BX`, `_CX`, `_DX`, `_SI`, `_DI`, `_BP`, `_SP`.

В строке Си-программы разрешается запись вида:

```
var=_AX;
```

или

```
_DX= var;
```

Такая запись позволяет выполнять операции с регистрами без формального использования ассемблерных вставок.

Среда разработки Borland C 3.1 и ее средства трансляции по сей день используются для программирования процессоров с 16-разрядной архитектурой 8086 в управляющих встраиваемых системах.

Обратите внимание на то, что если компилятор связал имя переменной и регистр, то он будет избегать использования этого регистра при компиляции соседних операторов. Поэтому целесообразно обращаться к регистрам только для локальных переменных с коротким временем жизни и с ограниченной областью видимости. Вне областей видимости и существования регистр свободен для использования компилятором.

В других реализациях программист может сам выбрать для хранения локальной переменной один из регистров и указать его компилятору. Такую возможность предоставляет компилятор GCC фирмы RedHat.

Бывают ситуации, в которых, наоборот, необходима гарантия того, что компилятор не будет использовать регистр для хранения значения какого-либо программного объекта. Это необходимо делать, в частности, при чтении содержимого регистров периферийных устройств, поскольку содержимое этих регистров может быть изменено аппаратно в любой момент.

В таком случае следует использовать ключевое слово **volatile** при объявлении переменной — в результате компилятор будет при каждом обращении к этому элементу выполнять чтение непосредственно самого элемента, указанного в исходном тексте программы.

П2.1.2. Определение положения программы в пространстве адресов

Зачем может понадобиться доступ к счетчику команд по чтению?

Иногда возникает потребность выяснить из программы, в каких адресах она расположена. Это легко сделать, если счетчик команд доступен по чтению, тогда можно просто скопировать его содержимое в переменную.

В некоторых архитектурах (например, в x86) счетчик команд программно недоступен. Но его содержимое, тем не менее, можно получить, выполнив команду вызова подпрограммы, а затем — прочитав адрес возврата из того места, где он был сохранен (обычно это стек). Доступ к элементам стека можно организовать через указатель. Для инициализации этого указателя надо прочитать содержимое регистра — указателя стека, а вот для этого уже придется использовать ассемблерные команды.

Чтобы контролировать состояние стека, можно периодически (или в тех местах программы, где возникают опасения) сравнивать содержимое указателя стека с известными граничными значениями. Для этого необходимо иметь возможность прочитать содержимое указателя стека в переменную, а это можно сделать, только используя ассемблерные команды.

Чтобы можно было выполнять действия с содержимым управляющих (системных) регистров, надо иметь возможность инициировать выполнение процессорных команд чтения/записи этих регистров. А это обычно можно сделать, только используя средства уровня процессорных команд (Ассемблера).

П2.1.3. Использование средств уровня языка Ассемблера в программах на Си

Почти все современные среды программирования (компиляторы) позволяют делать ассемблерные вставки в исходный код программы на Си. На практике это чаще всего означает, что компилятор выполняет трансляцию с языка Си в язык Ассемблера, а затем вызывает для дальнейшей трансляции Ассемблер (так, например, делает наиболее популярный транслятор для программ реального режима процессоров x86 из среды Borland C3.1).

В среде Visual Studio ассемблерная вставка выглядит следующим образом:

```
__asm команда_Ассемблера;
```

или

```
__asm { команда_Ассемблера  
команда_Ассемблера
```

```
... ..
    команда_Ассемблера
}
```

В качестве примера использования ассемблерных команд при программировании для процессоров семейства x86 Pentium+, рассмотрим возможность измерения малых интервалов посредством обращения к счетчику меток времени `TimeStampCounter`. Этот счетчик длиной в 64 бита имеется во всех процессорах Pentium и считает тактовую частоту процессора. Для обращения к этому счетчику в системе команд имеется команда **RD TSC** (**ReaD Time Stamp Counter**). Эта команда копирует содержимое `TimeStampCounter` в пару регистров `EDX:EAX`. После выполнения этой команды надо организовать формирование либо 64-разрядного беззнакового целого (тип **long long int**), либо плавающего значения (тип **double** или **float**). Это можно сделать следующим фрагментом:

```
long long int llTime1, llTime2, llDeltaT;
... ..
__asm rdtsc;           // Фрагмент, захватывающий время
__asm mov dword ptr [llTime1],eax;
__asm mov dword ptr [llTime1+4],edx;
```

Ни одно из трех действий, которые выполняют приведенные ассемблерные команды, невозможно запрограммировать одними только средствами языка Си. В результате трех первых команд значение счетчика меток оказывается скопированным в переменную `llTime`.

Приведенный фрагмент следует выполнить непосредственно перед началом интервала времени, подлежащего измерению. Далее можно аналогично захватить отсчет времени в конце измеряемого интервала.

```
//здесь блок кода, время выполнения которого
    требуется измерить
__asm rdtsc;
__asm mov dword ptr [llTime2],eax;
__asm mov dword ptr [llTime2+4],edx;
llDeltaT = llTime2-llTime1; //вычислим разность
```

Разность двух захваченных значений даст длительность измеряемого интервала в периодах тактового генератора процессора на данном компьютере. При таком измерении возникает систематическая положительная ошибка, равная времени выполнения трех команд захвата времени.

Если фрагмент кода, содержащий ассемблерные команды, предполагается использовать неоднократно, его можно инкапсулировать в макрос. Далее

приводимый фрагмент кода использует макросы и вычисляет время в формате с плавающей точкой. Этот формат удобен для последующего масштабирования результатов измерения времени и представления его в наиболее удобных единицах измерения.

... ..

```
unsigned int iEAX1,iEDX1,iEAX2,iEDX2;
double dT1,dT2,      //для разности тиков
        dT0,dTd,      //для мертвого времени
        dKT;          //масштаб пересчета тактов в микросекунды
```

```
#define GetTSC1    __asm rdtsc\
                  __asm mov dword ptr [iEAX1],eax\
                  __asm mov dword ptr [iEDX1],edx
```

```
#define GetTSC2    __asm rdtsc\
                  __asm mov dword ptr [iEAX2],eax\
                  __asm mov dword ptr [iEDX2],edx
```

```
#define DeltaT     dT1=(iEDX2-iEDX1)*4294967296.0+iEAX2-iEAX1-dTd
```

... ..

//установка мертвого времени

```
GetTSC1;
```

```
GetTSC2;
```

```
DeltaT;
```

```
dT0=dT1; //однократное измерение и получение начального значения
n=11;
```

```
while ((n-->1) {      //повторение измерений
```

```
    GetTSC1;
```

```
    GetTSC2;
```

```
    DeltaT;          //... и нахождение минимума
```

```
    if (dT0>dT1)
```

```
        dT0=dT1; //по нескольким измерениям
```

```
}
```

```
dTd=dT0;
```

```
wprintf(L"      Мертвое время установлено %5.0lf тактов\n", dT0);
```

//Измерение тактовой частоты процессора

```

...
GetTSC1;           //захват времени
Sleep(1000);       //задержка на 1000 мс
GetTSC2;           //захват второго времени
DeltaT;            //вычисление разности времен
dKT=100000/dT1;    //множитель для перевода системных тактов в
                  микросекунды
wprintf(L" Тактовая частота данного процессора %10.6lf
        МегаГерц\n\n",dT1/1000000); //напечатаем разность

```

СУЩЕСТВЕННОЕ ЗАМЕЧАНИЕ!

В некоторых универсальных процессорах реализован программно-аппаратный механизм энергосбережения, который анализирует текущую загрузку процессора и при уменьшении загрузки динамически понижает тактовую частоту ядра (ту самую, что поступает на вход счетчика меток времени — TimeStampCounter). Так, например, сделано в ряде мобильных (предназначенных для ноутбуков) процессоров фирмы AMD. В них скорость счета TimeStampCounter оказывается непостоянной, и использовать TimeStampCounter для точных измерений времени может быть затруднительно.

Для определения состояния стека можно использовать такую ассемблерную вставку:

```

unsigned int uiSP;

... ..
__asm mov dword ptr [uiSP],esp;
... ..//содержимое ESP в переменной uiSP, можно проверять

```

П2.1.4. Работа с регистрами периферийных устройств

Каждое периферийное устройство (ПУ) имеет в своем составе один или несколько регистров, которые можно прочитать или записать командами программы. Сложные периферийные устройства могут содержать несколько десятков регистров. Таким образом, общее количество регистров ПУ в системе может быть значительным.

В некоторых процессорах доступ к регистрам ПУ осуществляется аналогично доступу к ячейкам памяти. Каждому регистру присвоен адрес в адресном пространстве памяти. В этом случае для обращения к регистрам ПУ можно использовать те же команды, что и для доступа к ячейкам памяти. Такая организация носит название — ввод/вывод, отображаемый на память (Memory-Mapped Input/Output). Обычно разработчики вычислительной системы выде-

ляют для адресации регистров ПУ какой-либо фиксированный диапазон адресов, а затем выделяют конкретные адреса каждому регистру ПУ.

В других процессорах регистры ПУ могут иметь свою систему адресации (адресное пространство), никак не связанную с адресацией ячеек памяти. Для обращения к регистрам ПУ в системе команд имеются специальные команды ввода/вывода. Такая организация обмена с ПУ носит название — изолированный ввод/вывод.

Для обозначения программно-доступных регистров периферийных устройств в компьютерной литературе употребляют термин "порты ввода/вывода".

В процессорах семейства x86 реализован изолированный ввод/вывод. В составе системы команд имеются команды, позволяющие программисту указать адрес порта в адресном пространстве ввода/вывода и прочитать из порта (или записать в него) двоичное слово. Однако разработчики периферийных устройств при их проектировании используют возможность сделать видимыми отдельные объекты ПУ через адресное пространство памяти.

Как можно из Си-программы получить доступ к портам ввода-вывода?

Если в ВС использован (как основной) ввод/вывод, отображаемый на память, то можно объявить указатель, проинициализировать его значением соответствующего адреса (в последующем примере этот адрес равен 0xFFFF0000, после чего можно обращаться к регистру ПУ через этот указатель:

```
unsigned int *pPort = (unsigned int *)0xFFFF0000;
```

Однако здесь имеются два маленьких "но".

Первое "но" состоит в том, что регистр порта может модифицироваться аппаратно в результате внешних по отношению к процессору событий (таких, как нажатие на клавишу), и об этом следует сообщить компилятору, используя ключевое слово **volatile**:

```
volatile unsigned int * pPort = (unsigned int *)0xFFFF0000;
```

В этом случае компилятор при каждом обращении к переменной *pPort будет обращаться по указанному адресу (и никогда не будет использовать для хранения считанного значения регистры процессора).

Второе "но" — объявленный указатель хорошо бы защитить от случайной модификации ключевым словом **const**:

```
volatile unsigned int * const pPort1 =  
(volatile unsigned int * const) 0xFFFF0000;
```

```
.....
```

```
var=*pPort1;    //теперь можно читать порт
```

Можно обойтись и без объявления указателя (экономим память), просто написав следующее:

```
var=((volatile int*)0xFFFF0000); //читаем порт по его адресу
```

Но запись справа от знака равенства весьма длинна. Заменим ее коротким обозначением с помощью макроопределения:

```
#define PORT1 *((volatile int*)0xFFFF0000);
```

Теперь можно будет писать гораздо короче:

```
var=PORT1; //для чтения из порта в переменную ... и все !
```

```
PORT1=0x123; //запись в порт
```

Что делать, если обращение к регистрам ПУ организовано через изолированный ввод/вывод (как в системах на базе процессоров Intel x86)? Придется использовать команды доступа к адресному пространству ввода/вывода **in** и **out**.

Си-компилятор, увы, этих команд (и также некоторых других) не знает. Выход — в использовании встроенного (inline) Ассемблера. Для обращения к портам в отдельном адресном пространстве ввода/вывода можно написать и использовать в тексте Си-программы несколько макросов, которые позволяют выполнять действия с содержимым регистров портов. Примеры использования встроенного Ассемблера и макросов уже были рассмотрены в *разд. П2.1.3*.

Модель периферийного устройства для программиста — это набор регистров в периферийном устройстве, которые программист может прочитывать/записывать программно.

Простейшая модель ПУ для программиста может содержать весьма мало регистров:

- ❑ регистр состояния (статуса) — минимально содержит один бит — флаг состояния ПУ;
- ❑ регистр (буфер) данных — запись/чтение этого регистра обеспечивает обмен данными с ПУ.

П2.1.5. Синхронизация программы с внешним событием

Ситуации, когда требуется синхронизация выполнения программы с внешним событием, возникают, прежде всего, при обмене данными с периферийными устройствами, в частности — с устройствами ввода/вывода. Например:

- ❑ прежде чем записать байт в регистр данных передатчика СОМ-порта, программа должна убедиться, что процесс передачи предыдущего байта завершился, и регистр данных передатчика свободен;

- при выводе данных на принтер программа печати должна убедиться, что предыдущая порция данных напечатана (печать происходит довольно долго), и принтер способен принять новую порцию.

Возможны обратные ситуации, когда, наоборот, при наступлении события нельзя продолжать выполнение текущей программы. Например:

- произошло деление на нуль, и следующую операцию, которая должна использовать результат деления, выполнить невозможно;
- пользователь нажал <Ctrl>+<Break>, и выполнять программу дальше не следует.

Общим в приведенных ситуациях является то, что в определенные моменты программа должна прореагировать на некоторое событие (реакция — передача управления по условию наступления внешнего события).

Рассмотрим, каким образом программа может узнать о наступлении события. Простейший способ — программный опрос готовности (polling) представлен на рис. П2.1.

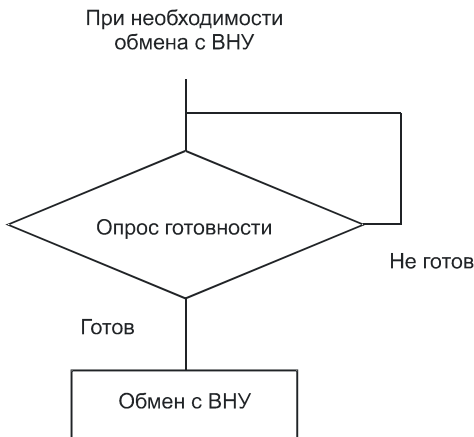


Рис. П2.1

Опрос готовности состоит чаще всего в анализе состояния определенных битов в регистре статуса (состояния). При достижении состояния готовности к обмену ПУ устанавливает этот(эти) бит(ы) в определенное состояние. Обмен с ПУ состоит в чтении/записи в регистр данных.

Так, если передатчик последовательного порта COM1 передал предыдущий байт, то автоматически устанавливается флаг готовности: бит 04 в порте LineStatus (0x3FD). Фрагмент Си-программы (для компилятора Borland C 3.1),

передающий байт в регистр данных передатчика DataPort, может выглядеть следующим образом:

```
while (inportb(LineStatus)&0x20==0) {}; //ждем готовности передатчика
outportb(DataPort, OutByte++);          //передаем следующий байт
```

Здесь:

inportb() и outportb() — стандартные библиотечные функции обращения к портам ввода/вывода.

Несмотря на простоту приведенного фрагмента, он имеет существенный недостаток — во время опроса состояния передатчика процессор не может выполнять другой (полезной) работы. А ведь опрос (по процессорным меркам) продолжается очень долго: даже при скорости передачи 115 200 кБит/с за время передачи байта процессор Pentium 4 успеет выполнить около 500 тыс. команд.

Поэтому в случаях, подобных описанному, для синхронизации программы с внешним событием (готовность ПУ) используют *механизм аппаратных прерываний*.

П2.2. Программирование обработчиков прерываний

Термин *прерывание* в русскоязычной компьютерной литературе многозначен и употребляется для обозначения трех различных вещей:

- hardware interrupt — аппаратное прерывание;
- exception — исключение, т. е. прерывание по исключительной (экстраординарной) внутренней ситуации;
- software interrupt — программное прерывание.

Термин *обработка прерывания* может использоваться для обозначений двух различных вещей:

- действия, которое автоматически выполняет процессор при возникновении запроса (они реализованы аппаратно);
- действия, которое выполняет подпрограмма-обработчик прерывания (handler).

Мы будем использовать термин *обработка прерывания* только во втором смысле, а для первой группы действий будем употреблять термин *вход в прерывание*.

ВАЖНОЕ ЗАМЕЧАНИЕ

Аппаратное прерывание аналогично вызову функции (подпрограммы), только этот вызов происходит не по команде вызова подпрограммы, а как следствие

внешнего (по отношению к процессору) события, которое вызывает приход в процессор внешнего электрического сигнала (запроса прерывания). Существенная особенность рассматриваемой ситуации состоит в том, что момент наступления события может быть никак не связан с текущим состоянием программы.

Будем далее называть функцию (подпрограмму), реагирующую на такое событие, *обработчиком прерывания* (interrupt handler или exception handler).

При программировании и/или использовании обработчиков прерываний программисту придется учитывать ряд обстоятельств, которые сейчас и рассмотрим.

П2.2.1. Запрет/разрешение прерываний процессору

В программе могут существовать участки кода, которые должны выполняться неразрывно (атомарные операции). Для обеспечения такой неразрывности, любой процессор имеет средства, позволяющие программисту запрещать/разрешать процессору реакцию на сигналы запросов прерываний. Обычно для этого служит один из битов регистра состояния процессора, который одним из двух своих значений разрешает, а другим запрещает реакцию процессора на сигнал запроса прерывания. Этот бит называют *флагом* или *маской* прерывания. Термин *флаг* используют, если единичное состояние этого бита разрешает прерывание, а термин *маска* используется, если единичное состояние бита запрещает прерывание.

Чтобы разрешить/запретить прерывание, программист должен иметь возможность изменять флаг прерывания. Для этого используется либо прямая модификация регистра состояний (если архитектура процессора это позволяет), либо пара специальных процессорных команд (разрешить прерывание и запретить прерывание). Собственно в языке Си средства для разрешения и запрета прерываний отсутствуют. Однако в стандартную библиотеку функций такие средства могут быть включены (а может быть, и нет).

Код таких функций (или макросов) может быть написан на Ассемблере. Например, в системе команд процессоров x86 имеются команды:

□ **sti** — для разрешения прерывания;

□ **cli** — для запрета прерывания.

Эти две команды изменяют состояние бита разрешения прерываний в регистре состояний. Для разрешения/запрета прерываний можно определить два макроса:

```
#define DIS_INT __asm cli;
```

```
#define EN_INT __asm sti;
```

При использовании таких средств следует иметь в виду, что в тот момент, когда выполняется макрос DIS_INT, прерывание уже может быть запрещено.

Поэтому по окончании критической секции кода следует не просто разрешить прерывание, но восстановить то состояние процессора, которое было на момент выполнения DIS_INT. Для этого при запрещении прерывания необходимо запомнить текущее состояние флага прерывания в регистре состояний:

```

...
int iFl;           //локальная переменная в функции или блоке

#define ENTER_CRITICAL __asm {push  ax\
                             pushf\
                             pop   ax\
                             and   ax, 0x0200\
                             mov   word ptr[iFl], ax\
                             pop   ax\
                             cli\
};

//для восстановления состояния процессора можно определить еще
    один макрос:
#define EXIT_CRITICAL  __asm {push  ax\
                             pushf\
                             pop   ax\
                             or    ax, word ptr[iFl]\
                             push  ax\
                             popf\
                             pop   ax\
};

```

П2.2.2. Приоритеты и управление ими

В более-менее сложной ВС одновременно могут быть активны несколько источников запросов аппаратных прерываний (т. е. может одновременно происходить взаимодействие с несколькими периферийными устройствами).

Если прерывание было запрещено в течение некоторого времени, а затем выполнялась команда *разрешить прерывание*, то к этому времени возможно наличие нескольких одновременно действующих запросов. В такой ситуации должно существовать правило, в соответствии с которым выбирается один из запросов для первоочередной обработки (приоритеты).

Разработчики вычислительной системы принимают решение об относительной важности (приоритетности) различных источников запросов. В некоторых ВС у программиста имеется возможность изменять приоритеты запросов.

Запрет вложенных прерываний и возможность их разрешения

Правило выбора запроса в соответствии с его приоритетом действует лишь в процессе входа в прерывание. Поэтому для исключения вложенного прерывания при входе в прерывание в процессоре аппаратно выполняется команда *запретить прерывание*, исключая тем самым реакцию на другие запросы и возникновение вложенного прерывания. Однако программисту не запрещено в обработчике прерывания выполнить команду *разрешить прерывание*. После этого вложенное прерывание оказывается возможным.

Маскирование запросов от отдельных источников

В вычислительной системе очень часто имеется возможность программно запретить (замаскировать) или разрешить прохождение запросов от отдельных ПУ. Это дает возможность программисту активировать только нужные ему запросы. Кроме того, маскируя на время отдельные запросы, можно управлять порядком их обработки (т. е. приоритетами).

Маскирование запросов обычно осуществляется изменением бита маски в одном из регистров периферийного устройства. Чаще всего, единичное значение этого бита запрещает (маскирует) запрос прерывания.

Понятие контекста программы и необходимость его сохранения/восстановления при обработке прерываний

Совокупность содержимого регистров процессора характеризует текущее состояние программы. Следуя принятой терминологии, будем эту совокупность называть *контекстом* программы.

Поскольку код обработчика прерывания (возможно) использует регистры процессора, в них к моменту входа в обработчик могут содержаться промежуточные результаты, сформированные прерванной программой. Поэтому обработчик прерывания первым делом должен сохранить (в стеке) контекст прерванной программы. В большинстве процессоров это сохранение следует выполнять программно. В некоторых процессорах операция сохранения контекста при входе в обработчик прерывания выполняется аппаратно (например, Motorola 6811, 6812).

По окончании кода обработчика перед возвратом в прерванную программу контекст следует восстановить (скопировать в регистры значения, ранее сохраненные в стеке).

Сохранение и восстановление контекста в стеке может быть запрограммировано только с использованием средств языка Ассемблера. Обычно эта функция

возлагается на компилятор, но он должен быть осведомлен о том, что данная подпрограмма (функция) — это обработчик прерывания. Для этого в некоторых диалектах языка Си имеется ключевое слово **interrupt**, которое сообщает компилятору, что данная функция является обработчиком прерывания, при входе в нее следует сохранить полный контекст, а перед выходом восстановить его. Обычно при входе в прерывание в стеке аппаратно сохраняется не только адрес возврата, но и содержимое регистра состояния процессора. Кроме того, компилятор организует выход из обработчика прерывания специальной командой (возврат из прерывания), по этой команде из стека дополнительно будет восстановлено содержимое регистра состояния процессора.

При организации многозадачного режима работы с вытесняющей многозадачностью Диспетчер Задач получает управление по запросу аппаратного прерывания, выполняет операцию планирования (принимает решение, следует ли выполнить переключение задачи) и при положительном решении должен выполнить процедуру переключения контекста. Обычно эта часть кода многозадачной операционной системы оказывается платформно-зависимой и пишется на языке Ассемблера.

П2.3. Программирование без операционной системы

Перед тем как начнет работать код, сгенерированный компилятором из вашей Си-программы, необходимо произвести в вычислительной системе ряд начальных настроек. Объем этой работы зависит от сложности аппаратной части и может варьироваться в весьма широких пределах.

Однако некоторые действия для конкретной ВС могут быть обязательными. Вот некоторые из них:

- задание местоположения стека (или стеков, если в системе их требуется несколько). Пример системы, в которой с самого начала может потребоваться несколько стеков — это ВС на базе процессоров с архитектурой ARM. Процессоры ARM требуют индивидуального стека для каждого из шести режимов, в которых может находиться процессор;
- настройка режимов работы процессора. Для этого может понадобиться доступ к системным регистрам. Если системные регистры видны через адресное пространство памяти (как регистры периферийных устройств), то это можно сделать средствами Си. Но иногда для доступа к системным регистрам в процессоре используются специальные команды, запрограммировать которые можно только средствами языка Ассемблера;

- ❑ настройка подсистемы прерывания:
 - переключение режимов работы контроллера прерываний;
 - занесение векторов используемых прерываний;
 - запрет/разрешение отдельных запросов;
 - разрешение прерывания процессору;
- ❑ конфигурирование памяти. В некоторых процессорах имеется возможность программно управлять диапазонами адресов, через которые программист видит различные программно-доступные элементы;
- ❑ настройка подсистемы тактирования и управления потребляемой процессором мощностью. Это позволяет обменивать производительность процессора на потребляемую им мощность. Перед началом работы программы могут потребоваться вполне определенные настройки подсистемы тактирования;
- ❑ инициализация глобальных переменных и, может быть, некоторых других областей памяти;
- ❑ инициализация отдельных периферийных устройств, необходимых для работы ВС, в частности — системного таймера.

Перечисленные действия (а может быть и еще какие-то — это зависит от процессора) должны быть выполнены до входа в функцию `main`, т. е. до того, как начнет выполняться код, соответствующий операторам Си-программы. Часть настроечных действий невозможно выполнить средствами Си. Поэтому пользуются приемом, позволяющим обеспечить выполнение настроечных действий — это включение в проект так называемого `start-up`-файла, который обычно написан на языке Ассемблера. Этот файл содержит стандартный набор настроечных действий, которые всегда должны быть выполнены до начала основной программы, написанной на Си. Компоновка производится таким образом, что этот файл получает управление первым, выполняет настроечные действия, а затем вызывает функцию `main`.

Для выполнения *низкоуровневой оптимизации* программист может при отладке произвести действия по профилированию программы — определить, сколько времени выполняются отдельные ее части и выявить критичные ко времени фрагменты кода. Затем можно проанализировать их и, если есть такая возможность, переписать эти фрагменты, используя либо средства встроенного Ассемблера, либо выделив оптимизированные фрагменты в отдельный файл на языке Ассемблера. В некоторых случаях это позволит увеличить скорость работы программы весьма существенно (на десятки процентов, а то и в разы).

Предметный указатель

C, L

const 109
Lvalue 30

N

namespace 97
 директива using 101
 неименованные 106
 стандартной библиотеки
 (std) 106

S, V

sizeof 69
static 91
volatile 110

Б

Блок кода 24

Д

Декорирование имен 88
Динамическое выделение памяти
 Динамические массивы 218, 225
 Инициализация динамических
 массивов 234
 Операторы C++ new и delete 222

Различие операторов delete
 и delete[] 233
Сборщик мусора 225
Функции языка Си 219

З

Заголовочные файлы 152
 Защита от повторных
 включений 166
Предкомпиляция 159
Стандартизованные 164
Стандартной библиотеки 162

И

Идентификаторы 29
Инструкции 113
 break 126, 133, 134
 continue 126, 132, 133, 134
 do...while 127
 for 129
 goto 134
 if, if...else 115
 return 134
 switch 118
 while 123

К

Ключевые слова 29
Комментарии 26

Л

- Литералы 51
 - есаре-последовательности 60
- С плавающей точкой 54
- Символьные 55
- Строковые 61
- Целые 52

М

- Массивы 196
 - Неявная инициализация 200
- Объявление 196
- Оператор [] 198
- Оператор sizeof 206
- Указателей 216
- Явная инициализация 200
- Модульность 13

О

- Объединение 345
 - Анонимное 351
- Инициализация 350
- Оператор sizeof 349
- Сравнение
 - со структурами 346
- Объявление 82
- Операторы 31, 47
 - Ассоциативность 32
 - Декремент 36
 - Инкремент 36
 - Логические 43
 - Отношения 43
 - Побитовые 39
 - Приоритет 31
 - Сдвига 40
 - совмещенные операторы
 - присваивания 39
 - Тернарный 45
- Определение 83

П

- Переменные 65
 - Время жизни 90

- Замещение области видимости 95
- Знаковость переменной 71
- Инициализация 108
- Область видимости 94
- Оператор разрешения области видимости 96
- Способы использования 86
- Способы размещения 89
- Тип bool 81
- Тип wchar_t 81
- Тип переменной 66
- Типы компоновки 87

- Перечисление (enum) 62
- Порядок синтаксического разбора выражения 37
- Препроцессор 137
 - assert 144
 - Директива #define 138
 - Директива #error 171
 - Директива #include 155
 - Директива #pragma 170
 - Директива #undef 145
 - Директивы #if, #elif 149
 - Директивы #ifdef, #ifndef, #else, #endif 147
 - Оператор defined 149
 - Предопределенные макросы 143
- Приведение типов 73
 - Неявное 74
 - Явное 78
- Псевдонимы типов (typedef) 85

Р

- Раздельная компиляция 15

С

- Связь массивов и указателей 207
 - Двухмерные массивы 210
 - Многомерные массивы 214
 - Одномерные массивы 207
- Ссылки 235
 - Инициализация 236
 - Константные 239
 - На указатель 238

Объявление 236
Сравнение с указателями 236
Структуры 313
 typedef 318
 Анонимные 319
 В качестве возвращаемого
 функцией значения 332
 В качестве параметра функции 329
 Вложенные 323
 Доступ к полям 317
 Инициализация 320
 Объявление 314
 Оператор sizeof 326
 Поля битов 334
 Поля пользовательского типа 322
 Селектор . 317
 Селектор -> 324
 Создание экземпляров 316
 Тип поля 333
 Указатели на структуры 324
 Упаковка полей 326

У

Указатели 173
 typedef и указатель на массив 308
 typedef и указатель
 на функцию 308, 311
 Арифметика указателей 180
 Инициализация 177
 Ключевые слова const и volatile 187
 На данные 174
 Нулевой указатель 184
 Объявление 175
 Оператор & 177
 Оператор * 179
 Оператор const_cast 192
 Оператор reinterpret_cast 178, 193
 Оператор static_cast 183
 Указатель на указатель 186

Указатель на функцию 301
Указатель типа void* 182

Ф

Фундаментальные типы 67
Функции 18, 241
 __cdecl 255
 __fastcall 257
 __stdcall 256
 inline 252
 main 21
 printf(), scanf() 277
 Возвращаемое значение 287
 Вызов 242, 248
 Выражения в качестве параметров 264
 Значения параметров
 по умолчанию 267
 Ключевое слово const 292
 Назначение 242
 Неиспользуемые параметры 269
 Объявление 242, 244
 Определение 242, 246
 Параметры функции main 270
 Перегрузка имен функций 295
 Передача массивов в функцию 265
 Передача параметров по адресу 260
 Передача параметров
 по значению 259
 Переменное число параметров 272
 Проблемы при возвращении
 адреса 290
 Рекурсивные 298
 Соглашения о вызове 254
 Указатель на функцию 301

Э

Этапы получения загрузочного
 модуля 13