Марина Полубенцева

C/C++

ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ

Санкт-Петербург

«БХВ-Петербург»

2008

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

Полубенцева М. И.

П53 С/С++. Процедурное программирование. — СПб.: БХВ-Петербург, 2008. — 448 с.: ил. — (Внесерийная)

ISBN 978-5-9775-0145-3

Подробно рассмотрены процедурные возможности языков программирования С/С++. Изложены основные принципы строения программы на языке С/С++: 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 препроцессора. Детально рассмотрены указатели и массивы, а также их взаимосвязь в языке С/С++. Приведена сравнительная характеристика ссылок С++ и указателей. Обсуждаются сложные программные элементы. Рассмотрены агрегатные пользовательские типы данных языка С: структуры, объединения.

Для программистов и разработчиков встраиваемых систем

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

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Римма Смоляк*
Компьютерная верстка	*Натальи Караваевой*
Корректор	*Виктория Пиотровская*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

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

ISBN 978-5-9775-0145-3

© Полубенцева М. И., 2008
© Оформление, издательство "БХВ-Петербург", 2008

Оглавление

Введение ... **1**

Предисловие ... 2

Особенности изложения ... 3

Благодарности ... 5

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 *enum* ... 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 *sizeof* и размеры переменных ... 69
3.4.4. Знаковость переменной ... 71
3.4.5. Приведение типов ... 73
3.4.6. Тип *wchar_t* ... 81
3.4.7. Тип *bool* и *BOOL* ... 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 (*type-safe linkage*) ... 88
3.7. Размещение и время существования переменных ... 89
3.7.1. Ключевое слово *static* ... 91
3.8. Область видимости переменной (*scope*) ... 94
3.8.1. Скрытие (замещение) имени переменной ... 95
3.8.2. Пространства имен — *namespace* ... 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 *const* и *volatile* ... 109
3.10.1. Ключевое слово *const* ... 109
3.10.2. Ключевое слово *volatile* ... 110

Глава 4. Инструкции (*statements*) C/C++ ... 113

4.1. Общая информация об инструкциях ... 113
4.2. Инструкции выбора (условия) ... 115
4.2.1. Инструкции *if, if...else* ... 115
4.2.2. Переключатель — *switch* ... 118
4.3. Инструкции цикла ... 122
4.3.1. Инструкция *while* ... 123
4.3.2. Инструкция *do...while* ... 127
4.3.3. Инструкция *for* ... 129
4.4. Инструкции безусловного перехода: *break*, *continue*, *return*, *goto* ... 134

Глава 5. Препроцессор. Заголовочные файлы 137
5.1. Директивы препроцессора 137
5.2. Директива *#define* 138
5.2.1. Использование директивы *#define* 139
5.2.2. Предопределенные макросы 143
5.2.3. Диагностический макрос *assert* 144
5.2.4. Рекомендации 145
5.3. Директива *#undef* 145
5.4. Директивы *#ifdef, #ifndef, #else и #endif* 147
5.5. Директивы *#if, #elif, #else, #endif*. Оператор препроцессора *defined* 149
5.6. Директива *#include*. Заголовочные файлы 152
5.6.1. Концепция разделения на интерфейс и реализацию. Механизм подключения заголовочных файлов 153
5.6.2. Формы директивы *#include* 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 *#pragma* 170
5.8. Директива *#error* 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 — & 177
6.1.3. Получение значения *объекта* посредством указателя: оператор разыменования — * 179
6.1.4. Арифметика указателей 180
6.1.5. Указатель типа *void** 182
6.1.6. Нулевой указатель (*NULL-pointer*) 184
6.1.7. Указатель на указатель 186
6.1.8. Указатель и ключевые слова *const* и *volatile* 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 [] 198
6.2.3. Инициализация массива 200
6.2.4. Массивы и оператор *sizeof* 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++ *new* и *delete* 222
6.4.3. Сборщик мусора (*garbage collector*) 225
6.4.4. Операторы *new[ ]* и *delete[ ]* 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 *inline*-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 (*Call-By-Value*) 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 *const* 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 *typedef* и сложные указатели 308
8.8.1. Ключевое слово *typedef* 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 *typedef* 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 (*nested*) структуры 323
9.10. Указатели и структуры 324
9.11. Упаковка полей структуры компилятором. Оператор *sizeof* 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 (*union*) 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, чтобы знать в чем дело, когда все наперепалох.

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 девчонкам 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 С/С++, 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 С++, 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 С/С++ 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 С/С++, 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 С++, довольно подробно описывают объектно-ориентированные возможности языка, уделяя минимум внимания базовым понятиям языка С/С++. Возможно, начинающий программист найдет в данном пособии структурированный материал по этой теме, а программист, уже имеющий опыт программирования на С/С++, 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 ;
- 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 *пользователь*, 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.

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;

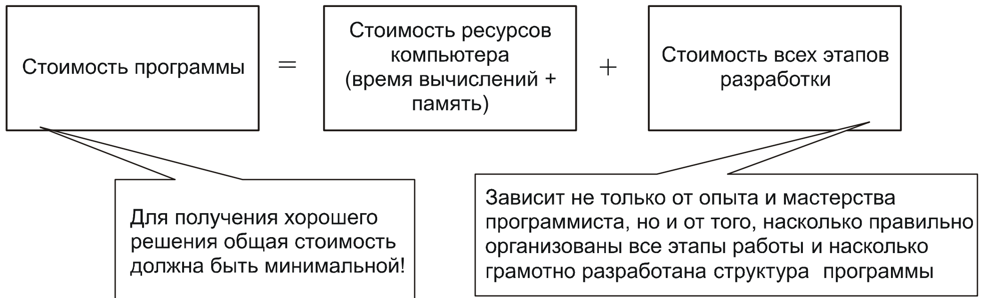

Рис. 1.2

- 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 удается 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 большего 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 приступать 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.

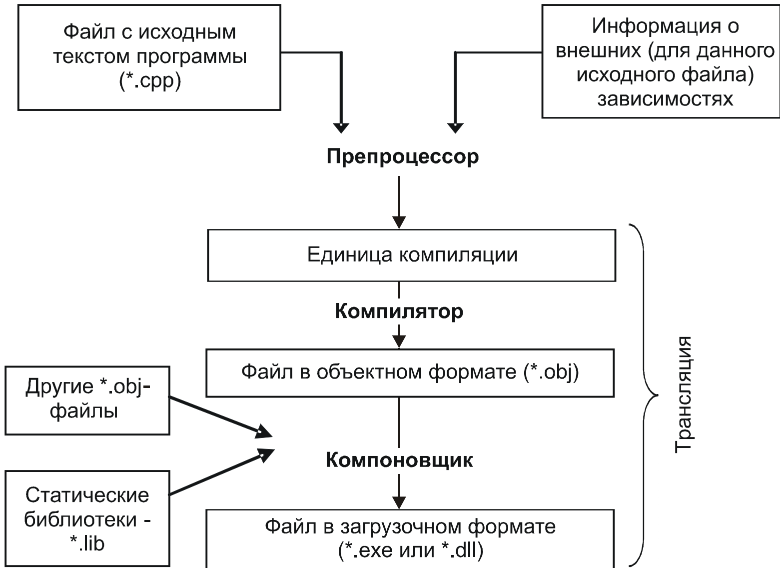

Рис. 1.5

2. 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).
3. 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.
4. 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 С/С++ 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 ввиду 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.

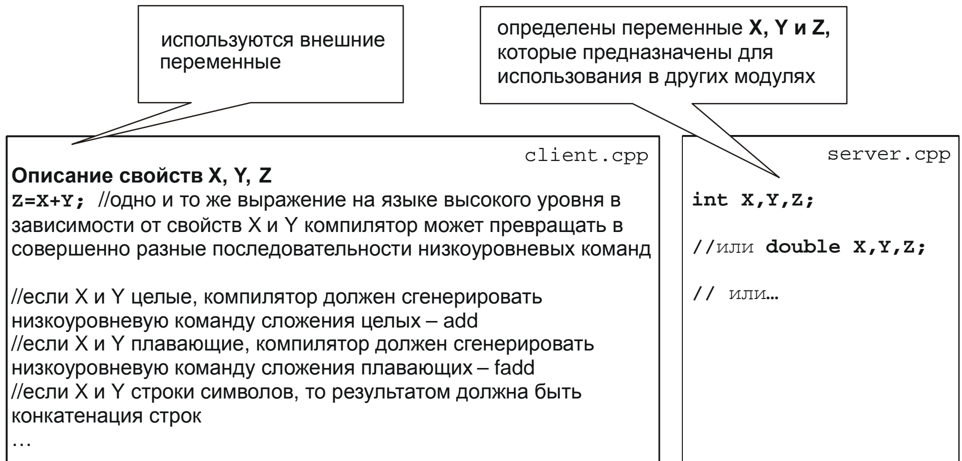

Рис. 1.6

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;

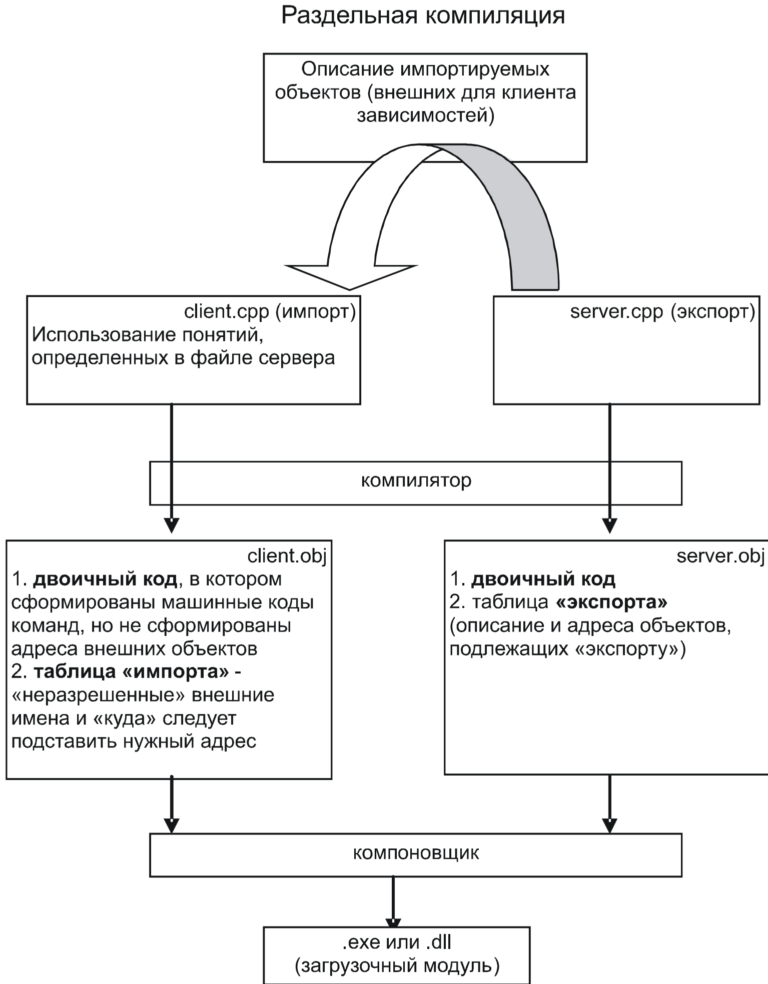

Рис. 1.7

- *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).

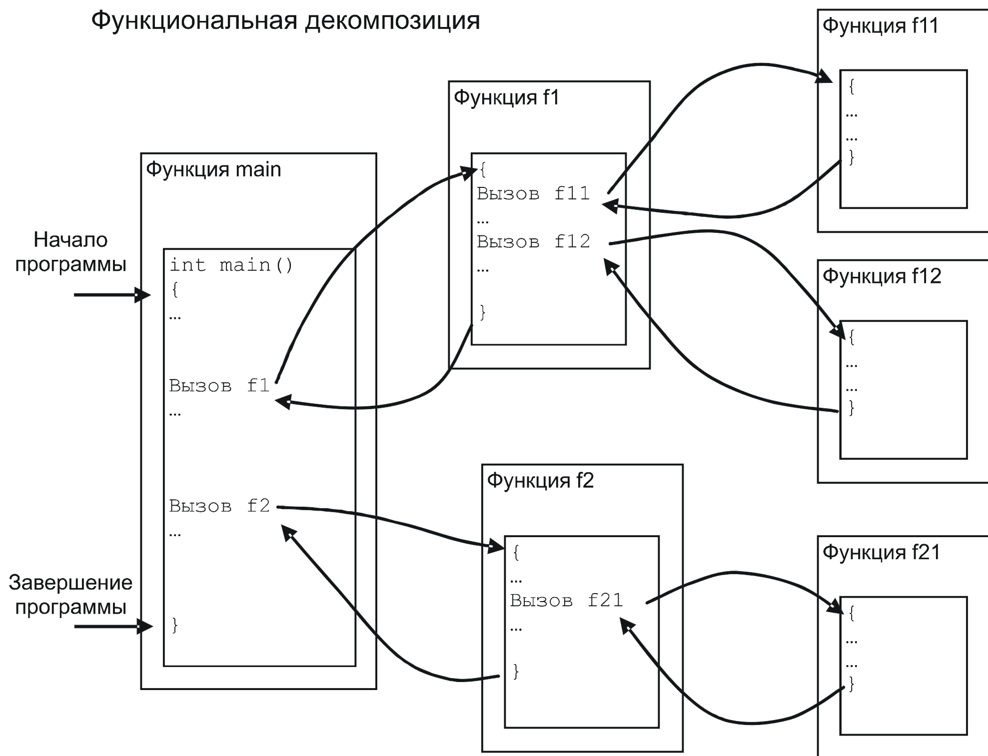

Рис. 1.8

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.

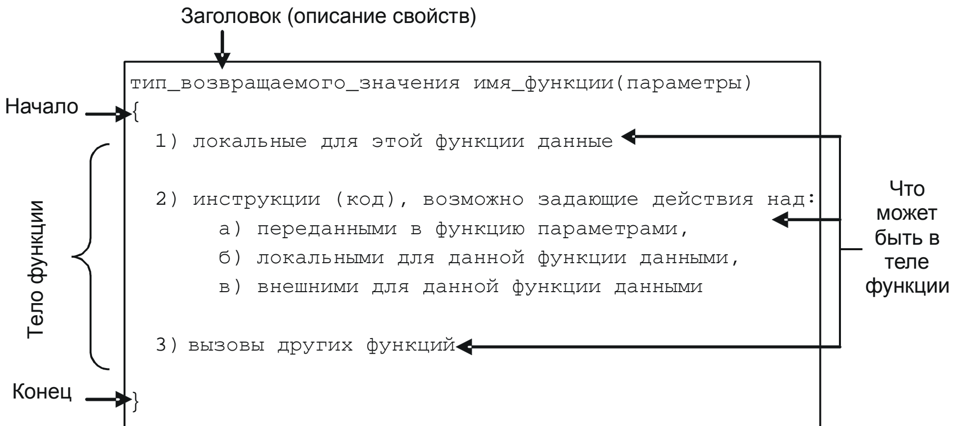

Рис. 1.10

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[ ] ])`

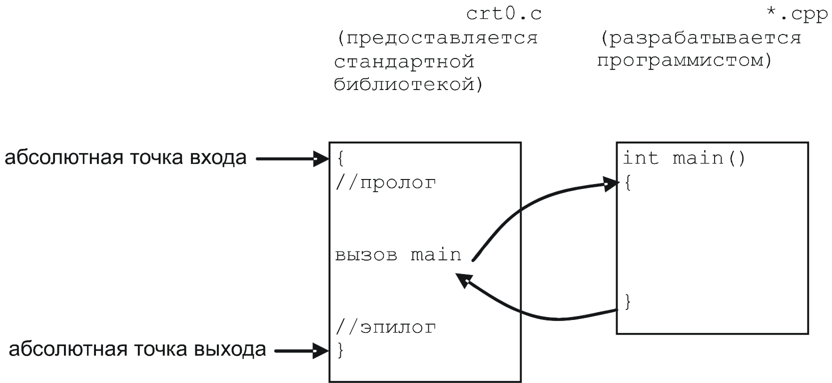

Рис. 1.11

- 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.

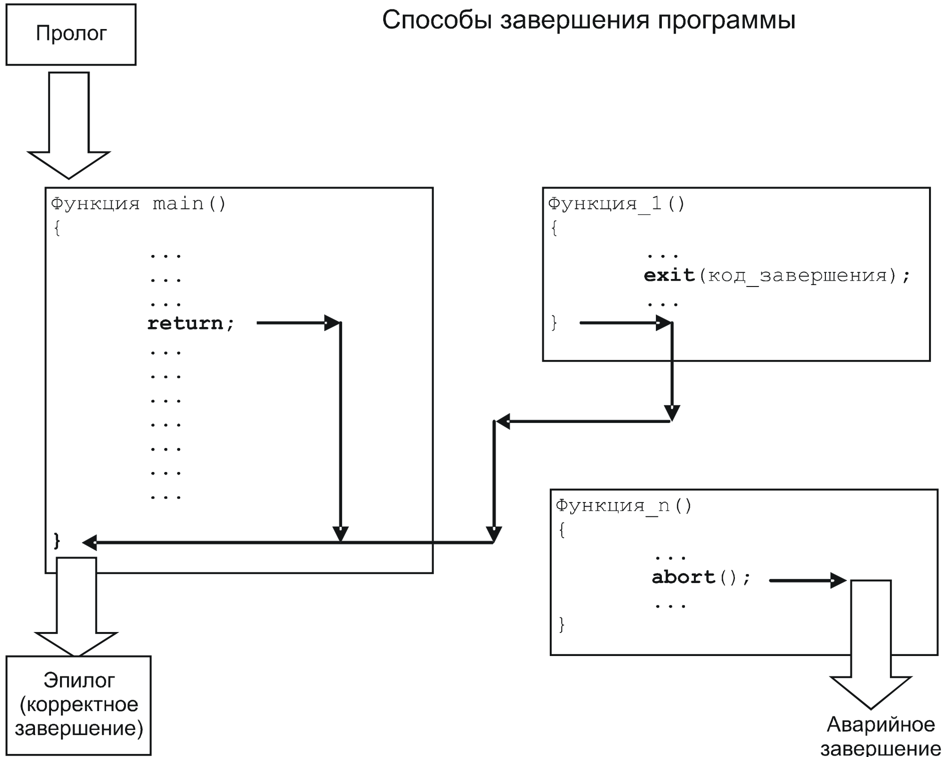

Рис. 1.12

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).