



МАРИНА ПОЛУБЕНЦЕВА

On-line-поддержка книги

C/C++

ПРОЦЕДУРНОЕ ПРОГРАММИРОВАНИЕ

Пространства имен

Препроцессор

Указатели и массивы

Динамическое выделение памяти

Ссылки

Функции

Структуры и объединения

Факультет переподготовки специалистов
Санкт-Петербургского государственного
политехнического университета
СПбПУ



<http://www.avalon.ru>
[mailto: info@avalon.ru](mailto:info@avalon.ru)
+7(812) 7030202

AVALON.RU — СОВЕДУЮТ ПРОФЕССИОНАЛЫ

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>Натальи Караваевой</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
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 <i>lvalue</i> и <i>rvalue</i>	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 (statements) C/C++	113
4.1. Общая информация об инструкциях	113
4.2. Инструкции выбора (условия)	115
4.2.1. Инструкции <i>if, 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, continue, return, 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



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

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;

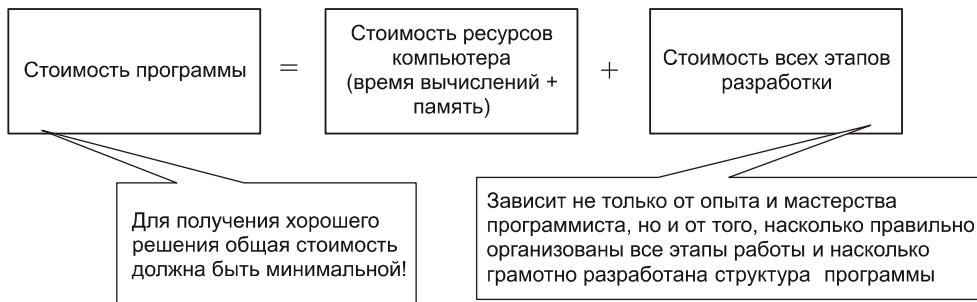


Рис. 1.2

- 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 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 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

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.

- 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 — со всем 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)

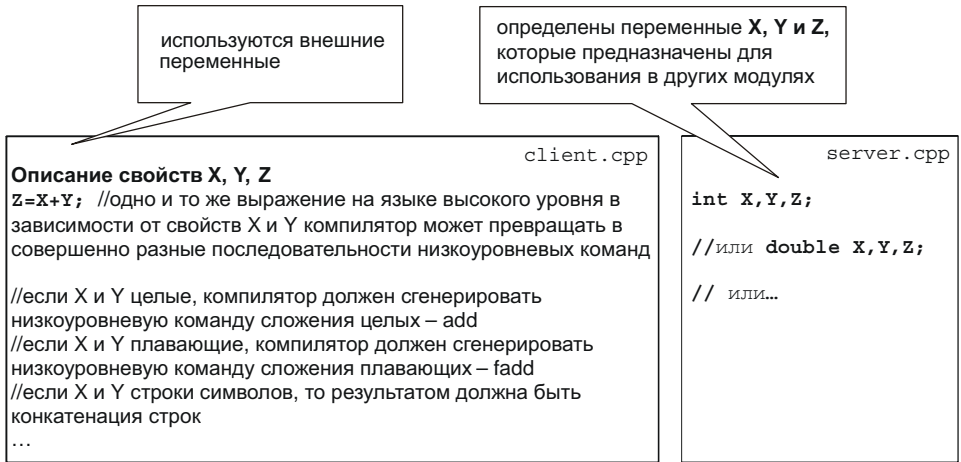


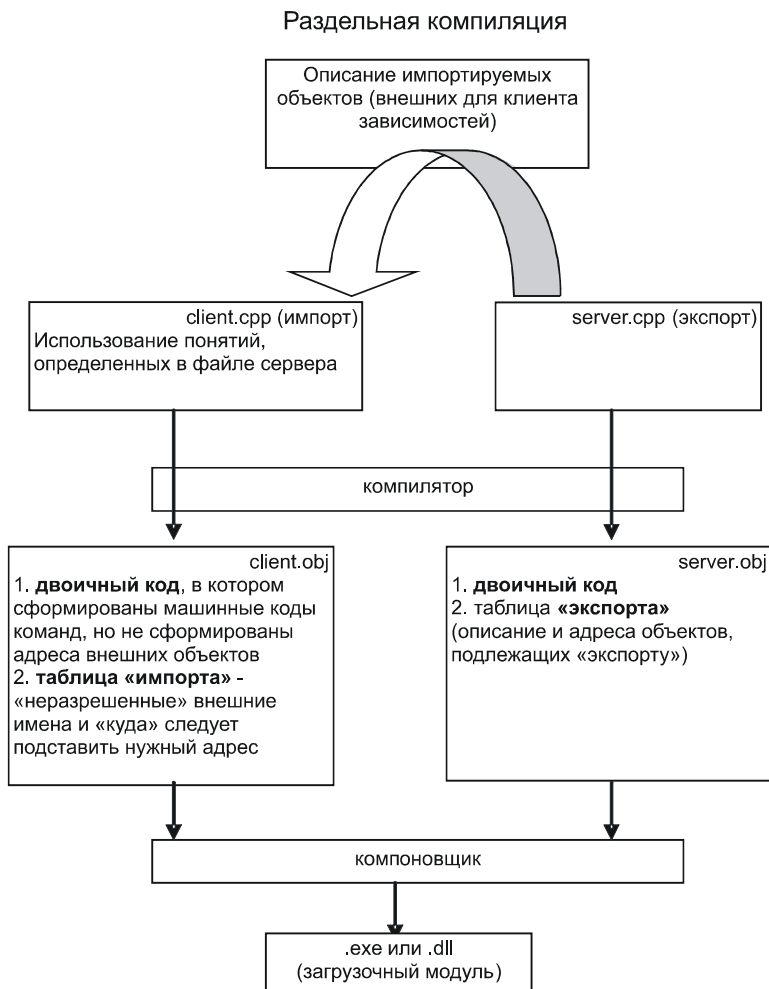
Рис. 1.6

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;



- *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.