ОГЛАВЛЕНИЕ

1 Начало 11
1.1. Введение 11
1.2. Среда разработки программ 11
1.2.1. Редактирование 12
1.2.2. Компилирование 12
1.2.3. Редактирование связей 14
1.3. Программа на C++ 14
1.3.1. Комментарии в программах 15
1.3.2. Заголовочные файлы 16
1.3.3. Синтаксис программы 17
1.3.4. Служебные слова 17
1.3.5. Тип возвращаемого значения 18
1.3.6. Тело функции main() 18
1.4. Функции 19
1.4.1. Программа с вызовом функции 20
1.5. Базовые типы данных 23
1.6. Функции с параметрами и возвращаемыми значениями 26
1.7. Заключение 28
1.8. Литература 29

2 Базовые сведения о параллельном порте и работа с ним 30
2.1. Введение 30
2.2. Что такое параллельный порт? 30
2.2.1 Цифровые логические схемы 31
2.2.2. Устройство параллельного порта 32
2.3. Представление данных 35
2.4. Программа для отображения шестнадцатеричных и десятичных чисел 37
2.5. Заключение 38
2.6. Литература 38

3 Тестирование параллельного порта 39
3.1. Введение 39
3.2. Источник питания интерфейсной платы 39
3.3. Интерфейс параллельного порта 42
3.3.1. Схема драйвера светодиодов 44
3.3.2. Работа светодиода 44
3.4. Элементарный вывод через параллельный порт 46
3.5. Ввод через параллельный порт 48
3.6. Коррекция внутренней инверсии 52
3.6.1. Вывод данных 52

3.6.2. Работа в качестве входа ... 55
3.7. Заключение ... 56
3.8. Литература ... 57

4 Объектно-ориентированное программирование ... 58
4.1. Введение ... 58
4.2. Воображаемые и реальные объекты ... 58
4.3. Реальные объекты ... 59
4.3.1. Внешний интерфейс объекта ... 60
4.3.2. Создание и уничтожение объекта ... 61
4.4. Классы объектов ... 61
4.5. Инкапсуляция ... 62
4.5.1. Инстанцирование объектов ... 63
4.6. Абстрактные классы ... 63
4.7. Иерархии классов ... 64
4.8. Наследование ... 64
4.9. Множественное наследование ... 65
4.10. Полиморфизм ... 66
4.11. Пример иерархии объектов ... 66
4.12. Преимущества объектно-ориентированного программирования ... 71
4.13. Недостатки объектно-ориентированного программирования ... 71
4.14. Заключение ... 72
4.15. Литература ... 72

5 Объектно-ориентированное программирование ... 73
5.1. Введение ... 73
5.2. Правила обозначения ... 73
5.3. Разработка класса ... 74
5.3.1. Данные-члены ... 75
5.3.2. Функции-члены ... 75
5.3.3. Атрибуты доступа ... 75
5.3.4. Определение класса ... 76
5.3.5. Конструктор ... 77
5.3.6. Автоматический конструктор ... 78
5.3.7. Перегрузка конструкторов ... 78
5.3.8. Деструкторы ... 78
5.4. Класс ParallelPort – этап 1 ... 79
5.4.1. Определение класса ... 79
5.5. Программирование с классами ... 83
5.5.1. Примеры с атрибутами доступа ... 86
5.6. Класс ParallelPort – этап 2 ... 89
5.7. Класс ParallelPort – этап 3 ... 93
5.7.1. Полнофункциональный класс ParallelPort ... 93
5.8. Заключение ... 96

5.9. Литература ... 97

6 **Цифро-аналоговое преобразование ... 98**
6.1. Введение ... 98
6.2. Цифро-аналоговое преобразование ... 98
6.2.1. Основные сведения об операционном усилиетеле ... 99
6.2.2. Принципы работы ЦАП ... 101
6.2.3. Работа DAC0800 ... 105
6.2.4. Характеристики и параметры ЦАП ... 108
6.3. Работа с цифро-аналоговым преобразователем ... 109
6.4. Производные классы ... 112
6.5. Добавление членов к производному классу ... 119
6.5.1. Спецификаторы доступа ... 122
6.5.2. Полиморфные функции ... 124
6.6. Заключение ... 134
6.7. Литература ... 134

7 **Управление светодиодами ... 135**
7.1. Введение ... 135
7.2. Циклы ... 135
7.2.1. Цикл for ... 135
7.2.2. Циклы while и do-while ... 138
7.3. Переходы ... 139
7.3.1. Выражение if ... 139
7.3.2. Выражения break и continue ... 141
7.3.3. Выражение switch-case ... 142
7.4. Массивы ... 143
7.5. Указатели ... 146
7.5.1. Объявление указателей ... 147
7.5.2. Указатели на скалярные величины ... 148
7.5.3. Указатели на объекты классов ... 149
7.5.4. Указатели на массивы ... 150
7.5.5. Массивы указателей ... 152
7.5.6. Арифметические операции над указателями ... 152
7.5.7. Указатели на функции ... 155
7.5.8. Указатели на void ... 158
7.5.9. Указатель this ... 159
7.6. Работа с указателями ... 160
7.6.1. Массивы чисел для светодиодов ... 160
7.7. Макросы ... 168
7.8. Динамическое выделение памяти ... 169
7.9. Обработка исключений ... 172
7.10. Заключение ... 177
7.11. Литература ... 178

8 Управление шаговыми и коллекторными электродвигателями..... 179
8.1. Введение 179
8.2. Двигатели постоянного тока 179
8.2.1. Конструкция и характеристики двигателей постоянного тока 180
8.2.2. Управление коллекторным двигателем 181
8.3. Шаговые двигатели 182
8.3.1. Конструкции шаговых двигателей 183
8.3.2. Устройство шаговых двигателей 183
8.3.3. Управление шаговым двигателем 189
8.3.4. Характеристики шаговых двигателей 190
8.4. Иерархия классов для двигателей 191
8.5. Введение в виртуальные функции 193
8.5.1. Чистая виртуальная функция 196
8.5.2. Формирование сигнала с ШИМ 203
8.6. Виртуальные функции в приложении 211
8.6.1. Виртуальные деструкторы 216
8.7. Ввод с клавиатуры 232
8.8. Заключение 245
8.9. Литература 245

9 Методы программирования 247
9.1. Введение 247
9.2. Методы эффективного программирования 247
9.3. Модульные программы 255
9.3.1. Разбиение программы на модули 255
9.3.2. Сборка многофайловой программы 256
9.4. Пример программы управления двигателями 261
9.5. Заключение 273
9.6. Литература 273

10 Измерение напряжения и температуры 274
10.1. Введение 274
10.2. Преобразование напряжения в последовательность импульсов 274
10.3. Измерение температуры 275
10.4. Класс ГУН 276
10.5. Измерение напряжения с помощью ГУН 280
10.6. Работа с графикой – отображение прямоугольных импульсов 287
10.6.1. Работа с экраном 288
10.7. Измерение температуры 292
10.7.1. Калибровка термистора 293
10.8. Заключение 297
10.9. Литература 297

11 Аналого-цифровое преобразование 298
11.1. Введение 298
11.2. Аналого-цифровое пребразование 298
11.3. Методы преобразования 301
11.4. Измерение напряжения при помощи АЦП 307
11.5. Класс АЦП 313
11.6. Измерение напряжения при помощи АЦП 320
11.7. Измерение температуры при помощи АЦП 323
11.8. Заключение 326
11.9. Литература 326

12 Сбор данных с использованием перегрузки операторов 327
12.1. Введение 327
12.2. Перегрузка операторов 327
12.2.1. Передача параметров функции по значению 329
12.2.2. Передача парметров функции по ссылке 330
12.2.3. Выбор способа передачи параметров 331
12.2.4. Конструктор копии 335
12.2.5. Перегрузка операторов при помощи функций-членов 342
12.2.6. Перегрузка операторов при помощи обычных функций 344
12.2.7. Дружественные связи 346
12.2.8. Потоки ввода/вывода 348
12.2.9. Транзитные объекты 349
12.2.10. Оператор присвоения 351
12.3. Сбор данных 354
12.4. Заключение 358
12.5. Литература 358

13 Таймер персонального компьютера 359
13.1. Введение 359
13.2. Устройство таймера персонального компьютера 359
13.2.1. Конфигурирование счётчиков 361
13.2.2. Регистр управления 362
13.2.3. Режимы работы таймеров 364
13.2.4. Чтение данных таймера 366
13.3. Работа с таймером 367
13.3.1. Чтение текущего значения таймера 0 и количества тиков 368
13.4. Класс PCTimer 368
13.5. Измерение времени 374
13.6. Измерение скорости реакции человека 376
13.7. Формирование развёртки во времени 378
13.8. Сбор данных с метками времени 381

13.8.1. Схема заряда/разряда ... 381
13.8.2. Сбор данных с метками времени ... 382
13.9. Заключение ... 387
13.10. Литература ... 388

A Приложение. Электронное оборудование ... 389

Принципиальная схема ... 389
Печатная плата ... 389
Сборка ... 390
Пайка ... 392
Правила чтения принципиальной схемы ... 393
Наладка ... 393
Демонтаж компонентов ... 394
Соединительные кабели и провода ... 395
Блок питания ... 396
Интерфейс параллельного порта ... 399
Драйвер светодиодов ... 402
Цифро-аналоговый преобразователь ... 404
Схема управления двигателями ... 408
Управляемый напряжением генератор импульсов ... 412
Аналого-цифровой преобразователь ... 415
Мультиплексор ... 418
Управляемый источник тока ... 421
Повторитель напряжения ... 423
Схема заряда/разряда ... 425
Пара светодиод-фотоприёмник ... 427
Кнопка, потенциометр, диод и стабилитрон ... 428
Перечень элементов и материалов интерфейсной платы ... 430

B Приложение ... 434

Служебные слова C++ ... 434
Приоритет операторов ... 435
Символы ASCII ... 436

Предметный указатель ... 437

1 Начало

Содержание главы:

- Что такое разработка программ?
- Написание и запуск первой программы на C++.
- Синтаксис программы.
- Функции.
- Основные типы данных.

1.1. Введение

В этой главе даны базовые сведения для начала программирования на C++. Будут сщзданы несколько простых программ на C++, а также освоен синтаксис и правила написания программ. Одной из основных структурных единиц любой программы на C++ является функция. Будут рассмотрены основные понятия и использование функций в C++. В C++ есть встроенные типы данных, на базе которых могут быть созданы пользовательские типы данных. Некоторые из этих типов данных описаны в этой главе.

По ходу главы мы поэтапно рассмотрим весь процесс создания программы: от проектирования небольшой программы до создания исполняемого файла при помощи *среды разработки программ*. При этом не будет задействован объектно-ориентированный подход, поэтому программы будут просты для понимания на начальном уровне. Основы объектно-ориентированного программирования будут даны в главе 4 и далее будут повсеместно использоваться.

1.2. Среда разработки программ

Разработка программы проходит в несколько этапов. Для создания программы понадобятся: *редактор*, *компилятор* и *редактор связей*. В современных средствах разработки программ эти средства интегрированы в один пакет и весь процесс протекает незаметно для пользователя. Такие пакеты называются *интегрированными средами разработки* или *IDE* (***I**ntegrated **D**evelopment **E**nvironment*). Большинство современных пакетов C++ (программы для создания программ на C++) представляют собой IDE. В качестве примеров можно привести Turbo C++, Borland C++, C++ Builder и Visual C++, являющиеся коммерческими пакетами. Существуют так называемые версии для *командной строки*. Для запуска редактора в таких версиях необходимо набрать команду (в командной строке DOS). Затем нужно набрать другую команду для запуска компилятора и т. д.

Кроме редактора, компилятора и редактора связей пакеты предоставляют возможность использования библиотек. Иногда эти библиотеки называют библиотеками времени выполнения (***r**un-**t**ime **l**ibrary, RTL*). В них содержатся разнообразные функции, которые можно использовать в программах. Важно понимать, что происходит на каждом этапе, независимо от используемого пакета. В следующих разделах будут описаны редактирование, работа препроцессора, компилирование и работа редактора связей.

1.2.1. Редактирование

Первым шагом при создании программы является набор её текста в каком-либо редакторе. Для этого подходит не каждый редактор. В DOS можно пользоваться командой *edit*, а в Windows можно воспользоваться редактором *Notepad*. В интегрированных средах разработки (IDE), входящих в состав пакетов C++, есть встроенные редакторы, так называемые текстовые редакторы. После редактирования нужно записать содержимое редактора в файл. Два упомянутых выше редактора запишут в файл только то, что было набрано с клавиатуры. Они не записывают в файл дополнительные символы (в отличие от некоторых других редакторов). Всё, что мы обычно печатаем, состоит из цифр, букв, знаков пунктуации, пробелов, знаков табуляции, возвратов каретки и переводов строки. Символ перевода строки используется редактором для перемещения курсора на следующую строку. Возврат каретки устанавливает курсор на начало строки. Файл с программой не должен содержать символы, не относящиеся к вышеперечисленным. Файл, в котором находятся все команды программы (текст программы), называется *исходным файлом* (*source file*). Об исходном файле говорят, что он содержит *исходный код* (*source code*), являющийся ничем иным, как командами программы.

1.2.2. Компилирование

Второй этап – это *компилирование* исходного файла. Компилирование осуществляется при помощи специальной программы, которая называется *компилятор*. Сначала выполняется входящая в состав компилятора программа, называемая *препроцессором*. Это происходит перед компилированием исходного кода. Препроцессор обрабатывает в исходном коде все выражения, начинающиеся со знака "`#`". Далее в листингах программ можно найти строки, начинающиеся с символа "`#`". Эти выражения называются *директивами препроцесора*. Препроцессор выполняет действия, предписанные этими выражениями, и вносит соответствующие изменения в текст исходного файла. После препроцессора все строки, начинающиеся символом "`#`", уже обработаны и не обрабатываются компилятором. Этот процесс показан на **Рис. 1.1**. Есть тенденция к объединению препроцессора и компилятора – большинство современных компиляторов имеют встроенные препроцессоры.

Полученный после препроцессора файл затем обрабатывается компилятором, который создаёт так называемый *объектный файл*. Объектный файл содержит объектный код, называемый также *машинным кодом* и «понятный» центральному процессору (ЦП) компьютера. Но всё же компьютер не может выполнять объектный код, так как пока ещё отсутствуют некоторые детали. На этом этапе программа похожа на недостроенное шоссе, где одни участки готовы, а другие ещё нет. Поэтому скомпилированная программа ещё не может выполняться компьютером.

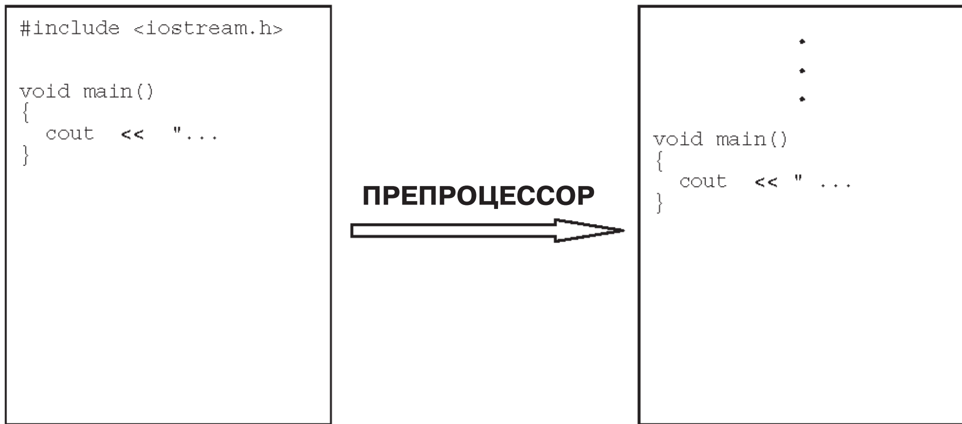

Рис. 1.1. Препроцессор обрабатывает все строки, начинающиеся символом “#”

На этой незавершённой стадии в объектном коде имеются *неопределённые ссылки*. Неопределённая ссылка указывает на хранящиеся где-то фрагменты объектного кода, которые нужно вставить в программу. В объектном файле нет повсеместно определённого кода, точно так же, как и в случае недостроенного шоссе. Процесс компилирования показан на **Рис. 1.2**.

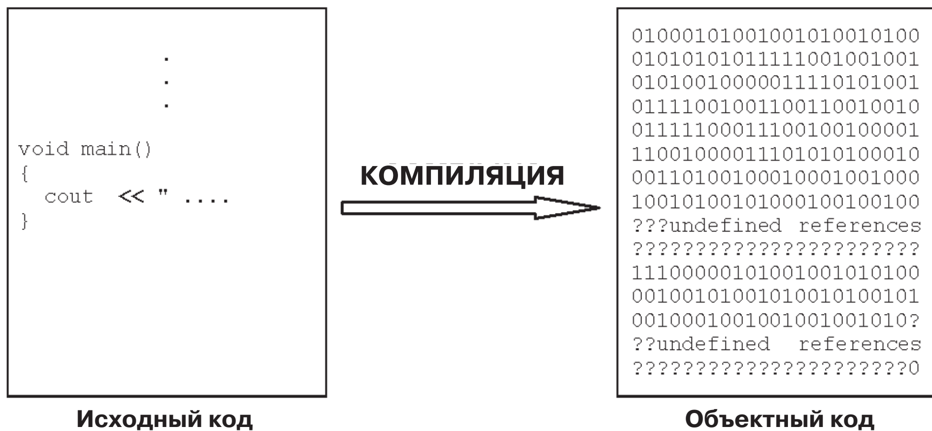

Рис. 1.2. Компилятор преобразовывает исходный код в объектный код

Синтаксис (правила написания) выражений программы крайне важен. Синтаксис определяет правила использования знаков пунктуации в исходном файле. Большинство знаков пунктуации служат *разделителями*. Разделитель отделяет переменные, служебные слова, числа, выражения и т. д. Пробел, запятая, точка с запятой, двоеточие, скобки являются разделителями в различных вариантах использования. Компиляторы обладают ограниченной интеллектуальностью. Если вы пропустите точку с запятой, то это будет обнаружено компилятором и он сообщит об ошибке, хотя сам исправить эту ошибку не сможет.

Выше упоминалось, что объектный код содержит неопределённые области и не может выполняться. Объектный код, например, может содержать вызовы различных подпрограмм. В объектном файле есть вызовы этих функций. Инструкции, на которые ссылаются вызовы, на самом деле отсутствуют. Эти инструкции могут находиться в этом же объектном файле, могут быть в библиотечном файле или другом объектном файле. Заметим, что поиск недостающей информации не является задачей компилятора – в общем случае компилятор можно рассматривать как преобразователь, выполняющий синтаксический анализ содержимого исходного файла.

1.2.3. Редактирование связей

Программа, заполняющая неопределённые участки и осуществляюшая сборку программы, называется *редактором связей (linker)*. Она ищет недостающие инструкции во всех объектных файлах и библиотеках. Иногда редактору связей нужно указать библиотеки и объектные файлы для поиска. Это могут быть как приобретённые библиотеки и объектные файлы сторонних производителей, так и созданные собственными силами. Редактор связей автоматически находит файлы библиотек и объектные файлы, относящиеся к пакету C++, каждый из которых называется библиотекой времени выполнения. Недостающие инструкции редактор связей записывает в соответствующие места, ликвидируя «дыры» в программе. Этот процесс называется редактированием связей. После редактирования связей получается файл, который может быть выполнен компьютером – так называемый *исполняемый файл*. Чтобы программа могла быть выполнена, её нужно загрузить в память компьютера. Это делает загрузчик, являющийся частью исполняемого кода. Большинство редакторов связей помещают загрузчик в начало исполняемого файла. Поэтому когда мы запускаем на выполнение программу, то сначала выполняется загрузчик и загружает программу в память, а затем уже начинается выполнение собственно программы. На **Рис. 1.3** изображён процесс редактирования связей.

1.3. Программа на C++

С точки зрения компьютера программа представляет собой набор инструкций, которые нужно выполнить. Программист упорядочивает эти инструкции различным образом, в зависимости от требуемых от компьютера действий. Простой пример: если вы хотите написать программу сложения двух чисел, числа сначала должны быть введены, а затем выполнено сложение. Значит, сначала должна выполняться команда чтения чисел, а затем выполнено их сложение.

У каждого языка программирования свой уникальный синтаксис. Синтаксис определяет оформление программы и использование знаков препинания. Здесь будет изучаться синтаксис языка программирования C++. Выше упоминалось, что основным структурным элементом программы на C++ можно считать функцию – процедуру, возвращающую результат. Поэтому в программе вместе с набором исполняемых команд обязательно ***должна*** быть функция. Одна из функций является особой и называется `main()`. Чтобы в тексте различать функции и другие имена, мы будем ставить пару скобок после имени функции. Простые про-

граммы могут иметь только одну функцию `main()`. Наша первая программа будет печатать сообщение на экране. Программа приведена в **Листинге 1.1**.

Листинг 1.1. Программа для вывода сообщения на экран.

```
/* Эта программа печатает текстовое сообщение на экране.
   Программа состоит только из одной функции main(). */
#include <iostream.h>

// Функция main()
void main()
{
    cout << "Getting Started" << endl;
}
```

После запуска этой программы на экран будет выведено сообщение:

```
Getting Started
```

Смысл текста этой программы будет объяснён далее.

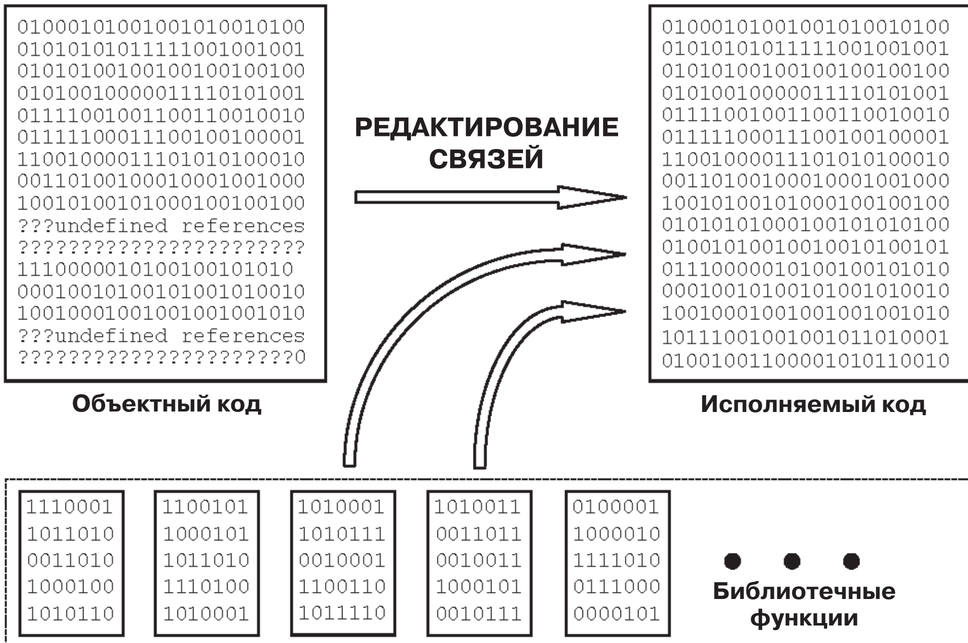

Рис. 1.3. Редактор связей ликвидирует неопределённые участки

1.3.1. Комментарии в программах

Комментарии – это пояснения, включенные в программу, чтобы программист мог описывать её работу. На практике они описывают программу или её отдельные части и не являются выполняемыми компьютером командами программы.

Комментарии должны быть соответсвующим образом обозначены в программе, чтобы он не считал их кодом при подготовке исполняемой пограммы. Есть два разных способа записи комментариев:

- Однострочные и многострочные комментарии могут быть обозначены символами "/*" в начале комментария и "*/" в конце комментария.
- Однострочный комментарий начинается с символов "//".

В **Листинге 1.1** присутствуют однострочный и многострочный комментарии. Многострочный комментарий имеет вид:

```
/* Эта программа печатает текстовое сообщение на экране.
   Программа состоит только из одной функции main(). */
```

А это однострочный комментарий:

```
// Функция main()
```

Текст между "/*" и "*/" игнорируется компилятором. То же относится и к тексту после "//" на той же строке.

1.3.2. Заголовочные файлы

В первой строке после многострочного комментария в **Листинге 1.1** находится директива включения:

```
#include <iostream.h>
```

Она заставляет препроцессор заменить её содержимым файла `iostream.h`. В нашей программе это происходит непосредственно перед функцией `main()`. Файлы с расширением «`.h`" называются *заголовочными файлами* или *включаемыми файлами*. Заголовочный файл может входить в состав пакета C++ или быть написанным программистом самим. Если этот файл из пакета C++, то он размещается в специальном каталоге, называемом *каталогом заголовочных файлов (Include Directory)*, как в случае файла `iostream.h`. В программе может быть несколько директив включения, приводящих к просмотру компилятором нескольких заголовочных файлов.

Заголовочные файлы – это текстовые файлы, содержащие выражения C++, в большинстве своём не являющиеся исполняемыми командами. Не все выражения в рассматриваемой программе являются исполнимыми командами. Тем не менее, выражения из заголовочного файла играют важную роль при формировании программы. Большинство выражений заголовочного файла способствует тщательной проверке компилятором выражений в вашей программе. После написания и тестирования заголовочных файлов они больше не изменяются. Если компилятор сообщает об ошибке или несоответствии, то изменения нужно вносить в программу, а не в заголовочный файл.

Библиотечные функции – это готовые к применению программы. При написании библиотек с функциями нужно создавать для них и заголовочные файлы. В заголовочных файлах устанавливаются правила использования библиотечных функций, строго контролируемые компилятором при написании программы.

В **Листинге 1.1** присутствуют такие элементы, как cout, двойной знак «меньше» << и endl. В данном случае они не имеют никакого отношения к языку С++. Компилятор не сможет их обработать, пока не будет дано описание этих элементов и правил их использования. В заголовочном файле iostream.h содержатся все необходимые указания для компилятора по правильному использованию этих элементов. Эта информация должна быть предоставлена до появления cout, << и endl в программе. Компилятору не нужно всё содержимое файла iostream.h для преобразования программы из **Листинга 1.1** в понятный компьютеру код. В нашем случае нужна только та часть, где описываются cout, endl и действия оператора <<. Но на самом деле очень трудно определить нужные для каждой конкретной программы части заголовочного файла. Поэтому компиляторы вынуждены обрабатывать их полностью. Размер заголовочных файлов никак не влияет на размер исполнимых файлов, только лишь незначительно увеличивает время обработки программы. При необходимости мы можем использовать несколько заголовочных файлов. Также один заголовочный файл может включать другой заголовочный файл.

В заключение отметим, что сначала нужно включить в программу заголовочный файл с объявлениями констант, типов данных и функций и только после этого начинать использовать их в программе.

1.3.3. Синтаксис программы

Синтаксис касается применения знаков пунктуации в программе. В нашей программе использованы символ решетка "#", угловые скобки (<>), пары круглых скобок, фигурные скобки ({}), точка с запятой и двойной знак «меньше» <<. Эти знаки должны быть расставлены в соответствующих местах, чтобы компилятор «понял» программу. Программа в **Листинге 1.1** написана с применением основного синтаксиса. Чем сложнее программа, тем более усложняется её синтаксис.

Все строки, начинающиеся с символа решетки "#", являются директивами препроцессора, обсуждавшегося выше, и имеющего особую роль в пакете разработки программ.

В нашей программе только одна функция – main(). Начало *тела функции* main() обозначено открывающейся фигурной скобкой "{", а конец – закрывающейся фигурной скобкой "}". Между скобками находятся команды программы. Выполнение программы всегда начинается со строки, содержащей main(), а завершается на закрывающейся фигурной скобке тела функции.

Синтаксис функции main() можно выразить в компактной форме:

```
void main() {выражение1; выражение2; выражение3;}
```

Это функция с именем main(). Пара круглых скобок после имени main могут быть как пустыми, так и заполненными – в нашей простой программе они пусты. Как можно было заметить, выражения отделяются друг от друга точкой с запятой. Точка с запятой необходима и в конце последнего выражения, хотя и кажется ненужной.

1.3.4. Служебные слова

Служебные слова являются уже занятыми для нужд языка программирования. Их нельзя использовать в качестве любых элементов программы, а только по

назначению, в соответствии с правилами языка программирования. Служебное слово, например, не может быть *идентификатором*. Идентификаторы – это придуманные нами имена для обозначения таких объектов, как функции, пользовательские типы данных и сами данные. Пока нам встретилось только одно ключевое слово – `void`. Перечень служебных слов приведён в конце книги (приложение А).

1.3.5. Тип возвращаемого значения

Слово `void` справа от функции `main()` определяет *тип возвращаемого значения* этой функции. Тип возвращаемого значения должен указываться для любой функции, будь то `main()` или какая-либо другая функция. Возвращаемое значение играет роль результата или итога. Если функция должна возвращать значение, то программисту нужно указать тип возвращаемого (выдаваемого) значения. Можно написать функцию, которая ничего не возвращает. Такие функции обычно выполняют какие-либо действия, но не формируют никаких возвращаемых значений. В этом случае тип возвращаемого значения будет `void`. В нашей программе как раз такая функция. Заметим, что если функция не возвращает значение, это должно указываться словом `void`.

Примечание

Если для функции `main()` не указан тип возвращаемого значения, то по умолчанию будет подразумеваться тип `integer`. Это значит, что функция обязана возвращать результат типа `integer`.

1.3.6. Тело функции `main()`

Тело функции `main()` состоит только из одного выражения. Эта строка заключена в фигурные скобки "`{`" и "`}`". Если тело функции `main()` состоит более чем из одного выражения, то все они должны находиться между этими скобками. Это единственное выражение нашей программы выглядит так:

```
cout << "Getting Started" << endl;
```

Слово `cout` заставляет компьютер направить *поток* из того, что следует за этим словом в стандартное устройство вывода, в данном случае на экран. Язык C++ поддерживает работу с потоками. На данном этапе достаточно понимать, что поток – это упорядоченная передача каких-либо элементов (символов, целых чисел и т. д.) куда-либо. Сначала на экране появится `Getting Started`. Затем на экран будет направлено `endl`. В результате этого курсор будет перемещён в начало следующей строки. На этом выполнение программы завершится.

Можно поэкспериментировать, заменив предыдущее выражение на другое:

```
cout << "Getting Started";
```

На экран будет направляться только `Getting Started`, без `endl`. Курсор будет находится в конце фразы "`Getting Started`".

1.4. Функции

Выше было сказано, что функции являются важной неотъемлемой частью языка C++. В этом разделе будет рассмотрена работа с функциями. Ранее говорилось, что функция может рассматриваться как процедура, формирующая результат, используя входные данные. Входные данные передаются в функцию при помощи *параметров* или *формальных аргументов*. Формальные аргументы служат для указания типа принимаемых функцией значений при её написании. Когда функция выполняется на компьютере, то формальные аргументы заменяются *фактическими аргументами*. Например, мы можем создать функцию с формальным аргументом `a`. При выполнении функции формальный аргумент нужно заменить фактическим аргументом, например числом 3. Одну и ту же функцию можно ***вызывать*** (выполнять) много раз, меняя при этом фактические аргументы и получая разные возвращаемые значения. Отметим, что кроме обычного способа данные из функции можно передавать и по-другому.

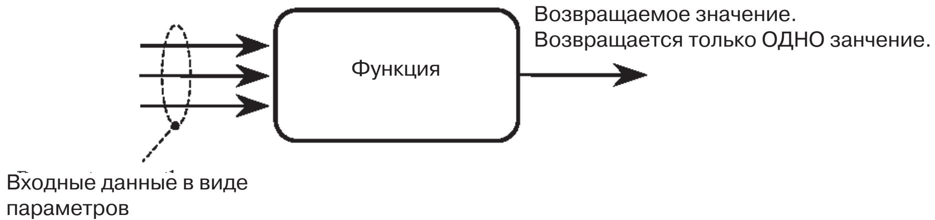

Рис. 1.4. Функция в общем случае

До этого момента функция рассматривалась только в общем виде. Это иллюстрирует **Рис. 1.4**. Существует несколько частных случаев. В этих случаях функция принимает или не принимает аргументы, возвращает или не возвращает результат. Функции могут иметь различное количество аргументов.

Функция всегда возвращает только один результат, и это должна быть *скалярная величина*. Под термином скалярная величина понимается целый и неделимый объект. Другими словами, функция не может возвращать *массивы* (наборы объектов). Функция может возвращать, например, целое число. Она не может вернуть несколько целых чисел. На **Рис. 1.5** и **1.6** показаны некоторые типичные виды функций.

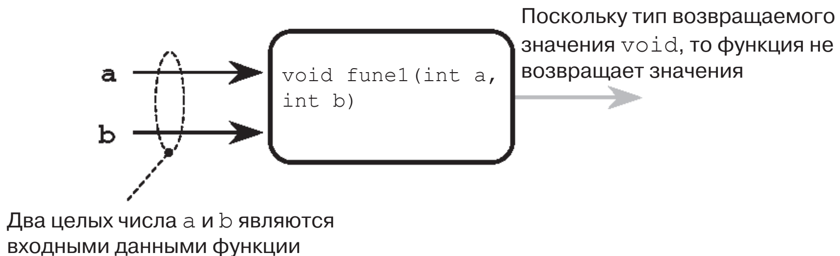

Рис. 1.5. Функция принимает два аргумента и не возвращает результата

Функция, соответствующая **Рис. 1.5**, может, например, выполнять вычисления и выводить результат на экран. Если задачи функции на этом исчерпаны, то нет необходимости возвращать значение.

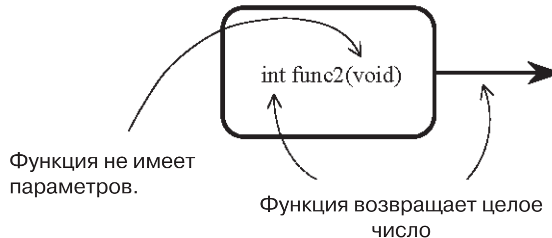

Рис. 1.6. Функция не имеет параметров, но значение возвращает

Согласно **Рис. 1.6** функция может принимать данные от внешнего источника, например, порта принтера, и возвращать целое число. Целое число, например, может указывать на наличие бумаги в принтере: 0 означает отсутствие бумаги, а 1 – её наличие.

1.4.1. Программа с вызовом функции

Программа из **Листинга 1.2** делает то же, что и программа из **Листинга 1.1**. Отличие в использовании функции для вывода данных на экран. Эта функция не принимает значений и не возвращает результат. В этом разделе обсуждается *абстрактность подпрограмм*. Абстрактность подразумевает сокрытие деталей реализации функции внутри её при вызовах для выполнения нужных действий.

Листинг 1.2. Программа с вызовом функции

```
/* Программа печатает текстовое сообщение
   на экране компъютера. Программа состоит
   из двух функций PrintMessage() и main().

#include <iostream.h>

// Функция PrintMessage()
void PrintMessage()
{
    cout << "Getting Started" << endl;
}

// Функция main()
void main()
{
    PrintMessage(); // вызов функции
}
```

В этой программе есть новая функция `PrintMessage()`. Имя `PrintMessage` мы придумали сами. К имени `PrintMessage` мы добавили пару круглых скобок, чтобы показать, что это функция. Между скобками мы ничего не пишем (что эквивалентно слову `void` между скобок), потому что функция не имеет параметров. Тип возвращаемого значения `void`, так как функция `PrintMessage()` не возвращает результат. Определение функции `PrintMessage()` выглядит следующим образом:

```
void PrintMessage()
{
    cout << "Getting Started" << endl;
}
```

В определении функции должны быть указаны четыре вещи, а именно:

1. Тип возвращаемого значения.
2. Имя функции.
3. Параметры и их типы.
4. Тело функции.

Синтаксис функции изображён на **Рис. 1.7**.

Тип возвращаемого значения

Имя функции

Параметры и их типы

```
void PrintMessage()
{
    cout << "Getting Started" << endl;
}
```

Тело функции

Рис. 1.7. Синтаксис определения функции

Определение функции является собственно функцией, компилятору указываются команды, которые нужно выполнять. Другими словами, тело функции `PrintMessage()` начинается с открывающей фигурной скобки, а заканчивается закрывающей фигурной скобкой. Обратите внимание, что после имени функции `PrintMessage()` точка с запятой не ставится. Тем самым следующее далее тело функции ставится в соответствие имени функции.

У функции `PrintMessage()` тип возвращаемого значения `void`. Имя функции `PrintMessage`. Параметров функция не имеет, а тело функции состоит из выражения `cout`.

Объявление (*declaration*) функции незначительно отличается (как показано на **Рис. 1.8**). Для компилирования вызовов функции компилятору C++ достаточно встретить её объявление до вызова функции. На этом этапе определение функции не нужно. Но для того, чтобы функция могла быть выполнена, нужно откомпилированное определение функции.

В объявлении функции указываются только три элемента:

1. Тип возвращаемого значения.
2. Имя функции.
3. Параметры и их типы.

Тип возвращаемого значения
Имя функции
Параметры и их типы

```
void PrintMessage();
```

Рис. 1.8. Объявление функции, также называемое прототипом функции

Тело функции не является обязательным элементом, но для формирования выполнимой программы оно необходимо. Если тело функции находится в библиотеке, то оно будет позаимствовано оттуда на этапе редактирования связей. Если тела функции нет в библиотеке или другом объектном файле, то его нужно определить где-либо в тексте программы. В нашем случае объявление функции будет выглядеть так:

```
void PrintMessage();
```

Обратите внимание, что в конце строки стоит точка с запятой.

В C++ *прототип* функции – это то же самое, что и её объявление. Но в языке С объявление функции и её прототип – разные вещи. Смотрите пример в разделе 1.6.

Тело функции `main()` немного изменено. Оно состоит из единственного выражения:

```
PrintMessage();
```

Обратите внимание, что в конце стоит точка с запятой, а в начале нет указания типа возвращаемого значения. Такое выражение называется *вызовом функции*. При вызове функции указываются два элемента:

1. Имя функции.
2. Фактические аргументы.

Тип возвращаемого значения не указывается
Имя функции
Формальные аргументы заменяются фактическими аргументами

```
PrintMessage();
```

Рис. 1.9. Пример синтаксиса вызова функции

При выполнении функции формальные аргументы заменяются фактическими аргументами. Обратите внимание на синтаксис, и что в конце стоит точка с запятой. Пример функции с несколькими параметрами приведён в разделе 1.6.

Выполнение программы всегда начинается с функции `main()`. Выполняется тело функции `main()`. Компьютер встречает команду:

```
PrintMessage();
```

Это вызов функции, который приводит к выполнению тела функции `PrintMessage()`. Поэтому на экране будет напечатано `Getting Started`. Как упоминалось в начале этого раздела, функция `PrintMessage()` в теле функции `main()` скрывает свои действия – это и называется абстрактностью подпрограмм (раздел 1.6).