Глава 1

Краткий обзор C++

C++ — это расширенная версия языка C. C++ содержит в себе все, что имеется в C, но кроме этого он поддерживает объектно-ориентированное программирование (Object Oriented Programming, OOP). В C++ имеется множество дополнительных возможностей, которые независимо от объектно-ориентированного программирования делают его просто "лучше, чем C". За небольшими исключениями C++ — это более совершенный C. В то время как все, что вы знаете о языке C, вполне применимо и к C++, понимание его новых свойств все же потребует от вас значительных затрат времени и сил. Однако преимущества программирования на C++ с лихвой окупят ваши усилия.

Целью этой главы должно стать знакомство с некоторыми наиболее важными свойствами C++. Как вы знаете, элементы языка программирования не существуют в пустоте, изолированно от других. Они работают вместе в виде полной, законченной конструкции. В C++ эта взаимозависимость еще более ярко выражена, чем в C. В результате, трудно обсуждать любой аспект C++ без других его аспектов. Поэтому первая глава посвящена предварительному знакомству с теми свойствами C++, без которых сложно понять приводимые здесь примеры программ. Более подробно эти свойства будут изучаться в следующих главах.

Эта глава также освещает некоторые отличия между стилями программирования на языках C и C++. В C++ имеется несколько возможностей для написания более гибких, чем в C, программ. Хотя некоторые из этих возможностей имеют очень слабую связь с объектно-ориентированным программированием, или вообще ее не имеют, тем не менее, поскольку они содержатся в большинстве программ C++, стоит обсудить их в первую очередь.

Поскольку C++ был задуман для поддержки объектно-ориентированного программирования, эта глава начинается с описания OOP. Как вы увидите, многие свойства C++ тем или иным образом касаются OOP. Однако важно понимать, что C++ может использоваться для написания не только объектно-ориентированных программ. То, как вы используете C++, полностью зависит от вас.

К моменту написания этой книги процесс стандартизации языка программирования C++ был завершен. По этой причине здесь описываются некоторые важные отличия между обычными для последних нескольких лет версиями C++ и новым стандартом языка (Standard C++). Поскольку настоящая книга задумывалась как пособие для обучения языку Standard C++, этот материал особенно важен для тех, кто работает с устаревшим компилятором.

Помимо знакомства с некоторыми важными свойствами C++, в этой главе описываются существующие отличия между стилями программирования C и C++. Есть несколько аспектов C++, которые позволяют писать программы с большей гибкостью. Некоторые из этих аспектов C++ имеют очень незначительную связь с объектно-ориентированным программированием или вообще ее не имеют, но поскольку они встречаются в большинстве программ на C++, стоит обсудить их в начале книги.

Перед тем как начать собственно изложение материала, имеет смысл сделать несколько важных замечаний о природе и форме C++. Как правило, программы на C++ внешне напоминают программы на C. Так же, как и на C, программы на C++ начинают выполняться с функции **main().** Для получения аргументов командной строки C++ использует те же параметры **argc, argv,** что и C. Хотя C++ 'Определяет их в собственной объектно-ориентированной библиотеке, он также поддерживает все функции стандартной библиотеки C. В C++ используются те же управляющие структуры и те же встроенные типы данных, что и в C.

Запомните, в этой книге предполагается, что вы уже знаете язык программирования C. Проще говоря, вы уже должны уметь программировать на C перед; тем, как начнете изучать программирование на C++. Если вы еще не знаете C, то желательно потратить некоторое время на его изучение.

1.1. Что такое объектно-ориентированное программирование?

Объектно-ориентированное программирование — это новый подход к созданию программ. По мере развития вычислительной техники возникали разные методики программирования. На каждом этапе создавался новый подход, который помогал программистам справляться с растущим усложнением программ. Первые программы создавались посредством ключевых переключателей на передней панели компьютера. Очевидно, что такой способ подходит только для очень небольших программ. Затем был изобретен язык ассемблера, который позволял писать более длинные программы. Следующий шаг был сделан в 1950 году, когда был создан первый язык высокого уровня Фортран.

Используя язык высокого уровня, программисты могли писать программы до нескольких тысяч строк длиной. Для того времени указанный подход к программированию был наиболее перспективным. Однако язык программирования, легко понимаемый в коротких программах, когда дело касалось больших программ, становился нечитабельным (и неуправляемым). Избавление от таких неструктурированных программ пришло после изобретения в 1960 году *языков структурного программирования* (structured programming language). К ним относятся языки Алгол, Паскаль и C. Структурное программирование подразумевает точно обозначенные управляющие структуры, программные блоки, отсутствие (или, по крайней мере, минимальное использование) инструкций GOTO, автономные подпрограммы, в которых Поддерживается рекурсия и локальные переменные. Сутью структурного Программирования является возможность разбиения программы на составляющие ее элементы. Используя структурное программирование, средний программист может создавать и поддерживать программы свыше 50 000 строк длиной.

Хотя структурное программирование, при его использовании для написания умеренно сложных программ, принесло выдающиеся результаты, даже оно оказывалось несостоятельным тогда, когда программа достигала определенной длины. Чтобы написать более сложную программу, необходим был новый подход к программированию. В итоге были разработаны принципы объектно-ориентированного программирования. OOP аккумулирует лучшие идеи, воплощенные в структурном программировании, и сочетает их с мощными новыми концепциями, которые позволяют оптимально организовывать ваши программы. Объектно-ориентированное программирование позволяет вам разложить

проблему на составные части. Каждая составляющая становится самостоятельным объектом, содержащим свои собственные коды и данные, которые относятся к этому объекту. В этом случае вся процедура в целом упрощается, и программист получает возможность оперировать с гораздо большими по объему программами.

Все языки ООР, включая C++, основаны на трех основополагающих концепциях, называемых инкапсуляцией, полиморфизмом и наследованием. Рассмотрим эти концепции.

Инкапсуляция

Инкапсуляция (encapsulation) — это механизм, который объединяет данные и код, манипулирующий с этими данными, а также защищает и то, и другое от внешнего вмешательства или неправильного использования. В объектно-ориентированном программировании код и данные могут быть объединены вместе; в этом случае говорят, что создается так называемый "черный ящик". Когда коды и данные объединяются таким способом, создается *объект (object).* Другими словами, объект — это то, что поддерживает инкапсуляцию.

Внутри объекта коды и данные могут быть *закрытыми (private)* для этого объекта или *открытыми (public).* Закрытые коды или данные доступны только для других частей этого объекта. Таким образом, закрытые коды и данные недоступны для тех частей программы, которые существуют вне объекта. Если коды и данные являются открытыми, то, несмотря на то, что они заданы внутри объекта, они доступны и для других частей программы. Характерной является ситуация, когда открытая часть объекта используется для того, чтобы обеспечить контролируемый интерфейс закрытых элементов объекта.

На самом деле объект является переменной определенного пользователем, типа. Может показаться странным, что объект, который объединяет коды и данные, можно рассматривать как переменную. Однако применительно к объектно-ориентированному программированию это именно так. Каждый элемент данных такого типа является составной переменной.

Полиморфизм

Полиморфизм (polymorphism) (от греческого polymorphos) — это свойство, которое позволяет одно и то же имя использовать для решения двух или более схожих, но технически разных задач. Целью полиморфизма, применительно к объектно-ориентированному программированию, является использование одного имени для задания общих для класса действий. Выполнение каждого конкретного действия будет определяться типом данных. Например, для языка С, в котором полиморфизм поддерживается недостаточно, нахождение абсолютной величины числа требует трех различных функций: **abs(), Iabs() и fabsQ.** Эти функции подсчитывают и возвращают абсолютную величину целых, длинных целых и чисел с плавающей точкой соответственно. В C++ каждая из этих функций может быть названа **abs().** (Один из способов, который позволяет это делать, показан далее в этой главе.) Тип данных, который используется при вызове функции, определяет, какая конкретная версия функции действительно выполняется. Как вы увидите, в C++ можно использовать одно имя функции для множества различных действий. Это называется *перегрузкой функций (function overloading).*

В более общем смысле, концепцией полиморфизма является идея "один интерфейс, множество методов". Это означает, что можно создать общий интерфейс для группы близких по смыслу действий. Преимуществом полиморфизма является то, что он помогает снижать сложность программ, разрешая использование того же интерфейса для задания *единого класса действий.* Выбор же *конкретного действия,* в зависимости от ситуации,

возлагается на компилятор. Вам, как программисту, не нужно делать этот выбор самому. Нужно только помнить и использовать общий интерфейс. Пример из предыдущего абзаца показывает, как, имея три имени для функции определения абсолютной величины числа вместо одного, обычная задача становится более сложной, чем это действительно необходимо.

Полиморфизм может применяться также и к операторам. Фактически во всех языках программирования ограниченно применяется полиморфизм, например, в арифметических операторах. Так, в С, символ + используется для складывания целых, длинных целых, символьных переменных и чисел с плавающей точкой. В этом случае компилятор автоматически определяет, какой тип арифметики требуется. В С++ вы можете применить эту концепцию и к другим, заданным вами, типам данных. Такой тип полиморфизма называется *перегрузкой операторов (operator overloading).*

Ключевым в понимании полиморфизма является то, что он позволяет вам манипулировать объектами различной степени сложности путем создания общего для них стандартного интерфейса для реализации похожих действий.

Наследование

Наследование (inheritance) — это процесс, посредством которого один объект может приобретать свойства другого. Точнее, объект может наследовать основные свойства другого объекта и добавлять к ним черты, характерные только для него. Наследование является важным, поскольку оно позволяет поддерживать концепцию *иерархии классов (hierarchical classification).* Применение иерархии классов делает управляемыми большие потоки информации. Например, подумайте об описании жилого дома. Дом — это часть общего класса, называемого **строением. С** другой стороны, **строение** — это часть более общего класса — **конструкции,** который является частью еще более общего класса объектов, который можно назвать **созданием рук Человека. В** каждом случае порожденный класс наследует все, связанные с родителем, качества и добавляет к ним свои собственные определяющие характеристики. Без использования иерархии классов, для каждого объекта пришлось бы задать все характеристики, которые бы исчерпывающе его определяли. Однако при использовании наследования можно описать объект путем определения того общего класса (или классов), к которому он относится, с теми специальными чертами, которые делают объект уникальным. Как вы увидите, наследование играет очень важную роль в ООР.

Примеры

1. Инкапсуляция не является исключительной прерогативой ООР. Некоторая степень инкапсуляция может быть достигнута и в языке С. Например, при применении библиотечной функции, в конечном итоге, имеет место концепция черного ящика, содержимое которого вы не можете изменить (исключая возможно, злой умысел). Рассмотрим функцию **fopen().** Если она используется для открытия файла, то создаются и инициализируются несколько внутренних переменных. В той мере, в которой это касается вашей программы, эти переменные скрыты и недоступны. Но, конечно, С++ обеспечивает более надежную поддержку инкапсуляции.)

2. В реальной жизни примеры полиморфизма вполне обычны. Например, рассмотрим рулевое колесо автомобиля. Оно работает одинаково, независимо от того, используется ли при этом электропривод, механический привод или стандартное ручное управление. Ключевым является то, что интерфейс (рулевое колесо) один и тот же, независимо от того, какой рулевой механизм (метод) применяется на самом деле.

3. Наследование свойств и базовая концепция классов являются основополагающими для организации пути познания. Например, **сельдерей** — член класса **овощей,** которые являются частью класса **растений.** В свою очередь, растения являются живыми организмами и так далее. Без иерархии классов систематизация знаний была бы невозможна.

1.2 Две версии C++

При написании этой книги C++ находился на перепутье. Как указано в предисловии, в последние годы активно шел процесс стандартизации C++. Целью стандартизации было создание стабильного, ориентированного на будущее языка, которой призван удовлетворить потребности программистов следующего столетия. Результатом стало параллельное существование двух версий C++. Первая, традиционная версия базируется на исходной разработке Бьярна Страуструпа. Это та версия, которая использовалась программистами последние лет десять. Вторая версия, названная Standard C++, создана Бьярном Страуструпом совместно с комитетом по стандартизации (ANSI — American National Standards Institute, Американский национальный институт стандартов; ISO — International Standards Organization, Международная организация по стандартам). Хотя по сути эти две версии очень похожи, Standard C++ содержит несколько усовершенствований, которых нет в традиционном C++. Таким образом Standard C++ по существу является надмножеством традиционного C++.

Настоящая книга учит языку Standard C++. Эта версия C++, определенная комитетом по стандартизации ANSI/ISO, должна быть реализована во всех современных компиляторах C++. Примеры программ этой книги отражают современный стиль программирования в соответствии с новыми реалиями языка Standard C++. Это означает актуальность содержания книги не только сегодня, но и в будущем. Проще говоря, Standard C++ — это будущее. А поскольку Standard C++ содержит в себе черты всех ранних версий C++, то, что вы узнаете из этой книги, позволит вам работать в любой программной среде C++.

Тем не менее, если вы используете устаревший компилятор, не все приведенные в книге программы будут вам доступны, поскольку в процессе стандартизации комитет ANSI/ISO добавил к языку массу новых черт. По мере определения новых черт, они реализовывались производителями компиляторов. Естественно, что между добавлением новой возможности языка и ее доступностью в коммерческих компиляторах всегда есть определенный промежуток времени, а поскольку такие возможности добавлялись в течение нескольких лет, старые компиляторы могут не поддерживать некоторые из них. Это важно, так как два недавних добавления к языку C++ имеют отношение ко всем, даже простейшим программам. Если у вас устаревший компилятор, в котором указанные новые возможности недоступны, не расстраивайтесь, в следующих разделах описано несколько простых способов обойти проблему.

Отличия между прежним и современным стилями программирования в числе прочих включают две новые черты: изменился стиль оформления заголовков (headers) и

появилась инструкция **namespace.** Чтобы продемонстрировать эти отличия, начнем с рассмотрения двух версий простейшей программы на C++. Первая показанная здесь версия написана в прежнем, еще совсем недавно основном стиле программирования.

```
/*
Программа на C++ в традиционном стиле
*/
#include <iostream.h>
int main ()
{
/ *  программный  код
return 0  ;
}
```

Поскольку C++ строится на C, этот каркас программы должен быть хорошо вам знаком, тем не менее обратите особое внимание на инструкцию # in elude Эта инструкция подключает к программе заголовочный файл **iostream.h,** который обеспечивает поддержку системы ввода/вывода C++. (В C++ этот файл имеет то же самое назначение, что и файл **stdio.h** в C.**)**

Ниже представлена вторая версия программы, в которой используется современный стиль:

```
/*
     Программа на C++ в современном стиле. Здесь используются новое оформление заголовков и ключевое слово namespace
*/
#include <iostream
using namespace std;
int main()
{
/* программный код */
return 0;
}
```

Обратите внимание на две строки в самом начале программы, в которых имеют место изменения. Во-первых, в инструкции **#include** после слова **iostream** отсутствуют символы **.h**. Во-вторых, в следующей строке задается так называемое *пространство имен* (namespace). Хотя подробно эти нововведения будут рассмотрены позднее, сейчас дадим их краткий обзор.

Новые заголовки в программах на C++

Как вам должно быть известно из опыта программирования на C, при использовании библиотечной функции в программу необходимо включить заголовочный файл. Это делается с помощью инструкции **#include**. Например, при написании программ на языке C заголовочным файлом для функций ввода/вывода является файл **stdio.h**, который включается в программу с помощью следующей инструкции:

```
#include <stdio.h>
```

Здесь **stdio.h** — это имя файла, который используется функциями ввода/вывода, и предыдущая инструкция заставляет компилятор *включить* указанный файл в вашу программу.

В первые несколько лет после появления C++ в нем использовался тот же стиль оформления заголовков, что и в C. Для совместимости с прежними программами в языке Standard C++ этот стиль по-прежнему поддерживается. Тем не менее при работе с библиотекой Standard C++ в соответствии с новым стилем вместо имен заголовочных файлов указываются стандартные идентификаторы, по которым компилятор находит требуемые файлы. Новые заголовки C++ являются абстракциями, гарантирующими объявление соответствующих прототипов и определений библиотеки языка Standard C++.

Поскольку новые заголовки не являются именами файлов, для них не нужно указывать расширение **.h,** а только имя заголовка в угловых скобках. Ниже представлены несколько заголовков, поддерживаемых в языке Standard C++:

```
<iostream>
<fstream>
<vector>
<string>
```

Такие заголовки по-прежнему включаются в программу с помощью инструкции **#include**. Единственным отличием является то, что новые заголовки совершенно не обязательно являются именами файлов.

Поскольку C++ содержит всю библиотеку функций C, по-прежнему поддерживается стандартный стиль оформления заголовочных файлов библиотеки C. Таким образом, такие заголовочные файлы, как **stdio.h** и **ctype.h** все еще доступны. Однако Standard C++ также определяет заголовки нового стиля, которые можно указывать вместо этих заголовочных файлов. В соответствии с версией C++ к стандартным заголовкам C просто добавляется префикс c и удаляется расширение **.h**. Например, заголовок **math.h** заменяется новым заголовком C++ **<cmath>,** а заголовок **siring.h** — заголовком **<cstring>.** Хотя в настоящее время при работе с функциями библиотеки C допускается включать в программы заголовочные файлы в соответствии со стилем C, такой подход не одобряется стандартом языка Standard C++. (То есть, он не рекомендуется.) По этой причине во всех имеющихся в книге инструкциях **#include** используется новый стиль написания заголовков программ. Если ваш компилятор для функций библиотеки C не поддерживает заголовки нового стиля, просто замените их заголовками в стиле C.

Поскольку заголовки нового стиля появились в C++ совсем недавно, во многих и многих прежних программах вы их не найдете. В этих программах в соответствии со стилем C в заголовках указаны имена файлов. Ниже представлен традиционный способ включения в программу заголовка для функций ввода/вывода:

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

Эта инструкция заставляет компилятор включить в программу заголовочный файл **iostream.h**. Как правило, в заголовках прежнего стиля вместе с расширением **.h** используется то же имя, что и в соответствующих им новых заголовках.

Как уже отмечалось, все компиляторы C++ поддерживают заголовки старого стиля. Тем не менее такие заголовки объявлены устаревшими и не рекомендуются. Именно поэтому в

книге вы их больше не встретите.

Пространства имен

Когда вы включаете в программу заголовок нового стиля, содержание этого заголовка оказывается в пространстве имен **std.** *Пространство имен* (namespace) — это просто некая объявляемая область, необходимая для того, чтобы избежать конфликтов имен идентификаторов. Традиционно имена библиотечных функций и других подобных идентификаторов располагались в глобальном пространстве имен (как, например, в С). Однако содержание заголовков нового стиля помещается в пространстве имен **std.** Позднее мы рассмотрим пространства имен более подробно. Сейчас же, чтобы пространство имен **std** стало видимым, просто используйте следующую инструкцию:

using namespace std;

Эта инструкция помещает std в глобальное пространство имен. После того как компилятор обработает эту инструкцию, вы сможете работать с заголовками как старого, так и нового стиля.

Если вы работаете со старым компилятором

Как уже упоминалось, заголовки нового стиля и пространства имен появились в С++ совсем недавно, поэтому эти черты языка могут не поддерживаться старыми компиляторами. Если у вас один из таких компиляторов, то при попытке компиляции первых двух строк кода, приводимых в книге примеров программ, вы получите одно или несколько сообщений об ошибках. Обойти эту проблему просто — удалите инструкцию **namespace** и используйте заголовки старого стиля. То есть замените, например, инструкции

```
#include <iostream>
using namespace std;
```

на инструкцию

```
#include <iostream,h>
```

Это простое действие превратит современную программу в такую же, но в традиционном стиле. Поскольку заголовок старого стиля считывает все свое содержание в глобальное пространство имен, необходимость в инструкции **namespace** отпадает.

И еще одно замечание. Еще в течение нескольких лет вы будете встречать программы, в которых заголовки будут оформлены в старом стиле и не будет инструкций **namespace.** Ваш компилятор С++ будет прекрасно справляться с такими программами. Тем не менее, что касается новых программ, вам следует использовать современный стиль, поскольку именно он определен стандартом языка Standard С++. Хотя программы прежнего стиля будут поддерживаться еще многие годы, технически они некорректны.

1.3 Консольный ввод и вывод в С++

Поскольку С++ — это улучшенный С, все элементы языка С содержатся также и в С++. Это подразумевает, что все программы, написанные на С, по умолчанию являются также и программами на С++. (На самом деле имеется несколько очень незначительных исключений из этого правила, которые будут рассмотрены позже.) Поэтому можно писать программы на С++, которые будут

выглядеть точно так же, как и программы на С. Ошибки не будет, это только будет означать, что вы не смогли использовать все преимущества С++. Чтобы по достоинству оценить С++, необходимо писать программы в стиле С++.

Вероятно, наиболее своеобразной чертой языка С++, используемой программистами, является подход к вводу и выводу. Хотя такие функции, как **printf() и scanf(),** по-прежнему доступны, С++ обеспечивает иной, лучший способ выполнения этих операций. В С++ ввод/вывод выполняется с использованием *операторов,* а не функций ввода/вывода. Оператор вывода — это «, а оператор ввода — ». Как вы знаете, в С эти операторы являются, соответственно, операторами левого и правого сдвига. В С++ они сохраняют свое первоначальное значение (левый и правый сдвиг), выполняя при этом еще ввод и вывод. Рассмотрим следующую инструкцию С++:

```
cout « "Эта строка выводится на экран.\ n " ;
```

Эта инструкция осуществляет вывод строки в заранее определенный поток **cout,** который автоматически связывается с терминалом, когда программа С++ начинает выполняться. Это напоминает действие функции **stdout** в языке С. Как и в С, терминал для ввода/вывода в С++ может быть переопределен, но пока будем считать, что используется экран.

С помощью оператора вывода « можно вывести данные любого базового типа С++. Например, следующая инструкция осуществляет вывод величины 100.99:

```
cout «  100.99;
```

В общем случае, для вывода на экран терминала используется следующая обычная форма оператора «:

cout **<<** ***выражение;***

Здесь ***выражение*** может быть любым действительным выражением С++, включая другие выражения вывода.

Для считывания значения с клавиатуры, используйте оператор ввода >>. Например, в этом фрагменте целая величина вводится в num:

```
int num;
cin >> num;
```

Обратите внимание, что переменной **num** *не* предшествует амперсанд **&**. Как вы знаете, при вводе с использованием функции **scanf()** языка С ей должны передаваться адреса переменных Только тогда они смогут получить значения, вводимые пользователем. В случае использования оператора ввода С++ все происходит иначе. (Смысл этого станет ясен после того, как вы больше узнаете о С++.)

В общем случае для ввода значения с клавиатуры, используйте следующую форму оператора »:

cin >> ***переменная***;

Примеры

1. В этой программе выводится строка, два целых числа и одно число с плавающей точкой двойной точности:

```
#include <iostream>
```

```
using namespace std;
int main()
{
   int i, j;
   double d;
   i = 10;
   j = 20;
   d = 99.101;

   cout << "Вот несколько чисел: ";
   cout << i;
   cout << ' ';
   cout << j;
   cout << ' ';
   cout << d;

   return 0;
}
```

2. В одном выражении ввода/вывода можно выводить более одной величины. Например, версия программы, описанной в примере 1, показывает один из эффективных способов программирования инструкций ввода/вывода.

```
#include <iostream>
using namespace std;

int main()
{
   int i, j;
   double d;

   i = 10;
   j = 20;
   d = 99.101;

   cout << "Вот несколько чисел: ";
   cout << i << ' ' << j << ' ' << d;

   return 0;
}
```

Здесь в строке

```
cout << i << ' ' << j << ' ' << d;
```

выводится несколько элементов данных в одном выражении. В общем случае вы можете использовать единственную инструкцию для вывода любого требуемого количества элементов данных. Если это кажется неудобным, просто запомните, что оператор вывода **<<** ведет себя так же, как и любой другой оператор C++, и может быть частью произвольно длинного выражения.

Обратите внимание, что по мере необходимости следует включать в программу пробелы между элементами данных. Если пробелов не будет, то данные, выводимые на экран, будет неудобно читать.

3. Это программа предлагает пользователю ввести целое число:

```
#include <iostream>
using namespace std;
```

```
int main()
{
   int i;

   cout << "Введите число: ";
   cin >> i;
   cout << "Вот ваше число: " << i << "\n";

   return 0;
}
```

Результат работы программы:

```
Введите число: 100
Вот ваше число: 100
```

Как видите, введенное пользователем значение действительно оказывается в **i.**

4. Следующая программа — это программа ввода целого, числа с плавающей точкой и строки символов. В ней для ввода всего перечисленного используется одна инструкция.

```
#include <iostream>
using namespace std;

int main()
{
   int i;
   float f;
   char s[80];

   cout << "Введите целое, число с плавающей точкой и строку: ";
   cin >> i >> f >> s;
   cout << "Вот ваши данные: ";
   cout << i << ' ' << f << ' ' << s;

   return 0;
}
```

Как видно из этого примера, можно ввести в одной инструкции ввода столько элементов данных, сколько нужно. Как и в C, различные элементы данных при вводе должны быть отделены друг от друга (пробелами, табуляциями или символами новой строки).

При считывании строки ввод будет остановлен после считывания первого разделительного символа. Например, если вы введете:

```
10 100.12  Это проверка
```

то на экран будет выведено:

```
10   100.12  Это
```

Так происходит потому, что считывание строки прекращается при вводе пробела после слова **Это**. Остаток строки остается в буфере ввода, в ожидании следующей операции ввода. (Это похоже на ввод строки с использованием функции **scanf()** в формате **%s**.)

5. По умолчанию при использовании оператора >> буферизуется весь ввод строки. Это означает, что до тех пор, пока вы не нажмете клавишу <Enter>, информация не будет

передана в вашу программу. (В языке С функция **scanf()** также буферизует ввод строки, поэтому такой стиль ввода не должен быть для вас чем-то новым.) Для исследования построчно-буферизованного ввода рассмотрим следующую программу:

```
#include <iostream>
using namespace std;

int main()
{
  char ch;

  cout << "Вводите символы, для окончания ввода введите x. \n";

  do {
    cout << ":";
    cin >> ch;
  } while (ch !='x');

  return 0;
}
```

Когда вы протестируете эту программу, то убедитесь, что для считывания каждого очередного символа необходимо нажимать клавишу <Enter>.

1 . 4 Комментарии в С++

В С++ комментарии в программу можно включать двумя различными способами. Первый способ — это использование стандартного механизма, такого же, как в С, т. е. комментарий начинается с **/*** и оканчивается ***/**. Как и в С, в С++ этот тип комментария не может быть вложенным.

Вторым способом, которым вы можете писать комментарии в программах С++, является *однострочный комментарий.* Однострочный комментарий начинается с символов // и заканчивается концом строки. Другого символа. помимо физического конца строки (такого, как возврат каретки/перевод строки), в однострочном комментарии не используется.

Примеры

1. Программа, в которой есть стили комментариев как С, так и С++:

```
/* Этот комментарий в стиле С. Данная программа определяет четность целого */
#include <iostream> using namespace std;
 int main () {
      int num; // это однострочный комментарий С++
      // чтение числа
      cout « "Введите проверяемое число:";
      cin » nurn;
      // проверка на четность
      if((num%2)==0) cout « "Число четное\п";
      else cout « "Число нечетное\п";
      return 0;
      }
```

2. Хотя многострочные комментарии не могут быть вложенными, однострочный комментарий в стиле C++ можно вкладывать внутрь многострочного комментария. Например, это совершенно правильный фрагмент:

```
/*  Это многострочный комментарий,
внутрь  которого  //  вложен однострочный  комментарий.
Это окончание многострочного комментария.
*/
```

Тот факт, что однострочный комментарий может быть вложен в многострочный, дает возможность при отладке "помечать" некоторые строки программы.

1.5. Классы. Первое знакомство

Вероятно, одним из наиболее важных понятий C++ является класс. Класс — это механизм для создания объектов. В этом смысле класс лежит в основе многих свойств C++. Хотя более детально понятие класса раскрывается в следующих главах, оно столь фундаментально для программирования на C++, что краткий обзор здесь необходим.

Класс объявляется с помощью ключевого слова **class.** Синтаксис объявления класса похож на синтаксис объявления структуры. Здесь показана основная
форма:

class ***имя_класса*** {
    закрытые функции и переменные **класса**
public :
    открытие функции и **переменные класса**
} ***список_объектов;***

В объявлении класса ***список _объектов*** не обязателен. Как и в случае со структурой, вы можете объявлять объекты класса позже, по мере необходимости. Хотя ***имя_класса*** также не обязательно, с точки зрения практики оно необходимо. Доводом в пользу этого является: то, что ***имя_класса*** становится именем нового типа данных, которое используется для объявления объектов класса.

Функции и переменные, объявленные внутри объявления класса, становятся, как говорят, *членами (members)* этого класса. По умолчанию все функции и переменные, объявленные в классе, становятся закрытыми для класса. Это означает, что они доступны только для других членов того же класса. Для объявления открытых членов класса используется ключевое слово **public,** за которым следует двоеточие. Все функции и переменные, объявленные после слова **public,** доступны как для других членов класса, так и для любой другой части программы, в которой находится этот класс.

Ниже приводится простое объявление класса:

```
class myclass {
    // закрытый элемент класса
    int a;
 public:
    void set_a(int num);
    int get_a ( ) ;
};
```

Этот класс имеет одну закрытую переменную a, и две открытые функции, **set_a() и get_a().**

Обратите внимание, что прототипы функций объявляются внутри класса. Функции, которые объявляются внутри класса, называются *функциями-членами (member functions).*

Поскольку a является закрытой переменной класса, она недоступна для любой функции вне **myclass.** Однако поскольку **set_a() и get_a()** являются членами **myclass,** они имеют доступ к a. Более того, **set_a() и get_a(),** являясь открытыми членами **myclass,** могут вызываться из любой части программы, использующей **myclass.**

Хотя функции **set_a() и get_a()** и объявлены в **myclass,** они еще не определены. Для определения функции-члена вы должны связать имя класса, частью которого является функция-член, с именем функции. Это достигается путем написания имени функции вслед за именем класса с двумя двоеточиями. Два двоеточия называются *оператором расширения области видимости (scope resolution operator).* Например, далее показан способ определения функций-членов **set_a() и get_a():**

```
void  myclass::set_a(int  num)
{
        a=num;
}
int myclass::get_a()
{
        return a;
}
```

Отметим, что и **set_a() и get_a()** имеют доступ к переменной a, которая для **myclass** является закрытой. Как уже говорилось, поскольку **set_a() и get_a()** являются членами **myclass,** они могут напрямую оперировать с его закрытыми .чанными.

При определении функции-члена пользуйтесь следующей основной формой:

Тип_возвр_значения имя_класса::имя функции(список параметров)
. . .// -тело функции

Здесь ***имя_класса*** — это имя того класса, которому принадлежит определяемая функция.

Объявление класса **myclass** не задает ни одного объекта типа **myclass,** оно определяет только тип объекта, который будет создан при его фактическом объявлении. Чтобы создать объект, используйте имя класса, как спецификатор типа данных. Например, в этой строке объявляются два объекта типа **myclass:**

myclass ob1, ob2; // это объекты типа myclass

Примеры

1. В качестве первого простого примера, рассмотрим программу, в которой используется

myclass, описанный в тексте, для задания значений a для ob1 и ob2 и вывода на экран этих значений для каждого объекта:

```
#include <iostream>
using namespace std;

class myclass {
   // закрытая часть myclass
   int a;
public:
   void set_a(int num);
   int get_a();
};

void myclass::set_a(int num)
{
   a=num;
}

int myclass::get_a()
{
   return a;
}

int main()
{
   myclass ob1, ob2;

   ob1.set_a(10);
   ob2.set_a(99);

   cout << ob1.get_a() << "\n";
   cout << ob2.get_a() << "\n";

   return 0;
}
```

Как и следовало ожидать, программа выводит на экран величины 10 и 99.

2. В предыдущем примере переменная **a** в **myclass** является закрытой. Это означает, что она непосредственно доступна только для членов **myclass.** (Это один из доводов в пользу существования открытой функции **get_a().)** Если вы попытаетесь обратиться к закрытому члену класса из той части вашей программы, которая не является членом этого класса, то результатом будет ошибка при компиляции. Например, предположим, что **myclass** задан так, как показано в предыдущем примере, тогда компиляция функции **main()** вызовет ошибку:

```
// Этот фрагмент содержит ошибку.
#include <iostream>
using namespace std;
int main()
{
   myclass ob1, ob2;
   ob1.a = 10;      // ОШИБКА! к закрытому члену нет
   ob2.a = 99;      // доступа для функции - не члена
   cout << ob1.get_a() << "\n";
   cout << ob2.get_a() << "\n";
   return 0;
}
```

3. Точно так же, как открытые функции-члены, могут существовать и открытые переменные-члены. Например, если бы a была объявлена в открытой секции **myclass,** тогда к ней, как показано ниже, можно было бы обратиться из любой части программы:

```
#include <iostream>
using namespace std;

class myclass {
public:
   // теперь a открыта
   int a;
   // и здесь не нужны функции set_a() и get_a()
};

int main()
{
   myclass ob1, ob2;

   // здесь есть явный доступ к a
   ob1.a = 10;
   ob2.a = 99;

   cout << ob1.a << "\n";
   cout << ob2.a << "\n";

   return 0;
}
```

В этом примере, поскольку a объявлена открытым членом **myclass,** к ней имеется явный доступ из **main().** Обратите внимание, как оператор точка (.) используется для доступа к a. Обычно, когда вы вызываете функцию-член, или осуществляете доступ к переменной-члену не из класса, которому они принадлежат, за именем объекта должен следовать оператор точка **(.)**, а за ним имя члена. Это необходимо для исчерпывающего определения того, с членом какого объекта вы имеете дело.

4. Чтобы по достоинству оценить возможности объектов, рассмотрим более практичный пример. В этой программе создается класс **stack,** реализующий стек, который можно использовать для хранения символов:

```
#include <iostream>
using namespace std;

#define SIZE 10

// Объявление класса stack для символов
class stack {
   char stck[SIZE]; // содержит стек
   int tos; // индекс вершины стека

public:
   void init(); // инициализация стека
   void push(char ch); // помещает в стек символ
   char pop(); // выталкивает из стека символ
};

// Инициализация стека
void stack::init()
{
   tos=0;
}
```

```
// Помещение символа в стек
void stack::push(char ch)
{
   if (tos==SIZE) {
      cout << "Стек полон";
      return;
   }
   stck[tos] = ch;
   tos++;
}

// Выталкивание символа из стека
char stack::pop()
{
   if (tos==0) {
      cout << "Стек пуст";
      return 0; // возврат нуля при пустом стеке
   }
   tos--;
   return stck[tos];
}

int main()
{
   stack s1, s2; // создание двух стеков
   int i;
   // инициализация стеков
   s1.init();
   s2.init();

   s1.push('a');
   s2.push('x');
   s1.push('b');
   s2.push('y');
   s1.push('c');
   s2.push('z');

   for(i=0;i<3;i++) cout << "символ из s1:" << s1.pop() << "\n";
   for(i=0;i<3;i++) cout << "символ из s2:" << s2.pop() << "\n";

   return 0;
}
```

Эта программа выводит на экран следующее:

```
символ из s1:    c
символ из s1:    b
символ из s1:    a
символ из s2:    z
символ из s2:    y
символ из з2:    x
```

Давайте теперь детально проанализируем программу. Класс **stack** содержит две закрытые переменные: **stck** и **tos**. Массив **stck** содержит символы, фактически помещаемые в стек, а tos содержит индекс вершины стека. Открытыми функциями стека являются **init(), push()** и pop(), которые, соответственно, инициализируют стек, помещают символ в стек и выталкивают его из стека. Внутри функции main() создаются два стека, **s1** и **s2**, и по три символа помещаются в каждый из них. Важно понимать, что один объект (стек) не зависит от другого. Поэтому у символов в **s1** нет способа влиять на символы в **s2**. Каждый объект содержит свою собственную копию **stck** и **tos**. Это фундаментальная для понимания объектов концепция. Хотя все объекты класса имеют общие функции-члены, каждый объект создает и поддерживает *свои собственные данные*.

1.6. Некоторые отличия языков C и C++

У языка C++ есть ряд небольших отличий от C. Хотя каждое из этих отличий само по себе незначительно, вместе они достаточно распространены в программах C++. Поэтому перед тем как двинуться дальше, обсудим эти отличия.

- Во-первых, если в C функция не имеет параметров, ее прототип содержит слово void в списке параметров функции. Например, если в C функция **f1**() не имеет параметров (и возвращает **char),** ее прототип будет выглядеть следующим образом:

  ```
  char   fI(void);
  ```

  В C++ слово **void** не обязательно. Поэтому в C++ прототип обычно пишется так:

  ```
  char  f1() ;
  ```

  C++ отличается от C способом задания пустого списка параметров. Если бы предыдущий прототип имел место в программе C, то это бы означало, что о параметрах функции сказать *ничего нельзя.* А в C++ это означает, что у функции *нет* параметров. Поэтому в предыдущих примерах для исчерпывающего обозначения пустого списка параметров слово **void** не использовалось. (Использование **void** для обозначения пустого списка параметров не ошибочно, скорее, оно излишне. Поскольку большинство программистов C++ гонятся за эффективностью с почти религиозным рвением, вы никогда не увидите **void** в таких случаях.) Запомните, в C++ следующие два объявления эквивалентны:

  ```
  int  f1();
  int   f1(void);
  ```

- Другим небольшим отличием между C и C++ является то, что в программах C++ все функции должны иметь прототипы. Запомните, в C прототипы функций рекомендуются, но технически они не обязательны, а в C++ прототипы необходимы. Как показывают примеры из предыдущего раздела, содержащийся в классе прототип функции-члена действует так же, как ее обычный прототип, и никакого иного прототипа не требуется.

- Третьим отличием между C и C++ является то, что если в C++ функция имеет отличный от **void** тип возвращаемого значения, то инструкция **return** внутри этой функции должна содержать значение данного типа. В языке C функции с отличным от **void** типом возвращаемого значения фактически не требуется возвращать что-либо. Если значения нет, то функция возвращает неопределенное значение.

  В C, если тип возвращаемого функцией значения явно не задан, функция по умолчанию возвращает значение целого типа. В C++ такого правила нет. Следовательно, необходимо явно объявлять тип возвращаемого значения всех функций.

Следующим отличием между C и C++ является то, что в программах C++ вы можете выбирать место для объявления локальных переменных. В C локальные переменные могут объявляться только в начале блока, перед любой инструкцией "действия". В C++ локальные переменные могут объявляться в любом месте программы. Одним из

преимуществ такого подхода является то, что локальные переменные для предотвращения нежелательных побочных эффектов можно объявлять рядом с местом их первого использования.

- И последнее. Для хранения значений булева типа (истина или ложь) в C++ определен тип данных **bool. В** C++ также определены ключевые слова **true и false** — единственные значения, которыми могут быть данные типа **bool.** В C++ результатом выполнения операторов отношения и логических операторов являются значения типа **bool,** и направление развития любой условной инструкции должно определяться относительно значения типа **bool.** Хотя такое отличие от C на первый взгляд кажется значительным, на самом деле это не так. Фактически оно совершенно прозрачно и вот почему: как вы знаете, в C любое ненулевое значение является истинным, а нулевое — ложным. В C++ это положение сохраняется, поскольку при использовании таких значений в булевом выражении ненулевое значение автоматически преобразуется в **true,** а нулевое — **в false.** Правильно и обратное: **true** преобразуется в 1, а **false** в 0, если значение типа **bool** оказывается в целом выражении. Добавление в C++ данных типа **bool** усиливает контроль типа и дает возможность различать данные булева и целого типов. Естественно, что использование данных булева типа не обязательно, скорее оно просто удобно.

Примеры

1. В программах C, при отсутствии в командной строке аргументов, функция **main()** обычно объявляется так:

   int main(void)

   Однако в C++ такое использование слова **void** избыточно и необязательно.

2. Эта короткая программа C++ не будет компилироваться, поскольку у функции **sum()** нет прототипа:

```
// Эта программа не будет компилироваться
#include <iostream>
using namespace std;

int main()
{
  int a,b,c;

  cout << "Введите два числа: ";
  cin >> a >> b;
  c=sum(a, b);
  cout << "Сумма равна:" << c;

  return 0;
```

```
}

// Этой функции необходим прототип
sum(int a, int b)
{
  return a+b;
}
```

3. Эта короткая программа иллюстрирует тот факт, что локальные переменные можно объявить в любом месте блока:

```
#include <iostream>
using namespace std;

int main()
{
  int i; // локальная переменная, объявленная в начале блока

  cout << "Введите число:";
  cin >> i;

  // расчет факториала
  int j, fact=1; // переменные, объявленные перед инструкциями
          // действия

  for (j=i; j>=1; j--) fact=fact * j;

  cout << "Факториал равен:" << fact;
  return 0;
}
```

Хотя объявление переменных **j** и **fact** рядом с местом их первого использования в этом коротком примере и не слишком впечатляет, в больших функциях такая возможность может обеспечить программе ясность и предотвратить нежелательные побочные эффекты.

4. В следующей программе создается булева переменная **outcome** и ей присваивается значение **false.** Затем эта переменная используется в инструкции **if.**

```
#include <iostream>
using namespace std;

int main()
{
  bool outcome;

  outcome = false;

  if(outcome) cout << "истина"
  else cout << "ложь";

  return 0;
}
```

Как и следовало ожидать, в результате выполнения программы на экране появляется слово **ложь.**

1 . 7 . Введение в перегрузку функций

C++ является *перегрузка функций (function overloading)*. Перегрузка функций не только обеспечивает механизм, посредством которого в C++ достигается один из типов полиморфизма, она также формирует то ядро, вокруг которого развивается вся среда программирования на C++. Ввиду важности темы в данном разделе предлагается только предварительное знакомство с перегрузкой функций, которой посвящена целая глава этой книги.

В C++ две или более функции могут иметь одно и то же имя, отличаясь либо типом, либо числом своих аргументов, либо и тем и другим. Если две или более функции имеют одинаковое имя, говорят, что они *перегружены.*

Перегруженные функции позволяют упростить программы, допуская обращение к одному имени для выполнения близких по смыслу действий.

Перегрузить функцию очень легко: просто объявите и определите все требуемые варианты. Компилятор автоматически выберет правильный вариант вызова на основании числа и/или типа используемых в функции аргументов.

Примепы

1. Одно из основных применений перегрузки функций — это достижение полиморфизма при компиляции программ, который воплощает в себе философию — один интерфейс, множество методов. Как вы знаете, при программировании на С необходимо иметь определенное число близких по назначению функций, отличающихся только типом данных, с которыми они работают. Классический пример этой ситуации дает набор библиотечных функций С. Как ранее упоминалось в этой главе, библиотека содержит функции **abs(), Iabs() и fabs(),** которые возвращают абсолютное значение, соответственно, целого, длинного целого и числа с плавающей точкой. Однако из-за того, что для трех типов данных требуется три типа функции, ситуация выглядит более сложной, чем это необходимо. Во всех трех случаях возвращается абсолютная величина числа, отличие только в типе данных. В то же время, программируя на C++, вы можете исправить эту ситуацию путем перегрузки одного имени для трех типов данных так, как показано в следующем примере:

```
#include <iostream>
using namespace std;

// Перегрузка abs() тремя способами
int abs(int n);
long abs(long n);
double abs(double n);

main()
{
   cout << "Абсолютная величина -10:" << abs(-10) << "\n";
   cout << "Абсолютная величина -10L:" << abs(-10L) << "\n";
   cout << "Абсолютная величина -10.01:" << abs(-10.01) << "\n";

   return 0;
}
```

```
// abs() для целых
int abs(int n)
{
  cout << "В целом abs()\n";
  return n<0 ? -n : n;
}

// abs() для длинных целых
long abs(long n)
{
  cout << "В длинном целом abs()\n";
  return n<0 ? -n : n;
}

// abs() для вещественных двойной точности
double abs(double n)
{
  cout << "В вещественном abs() двойной точности\n";
  return n<0 ? -n : n;
}
```

Как можно заметить, в программе задано три функции **abs(),** своя для каждого типа данных. Внутри **main()** функция **abs()** вызывается с тремя аргументами разных типов. Компилятор автоматически вызывает правильную версию **abs(),** основываясь на используемом в аргументе типе данных. В результате работы программы на экран выводится следующее:

```
В целом abs ( )
Абсолютная величина -10: 10

В длинном целом abs ( )

Абсолютная величина -10L: 10

В вещественном abs ( ) двойной точности

Абсолютная величина -10 . 01 : 10.01
```

Хотя этот пример достаточно прост, ценность перегрузки функций он все же демонстрирует. Поскольку одно имя используется для описания основного набора действий, искусственная сложность, вызванная тремя слабо различающимися именами, в данном случае **abs(), labs() и fabs(),** устраняется. Теперь вам необходимо помнить только одно имя — то, которое описывает *общее* действие. На компилятор возлагается задача выбора соответствующей *конкретной* версии вызываемой функции (а значит и метода обработки данных). Это имеет лавинообразный эффект в вопросе снижения сложности программ. В данном случае, благодаря использованию полиморфизма, из трех имен получилось одно.

Хотя использование полиморфизма в этом примере довольно тривиально, вы, должно быть, уже поняли, что для очень больших программ подход "один интерфейс, множество методов" может быть очень эффективным.

2. Ниже приведен другой пример перегрузки функций. В этом случае функция **date()** перегружается для получения даты либо в виде строки, либо в виде трех целых. В обоих этих случаях функция выводит на экран переданные ей данные.

```
#include <iostream>
using namespace std;

void date(char *date); // дата в виде строки
void date(int month, int day, int year); // дата в виде чисел
```

```
int main()
{
   date("8/23/99");
   date( 8, 23, 99);

   return 0;
}

// Дата в виде строки
void date(char *date)
{
   cout << "Дата:" << date << "\n";
}

// Дата в виде целых
void date(int month, int day, int year)
{
   cout << "Дата:" << month << "/";
   cout << day << "/" << year << "\n";
}
```

Этот пример показывает, как перегрузка функций может обеспечить для функции более понятный интерфейс. Поскольку дату очень естественно представлять либо в виде строки, либо в виде трех целых чисел, содержащих месяц, день и год, нужно просто выбрать наиболее подходящую версию в соответствии с ситуацией.

3. До сих пор мы рассматривали перегруженные функции, отличающиеся типом своих аргументов. Однако перегруженные функции могут также отличаться и числом аргументов, как показано в приведенном ниже примере:

```
#include <iostream>
using namespace std;

void f1(int a);
void f1(int a, int b);

int main()
{
   f1(10);
   f1(10, 20);

   return 0;
}

void f1(int a)
{
   cout << "В f1(int a) \n";
}

void f1(int a, int b)
{
   cout << "В f1(int a, int b) \n";
}
```

4. Важно понимать, что тип возвращаемого значения сам по себе еще не является достаточным отличием для перегрузки функции. Если две функции отличаются только типом возвращаемых данных, компилятор не всегда сможет выбрать нужную. Например, следующий фрагмент неправилен, поскольку в нем имеет место избыточность:

```
// Это все неправильно и не будет компилироваться
```

```
int   fl(int  a);
double  fl(int a);
fl(10);   //   какую функцию выбрать   компилятору???
```

Как написано в комментарии, у компилятора нет способа выяснить, какую версию **f1()** вызвать.

1.8. Ключевые слова C++

В C++ поддерживаются все ключевые слова C и кроме этого еще 30 ключевых слов, которые относятся только к языку C++. Все определенные для C++ ключевые слова представлены в табл. 1.1. Кроме них в ранних версиях C++ было определено ключевое слово **overload,** которое сейчас считается устаревшим.

Таблица 1.1. *Ключевые слова C++*

asm	const_cast	explicit	int	register	switch	union
auto	continue	extern	long	reinterpret_cast	template	unsigned
bool	default	false	mutable	return	this	using
break	delete	float	namespace	short	throw	virtual
case	do	for	new	signed	true	void
catch	double	friend	operator	sizeof	try	volatile
char	dynamic_cast	goto	private	static	typedef	wchar_t
class	else	if	protected	static_cast	typeid	while
const	enum	inline	public	struct	typename	