

# 第 1 章

# 开始

## 内容

---

1.1 编写一个简单的 C++ 程序 .....	2
1.2 初识输入输出 .....	5
1.3 注释简介 .....	8
1.4 控制流 .....	10
1.5 类简介 .....	17
1.6 书店程序 .....	21
小结 .....	23
术语表 .....	23

本章介绍 C++ 的大部分基础内容：类型、变量、表达式、语句及函数。在这个过程中，我们会简要介绍如何编译及运行程序。

在学习完本章并认真完成练习之后，你将具备编写、编译及运行简单程序的能力。后续章节将假定你已掌握本章中介绍的语言特性，并将更详细地解释这些特性。

学习一门新的程序设计语言的最好方法就是练习编写程序。在本章中，我们将编写一个程序来解决简单的书店问题。

我们的书店保存所有销售记录的档案，每条记录保存了某本书的一次销售的信息（一册或多册）。每条记录包含三个数据项：

0-201-70353-X 4 24.99

第一项是书的 ISBN 号（国际标准书号，一本书的唯一标识），第二项是售出的册数，最后一项是书的单价。有时，书店老板需要查询此档案，计算每本书的销售量、销售额及平均售价。

为了编写这个程序，我们需要使用若干 C++ 的基本特性。而且，我们需要了解如何编译及运行程序。

虽然我们还没有编写这个程序，但显然它必须

- 定义变量
- 进行输入和输出
- 使用数据结构保存数据
- 检测两条记录是否有相同的 ISBN
- 包含一个循环来处理销售档案中的每条记录

我们首先介绍如何用 C++ 来解决这些子问题，然后编写书店程序。

## 1.1 编写一个简单的 C++ 程序

每个 C++ 程序都包含一个或多个函数（function），其中一个必须命名为 **main**。操作系统通过调用 **main** 来运行 C++ 程序。下面是一个非常简单的 **main** 函数，它什么也不干，只是返回给操作系统一个值：

```
int main()
{
    return 0;
}
```

一个函数的定义包含四部分：返回类型（return type）、函数名（function name）、一个括号包围的形参列表（parameter list，允许为空）以及函数体（function body）。虽然 **main** 函数在某种程度上比较特殊，但其定义与其他函数是一样的。

在本例中，**main** 的形参列表是空的（() 中什么也没有）。6.2.5 节（第 196 页）将会讨论 **main** 的其他形参类型。

**main** 函数的返回类型必须为 **int**，即整数类型。**int** 类型是一种 **内置类型**（built-in type），即语言自身定义的类型。

函数定义的最后一部分是函数体，它是一个以左花括号（curly brace）开始，以右花括号结束的语句块（block of statements）：

```
{  
    return 0;  
}
```

这个语句块中唯一的一条语句是 **return**，它结束函数的执行。在本例中，**return**

还会向调用者返回一个值。当 `return` 语句包括一个值时，此返回值的类型必须与函数的返回类型相容。在本例中，`main` 的返回类型是 `int`，而返回值 0 的确是一个 `int` 类型的值。



请注意，`return` 语句末尾的分号。在 C++ 中，大多数 C++ 语句以分号表示结束。它们很容易被忽略，但如果忘记了写分号，就会导致莫名其妙的编译错误。

在大多数系统中，`main` 的返回值被用来指示状态。返回值 0 表明成功，非 0 的返回值的含义由系统定义，通常用来指出错误类型。

### 重要概念：类型

类型是程序设计最基本的概念之一，在本书中我们会反复遇到它。一种类型不仅定义了数据元素的内容，还定义了这类数据上可以进行的运算。

程序所处理的数据都保存在变量中，而每个变量都有自己的类型。如果一个名为 `v` 的变量的类型为 `T`，我们通常说“`v` 具有类型 `T`”，或等价的，“`v` 是一个 `T` 类型变量”。

## 1.1.1 编译、运行程序

编写好程序后，我们就需要编译它。如何编译程序依赖于你使用的操作系统和编译器。你所使用的特定编译器的相关使用细节，请查阅参考手册或询问经验丰富的同事。

很多 PC 机上的编译器都具备集成开发环境（Integrated Developed Environment, IDE），将编译器与其他程序创建和分析工具包装在一起。在开发大型程序时，这类集成环境可能是非常有用的工具，但需要一些时间来学习如何高效地使用它们。学习如何使用这类开发环境已经超出了本书的范围。

大部分编译器，包括集成 IDE 的编译器，都会提供一个命令行界面。除非你已经了解 IDE，否则你会觉得借助命令行界面开始学习 C++ 还是很容易的。这种学习方式的好处是，可以先将精力集中于 C++ 语言本身（而不是一些开发工具），而且，一旦你掌握了语言，IDE 通常是很容易学习的。

### 程序源文件命名约定

无论你使用命令行界面或者 IDE，大多数编译器都要求程序源码存储在一个或多个文件中。程序文件通常被称为源文件（source file）。在大多数系统中，源文件的名字以一个后缀为结尾，后缀是由一个句点后接一个或多个字符组成的。后缀告诉系统这个文件是一个 C++ 程序。不同编译器使用不同的后缀命名约定，最常见的包括 `.cc`、`.cxx`、`.cpp`、`.cp` 及 `.c`。◀ 4

### 从命令行运行编译器

如果我们正在使用命令行界面，那么通常是在一个控制台窗口内（例如 UNIX 系统中的外壳程序窗口或者 Windows 系统中的命令提示符窗口）编译程序。假定我们的 `main` 程序保存在文件 `prog1.cc` 中，可以用如下命令来编译它

```
$ CC prog1.cc
```

其中，`CC` 是编译器程序的名字，`$` 是系统提示符。编译器生成一个可执行文件。Windows 系统会将这个可执行文件命名为 `prog1.exe`。UNIX 系统中的编译器通常将可执行文件命名为 `a.out`。

为了在 Windows 系统中运行一个可执行文件，我们需要提供可执行文件的文件名，可

以忽略其扩展名.exe:

```
$ prog1
```

在一些系统中，即使文件就在当前目录或文件夹中，你也必须显式指出文件的位置。在此情况下，我们可以键入

```
$ .\prog1
```

“.”后跟一个反斜线指出该文件在当前目录中。

为了在 UNIX 系统中运行一个可执行文件，我们需要使用全文件名，包括文件扩展名：

```
$ a.out
```

如果需要指定文件位置，需要用一个“.”后跟一个斜线来指出可执行文件位于当前目录中。

```
$ ./a.out
```

访问 main 的返回值的方法依赖于系统。在 UNIX 和 Windows 系统中，执行完一个程序后，都可以通过 echo 命令获得其返回值。

在 UNIX 系统中，通过如下命令获得状态：

```
$ echo $?
```

在 Windows 系统中查看状态可键入：

```
$ echo %ERRORLEVEL%
```

5

### 运行 GNU 或微软编译器

在不同操作和编译器系统中，运行 C++ 编译器的命令也各不相同。最常用的编译器是 GNU 编译器和微软 Visual Studio 编译器。默认情况下，运行 GNU 编译器的命令是 g++:

```
$ g++ -o prog1 prog1.cc
```

此处，\$ 是系统提示符。-o prog1 是编译器参数，指定了可执行文件的文件名。在不同的操作系统中，此命令生成一个名为 prog1 或 prog1.exe 的可执行文件。在 UNIX 系统中，可执行文件没有后缀；在 Windows 系统中，后缀为.exe。如果省略了 -o prog1 参数，在 UNIX 系统中编译器会生成一个名为 a.out 的可执行文件，在 Windows 系统中则会生成一个名为 a.exe 的可执行文件（注意：根据使用的 GNU 编译器的版本，你可能需要指定 -std=c++0x 参数来打开对 C++11 的支持）。

运行微软 Visual Studio 2010 编译器的命令为 cl:

```
C:\Users\me\Programs> cl /EHsc prog1.cpp
```

此处，C:\Users\me\Programs> 是系统提示符，\Users\me\Programs 是当前目录名（即当前文件夹）。命令 cl 调用编译器，/EHsc 是编译器选项，用来打开标准异常处理。微软编译器会自动生成一个可执行文件，其名字与第一个源文件名对应。可执行文件的文件名与源文件名相同，后缀为.exe。在此例中，可执行文件的文件名为 prog1.exe。

编译器通常都包含一些选项，能对有问题的程序结构发出警告。打开这些选项通常是一个好习惯。我们习惯在 GNU 编译器中使用 -Wall 选项，在微软编译器中则使用 /W4。

更详细的信息请查阅你使用的编译器的参考手册。

### 1.1 节练习

**练习 1.1:** 查阅你使用的编译器的文档，确定它所使用的文件命名约定。编译并运行第 2 页的 main 程序。

**练习 1.2:** 改写程序，让它返回 -1。返回值 -1 通常被当作程序错误的标识。重新编译并运行你的程序，观察你的系统如何处理 main 返回的错误标识。

## 1.2 初识输入输出

C++ 语言并未定义任何输入输出 (IO) 语句，取而代之，包含了一个全面的标准库 (standard library) 来提供 IO 机制（以及很多其他设施）。对于很多用途，包括本书中的示例来说，我们只需了解 IO 库中一部分基本概念和操作。

本书中的很多示例都使用了 **iostream** 库。**iostream** 库包含两个基础类型 **istream** 和 **ostream**，分别表示输入流和输出流。一个流就是一个字符序列，是从 IO 设备读出或写入 IO 设备的。术语“流”(stream) 想要表达的是，随着时间的推移，字符是顺序生成或消耗的。

### 标准输入输出对象

&lt; 6

标准库定义了 4 个 IO 对象。为了处理输入，我们使用一个名为 **cin** (发音为 see-in) 的 **istream** 类型的对象。这个对象也被称为标准输入 (standard input)。对于输出，我们使用一个名为 **cout** (发音为 see-out) 的 **ostream** 类型的对象。此对象也被称为标准输出 (standard output)。标准库还定义了其他两个 **ostream** 对象，名为 **cerr** 和 **clog** (发音分别为 see-err 和 see-log)。我们通常用 **cerr** 来输出警告和错误消息，因此它也被称为标准错误 (standard error)。而 **clog** 用来输出程序运行时的一般性信息。

系统通常将程序所运行的窗口与这些对象关联起来。因此，当我们读取 **cin**，数据将从程序正在运行的窗口读入，当我们向 **cout**、**cerr** 和 **clog** 写入数据时，将会写到同一个窗口。

### 一个使用 IO 库的程序

在书店程序中，我们需要将多条记录合并成单一的汇总记录。作为一个相关的，但更简单的问题，我们先来看一下如何将两个数相加。通过使用 IO 库，我们可以扩展 main 程序，使之能提示用户输入两个数，然后输出它们的和：

```
#include <iostream>
int main()
{
    std::cout << "Enter two numbers:" << std::endl;
    int v1 = 0, v2 = 0;
    std::cin >> v1 >> v2;
    std::cout << "The sum of " << v1 << " and " << v2
        << " is " << v1 + v2 << std::endl;
    return 0;
}
```

这个程序开始时在用户屏幕打印

**Enter two numbers:**

然后等待用户输入。如果用户键入

3 7

然后键入一个回车，则程序产生如下输出：

**The sum of 3 and 7 is 10**

程序的第一行

```
#include <iostream>
```

告诉编译器我们想要使用 `iostream` 库。尖括号中的名字（本例中是 `iostream`）指出了一个头文件（header）。每个使用标准库设施的程序都必须包含相关的头文件。`#include` 指令和头文件的名字必须写在同一行中。通常情况下，`#include` 指令必须出现在所有函数之外。我们一般将一个程序的所有`#include` 指令都放在源文件的开始位置。

### 向流写入数据

`main` 的函数体的第一条语句执行了一个表达式（expression）。在 C++ 中，一个表达式产生一个计算结果，它由一个或多个运算对象和（通常是）一个运算符组成。这条语句中的表达式使用了输出运算符（`<<`）在标准输出上打印消息：

```
std::cout << "Enter two numbers:" << std::endl;
```

`<<` 运算符接受两个运算对象：左侧的运算对象必须是一个 `ostream` 对象，右侧的运算对象是要打印的值。此运算符将给定的值写到给定的 `ostream` 对象中。输出运算符的计算结果就是其左侧运算对象。即，计算结果就是我们写入给定值的那个 `ostream` 对象。

我们的输出语句使用了两次`<<` 运算符。因为此运算符返回其左侧的运算对象，因此第一个运算符的结果成为了第二个运算符的左侧运算对象。这样，我们就可以将输出请求连接起来。因此，我们的表达式等价于

```
(std::cout << "Enter two numbers:") << std::endl;
```

链中每个运算符的左侧运算对象都是相同的，在本例中是 `std::cout`。我们也可以用两条语句生成相同的输出：

```
std::cout << "Enter two numbers:";  
std::cout << std::endl;
```

第一个输出运算符给用户打印一条消息。这个消息是一个字符串字面值常量（string literal），是用一对双引号包围的字符序列。在双引号之间的文本被打印到标准输出。

第二个运算符打印 `endl`，这是一个被称为操纵符（manipulator）的特殊值。写入 `endl` 的效果是结束当前行，并将与设备关联的缓冲区（buffer）中的内容刷到设备中。缓冲刷新操作可以保证到目前为止程序所产生的所有输出都真正写入输出流中，而不是仅停留在内存中等待写入流。



程序员常常在调试时添加打印语句。这类语句应该保证“一直”刷新流。否则，如果程序崩溃，输出可能还留在缓冲区中，从而导致关于程序崩溃位置的错误推断。

## 使用标准库中的名字

细心的读者可能会注意到这个程序使用了 `std::cout` 和 `std::endl`，而不是直接的 `cout` 和 `endl`。前缀 `std::` 指出名字 `cout` 和 `endl` 是定义在名为 **std** 的命名空间（namespace）中的。命名空间可以帮助我们避免不经意的名字定义冲突，以及使用库中相同名字导致的冲突。标准库定义的所有名字都在命名空间 `std` 中。

8

通过命名空间使用标准库有一个副作用：当使用标准库中的一个名字时，必须显式说明我们想使用来自命名空间 `std` 中的名字。例如，需要写出 `std::cout`，通过使用作用域运算符（`::`）来指出我们想使用定义在命名空间 `std` 中的名字 `cout`。3.1 节（第 74 页）将给出一个更简单的访问标准库中名字的方法。

## 从流读取数据

在提示用户输入数据之后，接下来我们希望读入用户的输入。首先定义两个名为 `v1` 和 `v2` 的变量（variable）来保存输入：

```
int v1 = 0, v2 = 0;
```

我们将这两个变量定义为 `int` 类型，`int` 是一种内置类型，用来表示整数。还将它们初始化（initialize）为 0。初始化一个变量，就是在变量创建的同时为它赋予一个值。

下一条语句是

```
std::cin >> v1 >> v2;
```

它读入输入数据。输入运算符（`>>`）与输出运算符类似，它接受一个 `istream` 作为其左侧运算对象，接受一个对象作为其右侧运算对象。它从给定的 `istream` 读入数据，并存入给定对象中。与输出运算符类似，输入运算符返回其左侧运算对象作为其计算结果。因此，此表达式等价于

```
(std::cin >> v1) >> v2;
```

由于此运算符返回其左侧运算对象，因此我们可以将一系列输入请求合并到单一语句中。本例中的输入操作从 `std::cin` 读入两个值，并将第一个值存入 `v1`，将第二个值存入 `v2`。换句话说，它与下面两条语句的执行结果是一样的

```
std::cin >> v1;  
std::cin >> v2;
```

## 完成程序

剩下的就是打印计算结果了：

```
std::cout << "The sum of " << v1 << " and " << v2  
<< " is " << v1 + v2 << std::endl;
```

这条语句虽然比提示用户输入的打印语句更长，但原理上是一样的，它将每个运算对象打印在标准输出上。本例一个有意思的地方在于，运算对象并不都是相同类型的值。某些运算对象是字符串字面值常量，例如 "The sum of "。其他运算对象则是 `int` 值，如 `v1`、`v2` 以及算术表达式 `v1+v2` 的计算结果。标准库定义了不同版本的输入输出运算符，来处理这些不同类型的运算对象。

9

## 1.2 节练习

**练习 1.3:** 编写程序，在标准输出上打印 Hello, World。

**练习 1.4:** 我们的程序使用加法运算符+来将两个数相加。编写程序使用乘法运算符\*，来打印两个数的积。

**练习 1.5:** 我们将所有输出操作放在一条很长的语句中。重写程序，将每个运算对象的打印操作放在一条独立的语句中。

**练习 1.6:** 解释下面程序片段是否合法。

```
std::cout << "The sum of " << v1;
           << " and " << v2;
           << " is " << v1 + v2 << std::endl;
```

如果程序是合法的，它输出什么？如果程序不合法，原因何在？应该如何修正？

## 1.3 注释简介

在程序变得更复杂之前，我们应该了解一下 C++ 是如何处理注释（comments）的。注释可以帮助人类读者理解程序。注释通常用于概述算法，确定变量的用途，或者解释晦涩难懂的代码段。编译器会忽略注释，因此注释对程序的行为或性能不会有任何影响。

虽然编译器会忽略注释，但读者并不会。即使系统文档的其他部分已经过时，程序员也倾向于相信注释的内容是正确可信的。因此，错误的注释比完全没有注释更糟糕，因为它会误导读者。因此，当你修改代码时，不要忘记同时更新注释！

### C++ 中注释的种类

C++ 中有两种注释：单行注释和界定符对注释。单行注释以双斜线（//）开始，以换行符结束。当前行双斜线右侧的所有内容都会被编译器忽略，这种注释可以包含任何文本，包括额外的双斜线。

另一种注释使用继承自 C 语言的两个界定符（/\* 和 \*/）。这种注释以/\* 开始，以\*/ 结束，可以包含除\*/外的任意内容，包括换行符。编译器将落在/\* 和 \*/之间的所有内容都当作注释。

注释界定符可以放置于任何允许放置制表符、空格符或换行符的地方。注释界定符可以跨越程序中的多行，但这并不是必须的。当注释界定符跨过多行时，最好能显式指出其内部的程序行都属于多行注释的一部分。我们所采用的风格是，注释内的每行都以一个星号开头，从而指出整个范围都是多行注释的一部分。

10

程序中通常同时包含两种形式的注释。注释界定符对通常用于多行解释，而双斜线注释常用于半行和单行附注。

```
#include <iostream>
/*
 * 简单主函数：
 * 读取两个数，求它们的和
 */
int main()
{
```

```
// 提示用户输入两个数
std::cout << "Enter two numbers:" << std::endl;
int v1 = 0, v2 = 0; // 保存我们读入的输入数据的变量
std::cin >> v1 >> v2; // 读取输入数据
std::cout << "The sum of " << v1 << " and " << v2
    << " is " << v1 + v2 << std::endl;
return 0;
}
```



在本书中，我们用楷体来突出显示注释。在实际程序中，注释文本的显示形式是否区别于程序代码文本的显示，依赖于你所使用的程序设计环境是否提供这一特性。

### 注释界定符不能嵌套

界定符对形式的注释是以`/*`开始，以`*/`结束的。因此，一个注释不能嵌套在另一个注释之内。编译器对这类问题所给出的错误信息可能是难以理解、令人迷惑的。例如，在你的系统中编译下面的程序，就会产生错误：

```
/*
 * 注释对/* */不能嵌套。
 * “不能嵌套”几个字会被认为是源码,
 * 像剩余程序一样处理
 */
int main()
{
    return 0;
}
```

我们通常需要在调试期间注释掉一些代码。由于这些代码可能包含界定符对形式的注释，因此可能导致注释嵌套错误，因此最好的方式是用单行注释方式注释掉代码段的每一行。

```
// /*
// * 单行注释中的任何内容都会被忽略
// * 包括嵌套的注释对也一样会被忽略
// */
```

### 1.3 节练习

11

练习 1.7：编译一个包含不正确的嵌套注释的程序，观察编译器返回的错误信息。

练习 1.8：指出下列哪些输出语句是合法的（如果有的话）：

```
std::cout << "/*";
std::cout << "*/";
std::cout << /* "*/" */;
std::cout << /* "*/" /* "/*" */;
```

预测编译这些语句会产生什么样的结果，实际编译这些语句来验证你的答案（编写一个小程序，每次将上述一条语句作为其主体），改正每个编译错误。

## 1.4 控制流

语句一般是顺序执行的：语句块的第一条语句首先执行，然后是第二条语句，依此类推。当然，少数组程序，包括我们解决书店问题的程序，都可以写成只有顺序执行的形式。但程序设计语言提供了多种不同的控制流语句，允许我们写出更为复杂的执行路径。

### 1.4.1 while 语句

**while** 语句反复执行一段代码，直至给定条件为假为止。我们可以用 **while** 语句编写一段程序，求 1 到 10 这 10 个数之和：

```
#include <iostream>
int main()
{
    int sum = 0, val = 1;
    // 只要 val 的值小于等于 10，while 循环就会持续执行
    while (val <= 10) {
        sum += val; // 将 sum + val 赋予 sum
        ++val;       // 将 val 加 1
    }
    std::cout << "Sum of 1 to 10 inclusive is "
              << sum << std::endl;
    return 0;
}
```

我们编译并执行这个程序，它会打印出

**Sum of 1 to 10 inclusive is 55**

与之前的例子一样，我们首先包含头文件 `iostream`，然后定义 `main`。在 `main` 中我们定义两个 `int` 变量：`sum` 用来保存和；`val` 用来表示从 1 到 10 的每个数。我们将 `sum` 的初值设置为 0，`val` 从 1 开始。

12

这个程序的新内容是 `while` 语句。`while` 语句的形式为

```
while (condition)
    statement
```

`while` 语句的执行过程是交替地检测 `condition` 条件和执行关联的语句 `statement`，直至 `condition` 为假时停止。所谓条件（`condition`）就是一个产生真或假的结果的表达式。只要 `condition` 为真，`statement` 就会被执行。当执行完 `statement`，会再次检测 `condition`。如果 `condition` 仍为真，`statement` 再次被执行。`while` 语句持续地交替检测 `condition` 和执行 `statement`，直至 `condition` 为假为止。

在本程序中，`while` 语句是这样的

```
// 只要 val 的值小于等于 10，while 循环就会持续执行
while (val <= 10) {
    sum += val; // 将 sum + val 赋予 sum
    ++val;       // 将 val 加 1
}
```

条件中使用了小于等于运算符（`<=`）来比较 `val` 的当前值和 10。只要 `val` 小于等于 10，条件即为真。如果条件为真，就执行 `while` 循环体。在本例中，循环体是由两条语句组

成的语句块：

```
{  
    sum += val;      // 将 sum + val 赋予 sum  
    ++val;          // 将 val 加 1  
}
```

所谓语句块（block），就是用花括号包围的零条或多条语句的序列。语句块也是语句的一种，在任何要求使用语句的地方都可以使用语句块。在本例中，语句块的第一条语句使用了复合赋值运算符（`+=`）。此运算符将其右侧的运算对象加到左侧运算对象上，将结果保存到左侧运算对象中。它本质上与一个加法结合一个赋值（assignment）是相同的：

```
sum = sum + val; // 将 sum + val 赋予 sum
```

因此，语句块中第一条语句将 `val` 的值加到当前和 `sum` 上，并将结果保存在 `sum` 中。

下一条语句

```
++val; // 将 val 加 1
```

使用前缀递增运算符（`++`）。递增运算符将运算对象的值增加 1。`++val` 等价于 `val=val+1`。

执行完 `while` 循环体后，循环会再次对条件进行求值。如果 `val` 的值（现在已经增加了）仍然小于等于 10，则 `while` 的循环体会再次执行。循环连续检测条件、执行循环体，直至 `val` 不再小于等于 10 为止。

一旦 `val` 大于 10，程序跳出 `while` 循环，继续执行 `while` 之后的语句。在本例中，继续执行打印输出语句，然后执行 `return` 语句完成 `main` 程序。

### 1.4.1 节练习

13

**练习 1.9：**编写程序，使用 `while` 循环将 50 到 100 的整数相加。

**练习 1.10：**除了 `++` 运算符将运算对象的值增加 1 之外，还有一个递减运算符（`-`）实现将值减少 1。编写程序，使用递减运算符在循环中按递减顺序打印出 10 到 0 之间的整数。

**练习 1.11：**编写程序，提示用户输入两个整数，打印出这两个整数所指定的范围内的所有整数。

### 1.4.2 for 语句

在我们的 `while` 循环例子中，使用了变量 `val` 来控制循环执行次数。我们在循环条件中检测 `val` 的值，在 `while` 循环体中将 `val` 递增。

这种在循环条件中检测变量、在循环体中递增变量的模式使用非常频繁，以至于 C++ 语言专门定义了第二种循环语句——**for 语句**，来简化符合这种模式的语句。可以用 `for` 语句来重写从 1 加到 10 的程序：

```
#include <iostream>  
int main()  
{  
    int sum = 0;  
    // 从 1 加到 10  
    for (int val = 1; val <= 10; ++val)
```

```

        sum += val; // 等价于 sum = sum + val
    std::cout << "Sum of 1 to 10 inclusive is "
        << sum << std::endl;
    return 0;
}

```

与之前一样，我们定义了变量 `sum`，并将其初始化为 0。在此版本中，`val` 的定义是 `for` 语句的一部分：

```

for (int val = 1; val <= 10; ++val)
    sum += val;

```

每个 `for` 语句都包含两部分：循环头和循环体。循环头控制循环体的执行次数，它由三部分组成：一个初始化语句（*init-statement*）、一个循环条件（*condition*）以及一个表达式（*expression*）。在本例中，初始化语句为

```
int val = 1
```

它定义了一个名为 `val` 的 `int` 型对象，并为其赋初值 1。变量 `val` 仅在 `for` 循环内部存在，在循环结束之后是不能使用的。初始化语句只在 `for` 循环入口处执行一次。循环条件

```
val <= 10
```

**14** 比较 `val` 的值和 10。循环体每次执行前都会先检查循环条件。只要 `val` 小于等于 10，就会执行 `for` 循环体。表达式在 `for` 循环体之后执行。在本例中，表达式

```
++val
```

使用前缀递增运算符将 `val` 的值增加 1。执行完表达式后，`for` 语句重新检测循环条件。如果 `val` 的新值仍然小于等于 10，就再次执行 `for` 循环体。执行完循环体后，再次将 `val` 的值增加 1。循环持续这一过程直至循环条件为假。

在此循环中，`for` 循环体执行加法

```
sum += val; // 等价于 sum = sum + val
```

简要重述一下 `for` 循环的总体执行流程：

1. 创建变量 `val`，将其初始化为 1。
2. 检测 `val` 是否小于等于 10。若检测成功，执行 `for` 循环体。若失败，退出循环，继续执行 `for` 循环体之后的第一条语句。
3. 将 `val` 的值增加 1。
4. 重复第 2 步中的条件检测，只要条件为真就继续执行剩余步骤。

### 1.4.2 节练习

**练习 1.12：**下面的 `for` 循环完成了什么功能？`sum` 的终值是多少？

```

int sum = 0;
for (int i = -100; i <= 100; ++i)
    sum += i;

```

**练习 1.13：**使用 `for` 循环重做 1.4.1 节中的所有练习（第 11 页）。

练习 1.14：对比 `for` 循环和 `while` 循环，两种形式的优缺点各是什么？

练习 1.15：编写程序，包含第 14 页“再探编译”中讨论的常见错误。熟悉编译器生成的错误信息。

### 1.4.3 读取数量不定的输入数据

在前一节中，我们编写程序实现了 1 到 10 这 10 个整数求和。扩展此程序一个很自然的方向是实现对用户输入的一组数求和。在这种情况下，我们预先不知道要对多少个数求和，这就需要不断读取数据直至没有新的输入为止：

```
#include <iostream>
int main()
{
    int sum = 0, value = 0;
    // 读取数据直到遇到文件尾，计算所有读入的值的和
    while (std::cin >> value)
        sum += value; // 等价于 sum = sum + value
    std::cout << "Sum is: " << sum << std::endl;
    return 0;
}
```

15

如果我们输入

3 4 5 6

则程序会输出

**Sum is: 18**

`main` 的首行定义了两个名为 `sum` 和 `value` 的 `int` 变量，均初始化为 0。我们使用 `value` 保存用户输入的每个数，数据读取操作是在 `while` 的循环条件中完成的：

```
while (std::cin >> value)
```

`while` 循环条件的求值就是执行表达式

```
std::cin >> value
```

此表达式从标准输入读取下一个数，保存在 `value` 中。输入运算符（参见 1.2 节，第 7 页）返回其左侧运算对象，在本例中是 `std::cin`。因此，此循环条件实际上检测的是 `std::cin`。

当我们使用一个 `istream` 对象作为条件时，其效果是检测流的状态。如果流是有效的，即流未遇到错误，那么检测成功。当遇到文件结束符（end-of-file），或遇到一个无效输入时（例如读入的值不是一个整数），`istream` 对象的状态会变为无效。处于无效状态的 `istream` 对象会使条件变为假。

因此，我们的 `while` 循环会一直执行直至遇到文件结束符（或输入错误）。`while` 循环体使用复合赋值运算符将当前值加到 `sum` 上。一旦条件失败，`while` 循环将会结束。我们将执行下一条语句，打印 `sum` 的值和一个 `endl`。

## 从键盘输入文件结束符

当从键盘向程序输入数据时，对于如何指出文件结束，不同操作系统有不同的约定。在 Windows 系统中，输入文件结束符的方法是敲 Ctrl+Z（按住 Ctrl 键的同时按 Z 键），然后按 Enter 或 Return 键。在 UNIX 系统中，包括 Mac OS X 系统中，文件结束符输入是用 Ctrl+D。

16 &gt;

## 再探编译

编译器的一部分工作是寻找程序文本中的错误。编译器没有能力检查一个程序是否按照其作者的意图工作，但可以检查形式（form）上的错误。下面列出了一些最常见的编译器可以检查出的错误。

**语法错误 (syntax error)**: 程序员犯了 C++ 语言文法上的错误。下面程序展示了一些常见的语法错误；每条注释描述了下一行中语句存在的错误：

```
// 错误：main 的参数列表漏掉了
int main (
    // 错误：endl 后使用了冒号而非分号
    std::cout << "Read each file." << std::endl;
    // 错误：字符串字面常量的两侧漏掉了引号
    std::cout << Update master. << std::endl;
    // 错误：漏掉了第二个输出运算符
    std::cout << "Write new master." std::endl;
    // 错误：return 语句漏掉了分号
    return 0
}
```

**类型错误 (type error)**: C++ 中每个数据项都有其类型。例如，10 的类型是 int（或者更通俗地说，“10 是一个 int 型数据”）。单词“hello”，包括两侧的双引号标记，则是一个字符串字面值常量。一个类型错误的例子是，向一个期望参数为 int 的函数传递了一个字符串字面值常量。

**声明错误 (declaration error)**: C++ 程序中的每个名字都要先声明后使用。名字声明失败通常会导致一条错误信息。两种常见的声明错误是：对来自标准库的名字忘记使用 std::、标识符名字拼写错误：

```
#include <iostream>
int main()
{
    int v1 = 0, v2 = 0;
    std::cin >> v >> v2; // 错误：使用了 "v" 而非 "v1"
    // 错误：cout 未定义；应该是 std::cout
    cout << v1 + v2 << std::endl;
    return 0;
}
```

错误信息通常包含一个行号和一条简短描述，描述了编译器认为的我们所犯的错误。按照报告的顺序来逐个修正错误，是一种好习惯。因为一个单个错误常常会具有传递效应，导致编译器在其后报告比实际数量多得多的错误信息。另一个好习惯是在每修

正一个错误后就立即重新编译代码，或者最多是修正了一小部分明显的错误后就重新编译。这就是所谓的“编辑-编译-调试”(edit-compile-debug)周期。

### 1.4.3 节练习

17

练习 1.16：编写程序，从 cin 读取一组数，输出其和。

### 1.4.4 if 语句

与大多数语言一样，C++也提供了 **if** 语句来支持条件执行。我们可以用 **if** 语句写一个程序，来统计在输入中每个值连续出现了多少次：

```
#include <iostream>
int main()
{
    // currVal 是我们正在统计的数；我们将读入的新值存入 val
    int currVal = 0, val = 0;
    // 读取第一个数，并确保确实有数据可以处理
    if (std::cin >> currVal) {
        int cnt = 1; // 保存我们正在处理的当前值的个数
        while (std::cin >> val) { // 读取剩余的数
            if (val == currVal) // 如果值相同
                ++cnt; // 将 cnt 加 1
            else { // 否则，打印前一个值的个数
                std::cout << currVal << " occurs "
                << cnt << " times" << std::endl;
                currVal = val; // 记住新值
                cnt = 1; // 重置计数器
            }
        } // while 循环在这里结束
        // 记住打印文件中最后一个值的个数
        std::cout << currVal << " occurs "
        << cnt << " times" << std::endl;
    } // 最外层的 if 语句在这里结束
    return 0;
}
```

如果我们输入如下内容：

42 42 42 42 55 55 62 100 100 100

则输出应该是：

```
42 occurs 5 times
55 occurs 2 times
62 occurs 1 times
100 occurs 3 times
```

有了之前多个程序的基础，你对这个程序中的大部分代码应该比较熟悉了。程序以两个变量 **val** 和 **currVal** 的定义开始：**currVal** 记录我们正在统计出现次数的那个数；**val** 则保存从输入读取的每个数。与之前的程序相比，新的内容就是两个 **if** 语句。第一条 **if** 语句

18 > if (std::cin >> currVal) {  
    // ...  
} //最外层的 if 语句在这里结束

保证输入不为空。与 while 语句类似，if 也对一个条件进行求值。第一条 if 语句的条件是读取一个数值存入 currVal 中。如果读取成功，则条件为真，我们继续执行条件之后的语句块。该语句块以左花括号开始，以 return 语句之前的右花括号结束。

如果需要统计出现次数的值，我们就定义 cnt，用来统计每个数值连续出现的次数。与上一小节的程序类似，我们用一个 while 循环反复从标准输入读取整数。

while 的循环体是一个语句块，它包含了第二条 if 语句：

```
if (val == currVal)          // 如果值相同
    ++cnt;                  // 将 cnt 加 1
else {                      // 否则，打印前一个值的个数
    std::cout << currVal << " occurs "
        << cnt << " times" << std::endl;
    currVal = val;           // 记住新值
    cnt = 1;                 // 重置计数器
}
```

这条 if 语句中的条件使用了相等运算符（==）来检测 val 是否等于 currVal。如果是，我们执行紧跟在条件之后的语句。这条语句将 cnt 增加 1，表明我们再次看到了 currVal。

如果条件为假，即 val 不等于 currVal，则执行 else 之后的语句。这条语句是一个由一条输出语句和两条赋值语句组成的语句块。输出语句打印我们刚刚统计完的值的出现次数。赋值语句将 cnt 重置为 1，将 currVal 重置为刚刚读入的值 val。



C++用=进行赋值，用==作为相等运算符。两个运算符都可以出现在条件中。  
一个常见的错误是想在条件中使用==（相等判断），却误用了=。

#### 1.4.4 节练习

**练习 1.17：**如果输入的所有值都是相等的，本节的程序会输出什么？如果没有重复值，输出又会是怎样的？

**练习 1.18：**编译并运行本节的程序，给它输入全都相等的值。再次运行程序，输入没有重复的值。

**练习 1.19：**修改你为 1.4.1 节练习 1.10（第 11 页）所编写的程序（打印一个范围内的数），使其能处理用户输入的第一个数比第二个数小的情况。

#### 关键概念：C++程序的缩进和格式

C++程序很大程度上是格式自由的，也就是说，我们在哪里放置花括号、缩进、注释以及换行符通常不会影响程序的语义。例如，花括号表示 main 函数体的开始，它可以放在 main 的同一行中；也可以像我们所做的那样，放在下一行的起始位置；还可以放在我们喜欢的其他任何位置。唯一的要求是左花括号必须是 main 的形参列表后第一个非空、非注释的字符。

虽然很大程度上可以按照自己的意愿自由地设定程序的格式，但我们所做的选择会影响程序的可读性。例如，我们可以将整个 main 函数写在很长的单行内，虽然这样是合乎语法的，但会非常难读。

关于 C/C++ 的正确格式的辩论是无休止的。我们的信条是，不存在唯一正确的风格，但保持一致性是非常重要的。例如，大多数程序员都对程序的组成部分设置恰当的缩进，就像我们在之前的例子中对 main 函数中的语句和循环体所做的那样。对于作为函数界定符的花括号，我们习惯将其放在单独一行中。我们还习惯对复合 IO 表达式设置缩进，以使输入输出运算符排列整齐。其他一些缩进约定也都会令越来越复杂的程序更加清晰易读。

我们要牢记一件重要的事情：其他可能的程序格式总是存在的。当你要选择一种格式风格时，思考一下它会对程序的可读性和易理解性有什么影响，而一旦选择了一种风格，就要坚持使用。

## 1.5 类简介

在解决书店程序之前，我们还需要了解的唯一一个 C++ 特性，就是如何定义一个数据结构（data structure）来表示销售数据。在 C++ 中，我们通过定义一个类（class）来定义自己的数据结构。一个类定义了一个类型，以及与其关联的一组操作。类机制是 C++ 最重要的特性之一。实际上，C++ 最初的一个设计焦点就是能定义使用上像内置类型一样自然的类类型（class type）。

在本节中，我们将介绍一个在编写书店程序中会用到的简单的类。当我们在后续章节中学习了更多关于类型、表达式、语句和函数的知识后，会真正实现这个类。

为了使用类，我们需要了解三件事情：

- 类名是什么？
- 它是在哪里定义的？
- 它支持什么操作？

对于书店程序来说，我们假定类名为 Sales\_item，头文件 Sales\_item.h 中已经定义了这个类。

如前所见，为了使用标准库设施，我们必须包含相关的头文件。类似的，我们也需要使用头文件来访问为自己的应用程序所定义的类。习惯上，头文件根据其中定义的类的名字来命名。我们通常使用.h 作为头文件的后缀，但也有一些程序员习惯.h、.hpp 或.hxx。标准库头文件通常不带后缀。编译器一般不关心头文件名的形式，但有的 IDE 对此有特定要求。

### 1.5.1 Sales\_item 类

Sales\_item 类的作用是表示一本书的总销售额、售出册数和平均售价。我们现在不关心这些数据如何存储、如何计算。为了使用一个类，我们不必关心它是如何实现的，只需知道类对象可以执行什么操作。

每个类实际上都定义了一个新的类型，其类型名就是类名。因此，我们的 Sales\_item 类定义了一个名为 Sales\_item 的类型。与内置类型一样，我们可以定义类类型的变量。当我们写下如下语句

```
Sales_item item;
```

是想表达 `item` 是一个 `Sales_item` 类型的对象。我们通常将“一个 `Sales_item` 类型的对象”简单说成“一个 `Sales_item` 对象”，或更简单的“一个 `Sales_item`”。

除了可以定义 `Sales_item` 类型的变量之外，我们还可以：

- 调用一个名为 `isbn` 的函数从一个 `Sales_item` 对象中提取 ISBN 书号。
- 用输入运算符 (`>>`) 和输出运算符 (`<<`) 读、写 `Sales_item` 类型的对象。
- 用赋值运算符 (`=`) 将一个 `Sales_item` 对象的值赋予另一个 `Sales_item` 对象。
- 用加法运算符 (`+`) 将两个 `Sales_item` 对象相加。两个对象必须表示同一本书（相同的 ISBN）。加法结果是一个新的 `Sales_item` 对象，其 ISBN 与两个运算对象相同，而其总销售额和售出册数则是两个运算对象的对应值之和。
- 使用复合赋值运算符 (`+=`) 将一个 `Sales_item` 对象加到另一个对象上。

### 关键概念：类定义了行为

当你读这些程序时，一件要牢记的重要事情是，类 `Sales_item` 的作者定义了类对象可以执行的所有动作。即，`Sales_item` 类定义了创建一个 `Sales_item` 对象时会发生什么事情，以及对 `Sales_item` 对象进行赋值、加法或输入输出运算时会发生什么事情。

一般而言，类的作者决定了类型对象上可以使用的所有操作。当前，我们所知道的可以在 `Sales_item` 对象上执行的全部操作就是本节所列出的那些操作。

## 21 读写 `Sales_item`

既然已经知道可以对 `Sales_item` 对象执行哪些操作，我们现在就可以编写使用类的程序了。例如，下面的程序从标准输入读入数据，存入一个 `Sales_item` 对象中，然后将 `Sales_item` 的内容写回到标准输出：

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item book;
    // 读入 ISBN 号、售出的册数以及销售价格
    std::cin >> book;
    // 写入 ISBN、售出的册数、总销售额和平均价格
    std::cout << book << std::endl;
    return 0;
}
```

如果输入：

**0-201-70353-X 4 24.99**

则输出为：

**0-201-70353-X 4 99.96 24.99**

输入表示我们以每本 24.99 美元的价格售出了 4 册书，而输出告诉我们总售出册数为 4，总销售额为 99.96 美元，而每册书的平均销售价格为 24.99 美元。

此程序以两个`#include` 指令开始，其中一个使用了新的形式。包含来自标准库的头

文件时，也应该用尖括号（< >）包围头文件名。对于不属于标准库的头文件，则用双引号（" "）包围。

在 main 中我们定义了一个名为 book 的对象，用来保存从标准输入读取出的数据。下一条语句读取数据存入对象中，第三条语句将对象打印到标准输出上并打印一个 endl。

### Sales\_item 对象的加法

下面是一个更有意思的例子，将两个 Sales\_item 对象相加：

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;           // 读取一对交易记录
    std::cout << item1 + item2 << std::endl; // 打印它们的和
    return 0;
}
```

如果输入如下内容：

```
0-201-78345-X 3 20.00
0-201-78345-X 2 25.00
```

则输出为：

```
0-201-78345-X 5 110 22
```

此程序开始包含了 Sales\_item 和 iostream 两个头文件。然后定义了两个 Sales\_item 对象来保存销售记录。我们从标准输入读取数据，存入两个对象之中。输出表达式完成加法运算并打印结果。

值得注意的是，此程序看起来与第 5 页的程序非常相似：读取两个输入数据并输出它们的和。造成如此相似的原因是，我们只不过将运算对象从两个整数变为两个 Sales\_item 而已，但读取与打印和的运算方式没有发生任何变化。两个程序的另一个不同之处是，“和”的概念是完全不一样的。对于 int，我们计算传统意义上的和——两个数值的算术加法结果。对于 Sales\_item 对象，我们用了一个全新的“和”的概念——两个 Sales\_item 对象的成员对应相加的结果。

### 使用文件重定向

当你测试程序时，反复从键盘敲入这些销售记录作为程序的输入，是非常乏味的。大多数操作系统支持文件重定向，这种机制允许我们将标准输入和标准输出与命名文件关联起来：

```
$ addItems <infile >outfile
```

假定 \$ 是操作系统提示符，我们的加法程序已经编译为名为 addItems.exe 的可执行文件（在 UNIX 中是 addItems），则上述命令会从一个名为 infile 的文件读取销售记录，并将输出结果写入到一个名为 outfile 的文件中，两个文件都位于当前目录中。

### 1.5.1 节练习

**练习 1.20:** 在网站 <http://www.informit.com/title/0321714113> 上, 第 1 章的代码目录中包含了头文件 Sales\_item.h。将它拷贝到你自己的工作目录中。用它编写一个程序, 读取一组书籍销售记录, 将每条记录打印到标准输出上。

**练习 1.21:** 编写程序, 读取两个 ISBN 相同的 Sales\_item 对象, 输出它们的和。

**练习 1.22:** 编写程序, 读取多个具有相同 ISBN 的销售记录, 输出所有记录的和。

## 23 1.5.2 初识成员函数

将两个 Sales\_item 对象相加的程序首先应该检查两个对象是否具有相同的 ISBN。方法如下:

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item item1, item2;
    std::cin >> item1 >> item2;
    // 首先检查 item1 和 item2 是否表示相同的书
    if (item1.isbn() == item2.isbn()) {
        std::cout << item1 + item2 << std::endl;
        return 0;      // 表示成功
    } else {
        std::cerr << "Data must refer to same ISBN"
            << std::endl;
        return -1;      // 表示失败
    }
}
```

此程序与上一版本的差别是 if 语句及其 else 分支。即使不了解这个 if 语句的检测条件, 我们也很容易理解这个程序在干什么。如果条件成立, 如上一版本一样, 程序打印计算结果, 并返回 0, 表明成功。如果条件失败, 我们执行跟在 else 之后的语句块, 打印一条错误信息, 并返回一个错误标识。

### 什么是成员函数?

这个 if 语句的检测条件

```
item1.isbn() == item2.isbn()
```

调用名为 isbn 的成员函数 (member function)。成员函数是定义为类的一部分的函数, 有时也被称为方法 (method)。

我们通常以一个类对象的名义来调用成员函数。例如, 上面相等表达式左侧运算对象的第一部分

```
item1.isbn()
```

使用点运算符 (.) 来表达我们需要“名为 item1 的对象的 isbn 成员”。点运算符只能用于类类型的对象。其左侧运算对象必须是一个类类型的对象, 右侧运算对象必须是该类型的一个成员名, 运算结果为右侧运算对象指定的成员。

当用点运算符访问一个成员函数时，通常我们是想（效果也确实是）调用该函数。我们使用调用运算符（`()`）来调用一个函数。调用运算符是一对圆括号，里面放置实参（argument）列表（可能为空）。成员函数 `isbn` 并不接受参数。因此

```
item1.isbn()
```

调用名为 `item1` 的对象的成员函数 `isbn`，此函数返回 `item1` 中保存的 ISBN 书号。

< 24

在这个 `if` 条件中，相等运算符的右侧运算对象也是这样执行的——它返回保存在 `item2` 中的 ISBN 书号。如果 ISBN 相同，条件为真，否则为假。

## 1.5.2 节练习

**练习 1.23：** 编写程序，读取多条销售记录，并统计每个 ISBN（每本书）有几条销售记录。

**练习 1.24：** 输入表示多个 ISBN 的多条销售记录来测试上一个程序，每个 ISBN 的记录应该聚在一起。

## 1.6 书店程序

现在我们已经准备好完成书店程序了。我们需要从一个文件中读取销售记录，生成每本书的销售报告，显示售出册数、总销售额和平均售价。我们假定每个 ISBN 书号的所有销售记录在文件中是聚在一起保存的。

我们的程序会将每个 ISBN 的所有数据合并起来，存入名为 `total` 的变量中。我们使用另一个名为 `trans` 的变量保存读取的每条销售记录。如果 `trans` 和 `total` 指向相同的 ISBN，我们会更新 `total` 的值。否则，我们会打印 `total` 的值，并将其重置为刚刚读取的数据 (`trans`)：

```
#include <iostream>
#include "Sales_item.h"
int main()
{
    Sales_item total; // 保存下一条交易记录的变量
    // 读入第一条交易记录，并确保有数据可以处理
    if (std::cin >> total) {
        Sales_item trans; // 保存和的变量
        // 读入并处理剩余交易记录
        while (std::cin >> trans) {
            // 如果我们仍在处理相同的书
            if (total.isbn() == trans.isbn())
                total += trans; // 更新总销售额
            else {
                // 打印前一本书的结果
                std::cout << total << std::endl;
                total = trans; // total 现在表示下一本书的销售额
            }
        }
        std::cout << total << std::endl; // 打印最后一本书的结果
    } else {
}
```

```
25>     //没有输入！警告读者
        std::cerr << "No data?!" << std::endl;
        return -1; // 表示失败
    }
    return 0;
}
```

这是到目前为止我们看到的最复杂的程序了，但它所使用的都是我们已经见过的语言特性。

与往常一样，首先包含要使用的头文件：来自标准库的 `iostream` 和自己定义的 `Sales_item.h`。在 `main` 中，我们定义了一个名为 `total` 的变量，用来保存一个给定的 ISBN 的数据之和。我们首先读取第一条销售记录，存入 `total` 中，并检测这次读取操作是否成功。如果读取失败，则意味着没有任何销售记录，于是直接跳到最外层的 `else` 分支，打印一条警告信息，告诉用户没有输入。

假定已经成功读取了一条销售记录，我们继续执行最外层 `if` 之后的语句块。这个语句块首先定义一个名为 `trans` 的对象，它保存读取的销售记录。接下来的 `while` 语句将读取剩下的所有销售记录。与我们之前的程序一样，`while` 条件是一个从标准输入读取值的操作。在本例中，我们读取一个 `Sales_item` 对象，存入 `trans` 中。只要读取成功，就执行 `while` 循环体。

`while` 的循环体是一个单个的 `if` 语句，它检查 ISBN 是否相等。如果相等，使用复合赋值运算符将 `trans` 加到 `total` 中。如果 ISBN 不等，我们打印保存在 `total` 中的值，并将其重置为 `trans` 的值。在执行完 `if` 语句后，返回到 `while` 的循环条件，读取下一条销售记录，如此反复，直至所有销售记录都处理完。

当 `while` 语句终止时，`total` 保存着文件中最后一个 ISBN 的数据。我们在语句块的最后一条语句中打印这最后一个 ISBN 的 `total` 值，至此最外层 `if` 语句就结束了。

## 1.6 节练习

练习 1.25：借助网站上的 `Sales_item.h` 头文件，编译并运行本节给出的书店程序。

## 小结

&lt; 26

本章介绍了足够多的 C++ 语言的知识，以使你能够编译、运行简单的 C++ 程序。我们看到了如何定义一个 main 函数，它是操作系统执行你的程序的调用入口。我们还看到了如何定义变量，如何进行输入输出，以及如何编写 if、for 和 while 语句。本章最后介绍了 C++ 中最基本的特性——类。在本章中，我们看到了，对于其他人定义的一个类，我们应该如何创建、使用其对象。在后续章节中，我们将介绍如何定义自己的类。

## 术语表

**参数 (实参, argument)** 向函数传递的值。

**赋值 (assignment)** 抹去一个对象的当前值，用一个新值取代之。

**程序块 (block)** 零条或多条语句的序列，用花括号包围。

**缓冲区 (buffer)** 一个存储区域，用于保存数据。IO 设施通常将输入（或输出）数据保存在一个缓冲区中，读写缓冲区的动作与程序中的动作是无关的。我们可以显式地刷新输出缓冲，以便强制将缓冲区中的数据写入输出设备。默认情况下，读 cin 会刷新 cout；程序非正常终止时也会刷新 cout。

**内置类型 (built-in type)** 由语言定义的类型，如 int。

**Cerr** 一个 ostream 对象，关联到标准错误，通常写入到与标准输出相同的设备。默认情况下，写到 cerr 的数据是不缓冲的。cerr 通常用于输出错误信息或其他不属于程序正常逻辑的输出内容。

**字符串字面值常量 (character string literal)** 术语 string literal 的另一种叫法。

**cin** 一个 istream 对象，用来从标准输入读取数据。

**类 (class)** 一种用于定义自己的数据结构及其相关操作的机制。类是 C++ 中最基本的特性之一。标准库类型中，如 istream 和 ostream 都是类。

**类类型 (class type)** 类定义的类型。类名即为类型名。

**clog** 一个 ostream 对象，关联到标准错误。默认情况下，写到 clog 的数据是被缓冲的。clog 通常用于报告程序的执行信息，存入一个日志文件中。

**注释 (comment)** 被编译器忽略的程序文本。C++ 有两种类型的注释：单行注释和界定符对注释。单行注释以 // 开始，从 // 到行尾的所有内容都是注释。界定符对注释以 /\* 开始，其后的所有内容都是注释，直至遇到 \*/ 为止。

**条件 (condition)** 求值结果为真或假的表达式。通常用值 0 表示假，用非零值表示真。

**cout** 一个 ostream 对象，用于将数据写入标准输出。通常用于程序的正常输出内容。

**花括号 (curly brace)** 花括号用于划定程序块边界。左花括号 ({) 为程序块开始，右花括号 (}) 为结束。

**数据结构 (data structure)** 数据及其上所允许的操作的一种逻辑组合。

**编辑-编译-调试 (edit-compile-debug)** 使程序能正确执行的开发过程。

**文件结束符 (end-of-file)** 系统特定的标识，指出文件中无更多数据了。

**表达式 (expression)** 最小的计算单元。一个表达式包含一个或多个运算对象，通常还包含一个或多个运算符。表达式求值会产生一个结果。例如，假设 i 和 j 是 int 对象，则 i+j 是一个表达式，它产生两个

&lt; 27

`int` 值的和。

**for 语句 (for statement)** 迭代语句，提供重复执行能力。通常用来将一个计算反复执行指定次数。

**函数 (function)** 具名的计算单元。

**函数体 (function body)** 语句块，定义了函数所执行的动作。

**函数名 (function name)** 函数为人所知的名字，也用来进行函数调用。

**头文件 (header)** 使类或其他名字的定义可被多个程序使用的一种机制。程序通过 `#include` 指令使用头文件。

**if 语句 (if statement)** 根据一个特定条件的值进行条件执行的语句。如果条件为真，执行 `if` 语句体。否则，执行 `else` 语句体（如果存在的话）。

**初始化 (initialize)** 在一个对象创建的时候赋予它一个值。

**iostream 头文件**，提供了面向流的输入输出的标准库类型。

**istream** 提供了面向流的输入的库类型。

**库类型 (library type)** 标准库定义的类型，28如 `istream`。

**main** 操作系统执行一个 C++ 程序时所调用的函数。每个程序必须有且只有一个命名为 `main` 的函数。

**操纵符 (manipulator)** 对象，如 `std::endl`，在读写流的时候用来“操纵”流本身。

**成员函数 (member function)** 类定义的操作。通常通过调用成员函数来操作特定对象。

**方法 (method)** 成员函数的同义术语。

**命名空间 (namespace)** 将库定义的名字放在一个单一位置的机制。命名空间可以帮助避免不经意的名字冲突。C++ 标准库定义的名字在命名空间 `std` 中。

**ostream** 标准库类型，提供面向流的输出。

**形参列表 (parameter list)** 函数定义的一部分，指出调用函数时可以使用什么样的实参，可能为空列表。

**返回类型 (return type)** 函数返回值的类型。

**源文件 (source file)** 包含 C++ 程序的文件。

**标准错误 (standard error)** 输出流，用于报告错误。标准输出和标准错误通常关联到程序执行所在的窗口。

**标准输入 (standard input)** 输入流，通常与程序执行所在窗口相关联。

**标准库 (standard library)** 一个类型和函数的集合，每个 C++ 编译器都必须支持。

标准库提供了支持 IO 操作的类型。C++ 程序员倾向于用“库”指代整个标准库，还倾向于用库类型表示标准库的特定部分，例如用“`iostream` 库”表示标准库中定义 IO 类的部分。

**标准输出 (standard output)** 输出流，通常与程序执行所在窗口相关联。

**语句 (statement)** 程序的一部分，指定了当程序执行时进行什么动作。一个表达式接一个分号就是一条语句；其他类型的语句包括语句块、`if` 语句、`for` 语句和 `while` 语句，所有这些语句内都包含其他语句。

**std** 标准库所使用的命名空间。`std::cout` 表示我们要使用定义在命名空间 `std` 中的名字 `cout`。

**字符串常量 (string literal)** 零或多个字符组成的序列，用双引号包围 ("a string literal")。

**未初始化的变量 (uninitialized variable)** 未赋予初值的变量。类类型的变量如果未指定初值，则按类定义指定的方式进行初始化。定义在函数内部的内置类型变量默认是不初始化的，除非有显式的初始化语句。试图使用一个未初始化变量的值是错误的。未初始化变量是 bug 的常见成因。

**变量 (variable)** 具名对象。

**while 语句 (while statement)** 迭代语句，提供重复执行直至一个特定条件为假的机制。循环体会执行零次或多次，依赖于循环条件求值结果。

**()运算符 (() operator)** 调用运算符。跟随着在函数名之后的一对括号 “()”，起到调用函数的效果。传递给函数的实参放置在括号内。

**++运算符 (++ operator)** 递增运算符。将运算对象的值加 1， $++i$  等价于  $i=i+1$ 。

**+=运算符 (+= operator)** 复合赋值运算符，将右侧运算对象加到左侧运算对象上； $a+=b$  等价于  $a=a+b$ 。

**.运算符 (. operator)** 点运算符。左侧运算对象必须是一个类类型对象，右侧运算对象必须是此对象的一个成员的名字。运算结果即为该对象的这个成员。

**::运算符 (:: operator)** 作用域运算符。其用处之一是访问命名空间中的名字。例如，`std::cout` 表示命名空间 `std` 中的名字 `cout`。

**=运算符 (= operator)** 将右侧运算对象的值赋予左侧运算对象所表示的对象。

**--运算符 (-- operator)** 递减运算符。将运算对象的值减 1， $--i$  等价于  $i=i-1$ 。

**<<运算符 (<< operator)** 输出运算符。将

右侧运算对象的值写到左侧运算对象表示的输出流：`cout << "hi"` 表示将 `hi` 写到标准输出。输出运算符可以连接：`cout << "hi" << "bye"` 表示将输出 `hibye`。

**>>运算符 (>> operator)** 输入运算符。从左侧运算对象所指定的输入流读取数据，存入右侧运算对象中：`cin >> i` 表示从标准输入读取下一个值，存入 `i` 中。输入运算符可以连接：`cin >> i >> j` 表示先读取一个值存入 `i`，再读取一个值存入 `j`。

**#include** 头文件包含指令，使头文件中代码可被程序使用。

**==运算符 (== operator)** 相等运算符。检测左侧运算对象是否等于右侧运算对象。

**!=运算符 (!= operator)** 不等运算符。检测左侧运算对象是否不等于右侧运算对象。

**<=运算符 (<= operator)** 小于等于运算符。检测左侧运算对象是否小于等于右侧运算对象。

**<运算符 (< operator)** 小于运算符。检测左侧运算对象是否小于右侧运算对象。

**>=运算符 (>= operator)** 大于等于运算符。检测左侧运算对象是否大于等于右侧运算对象。

**>运算符 (> operator)** 大于运算符。检测左侧运算对象是否大于右侧运算对象。