第 1 章

开　　始

本章介绍 C++语言的基本元素，包括内置数据类型、命名对象的定义、表达式、语句、命名函数的定义和使用。本章将给出一个最小的合法 C++程序，主要用它来讨论程序的编译过程、预处理，并将首次介绍 C++ 对输入/输出的支持。我们还将给出一些简单但完整的 C++程序。

1.1　问题的解决

程序常常是针对某些要解决的问题和任务而编写的。我们来看一个例子，一个书店将每本被卖出的书的书名和出版社，输入到一个文件中，这些信息以书被卖出的时间顺序输入，每两周店主将手工计算每本书的销售量，以及每个出版社的销售量。报表以出版社名称的字母顺序排列，以便下订单。现在，我们希望写一个程序来做这件事。

解决大问题的一种方法，是把它分割成大量的小问题。理想情况下，这些小问题可以很容易地被解决。然后，再把它们合在一起，就可以解决大问题了。如果新分割的小问题解决起来还是太大，就把它分割得再小一些，重复整个过程，直到能够解决每个小问题。这个策略就是***分治***（*divide and conguer*）和***逐步细化***(*stepwise refinement*)。书店问题可以分割成四个子问题（或任务）：

1．读销售文件；

2．根据书名和出版社计算销售量；

3．以出版社名称对书名进行排序；

4．输出结果。

我们知道怎样解决第 1、2 和 4 个子问题，因此它们不需要进一步分割。但是，第 3 个

子问题解决起来还是有些大，所以对这个子问题重复我们的做法，继续分割：

3a. 按出版社排序；

3b. 对每个出版社的书，按书名排序；

3c. 在每个出版社的组中，比较相邻的书名，如果两者匹配，增加第一个的数量，删除第二个。

3a、3b 和 3c 所代表的问题，现在都已经能够解决了。由于我们能够解决这些子问题，因此也就能够有效地解决原始的大问题了。而且，我们也知道任务的原始顺序是不正确的，正确的动作序列应该是：

1．读销售文件；

2．对文件排序——先按出版社，然后在出版社内部按书名排序；

3．压缩重复的书名；

4．将结果写入文件。

这个动作序列就是*算法(algorithm)*。下一步我们把算法转换成一种特定的程序设计语言——在这里是 C++语言。

1.2 C++程序

在 C++中，动作被称为*表达式(expression)*。以分号结尾的表达式被称作*语句(statement)*。C++中最小的程序单元是语句，在自然语言中，类似的结构就是句子。例如，下面是一组 C++语句：

```
int book_count = 0;
book_count = books_on_shelf + books_on_order;
cout << "the value of book_count: " << book_count;
```

第一条语句是一个*声明(declaration)*语句，book_count 被称为*标识符(identifier)*或*符号变量*（*symbolic variable*，简称*变量*），或者*对象(object)*，它定义了计算机内存的一块区域，并且与名字 book_count 相关联，被用来存储整数值。0 是一个*字面常量*(*literal constant*)，book_count 被*初始化*为 0。

第二条语句是一个*赋值*(*assignment*)语句，它把 books_on_shelf 和 books_on_order 的值相加，并把结果放入与 book_count 相关联的计算机内存区域中。这里假定 books_on_shelf 和 books_on_order 已经在前面的代码中被声明为整型，并赋了初值。

第三条是*输出*(*output*)语句，cout 是与用户终端相关联的输出目标。<<是输出操作符，该语句向 cout——即用户终端——先输出用引号括起来的字符串文字，然后输出存储在与名字 books_count 相关联的内存区域中的值。假设此时 books_count 中的值为 11273，那么输出结果为：

```
the value of book_count: 11273
```

把语句按逻辑语义分成组，形成一些有名字的单元，这些单元被称为*函数*(*function*)。例如，把所有需要读销售文件的语句组织到一个被称为 readIn()的函数中。类似地，我们可以

构成 sort()、compact()和 print()函数。

在 C++中，每个程序必须包含一个被称作 main()的函数，它是由程序员提供的，并且只有这样的程序才能运行。下面是前述算法的一种可能的 main()函数的定义方式：

```
int main()
{
  readIn();
  sort();
  compact();
  print();
  return 0;
}
```

C++程序从 main()函数的第一条语句开始执行，在本例中，程序从函数 readIn()开始。并且，程序按顺序执行 main()函数中的语句。在执行完 main()函数的最后一条语句之后，程序正常结束。

函数由四部分组成：返回类型、函数名、参数表，以及函数体。前三部分合起来被称为*函数原型*(*function prototype*)。

参数表由小括号括起来，包含一个或多个由逗号分开的参数。函数体被一对花括号括起来，由程序语句序列构成。

在本例中，main()函数的函数体*调用*(*invoke*)函数 readIn()、sort()、compact()和 print()。当这些函数调用都完成时，下面的语句

```
return 0;
```

被执行。return 是 C++预定义的语句，它提供了终止函数执行的一种方法。当 return 语句提供了一个值时，例如 0，这个值就成为函数的*返回值*(*return value*)。本例中，返回值为 0 表示 main()函数成功执行完毕。（标准 C++中，如果 main()函数没有显式地提供返回语句，则它缺省返回 0）。

现在我们来看一下，如果想让我们的程序能够执行起来，我们还需要做哪些准备工作。首先，必须提供函数 readIn()、sort()、compact()以及 print()的定义。下面的哑函数实例已经足够满足这个要求了。

```
void readIn()  { cout << "readIn()\n";  }
void sort()    { cout << "sort()\n";    }
void compact() { cout << "compact()\n"; }
void print()   { cout << "print()\n";   }
```

void 被用来指定一个没有返回值的函数。正如上面的定义所示，每个函数在被 main()函数调用时只会简单地在用户终端上显示它的存在，以后，我们会用真正的实现函数来代替这些哑函数。

这种渐进式生成程序的方法，为控制程序设计中不可避免的错误，提供了一种有效的控制手段。试图一下子就能写出一个完全成功的程序，几乎是不可思议的。

程序源文件的名字，一般包括两部分：文件名(例如 bookstore)以及文件后缀。文件后缀

一般用来标识文件的内容。文件

```
bookstore.h
```

在 C 或 C++中习惯上被称为*头*(*header*)文件。（标准 C++头文件没有后缀，这是个例外。）文件

```
bookstore.c
```

习惯上被当作 C 程序文本文件。但在 UNIX 操作系统中，文件

```
bookstore.C
```

习惯上被当作C++程序的文本文件。C++程序文件的后缀在不同的C++实现中是不同的，尤其是，在 DOS 中大写的字母 C 与小写的字母 c 是不能区分的。其他常用来识别 C++程序文本文件的后缀还包括：

```
bookstore.cxx
bookstore.cpp
```

类似地，头文件的后缀在 C++的不同实现中也不相同（这也是标准 C++没有指定头文件后缀的一个原因）。请查阅你的编译器的用户指南，以确定在当前平台上使用什么后缀。

使用某个文本编辑器，将下面这段完整的程序输入到一个 C++文本文件中。

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

void read()    { cout << "read()\n";    }
void sort()    { cout << "sort()\n";    }
void compact() { cout << "compact()\n"; }
void write()   { cout << "write()\n";   }

int main() {
    read();
    sort();
    compact();
    write();

    return 0;
}
```

iostream 是输入/输出流库标准文件（注意它没有后缀），它包含 cout 的信息，这对我们的程序是必需的。#include 是*预处理器指示符*(*preprocessor directive*)，它使 iostream 的内容被读到我们的文本文件中（1.3 节将讨论预处理器指示符）。

在 C++标准库中定义的名字，如 cout，不能在程序中直接使用，除非在预处理器指示符

```
#include <iostream>
```

后面加上语句

```
using namespace std;
```

这条语句被称作 *using 指示符*（*using directive*）。C++标准库中的名字被声明在一个称作 std 的名字空间中，这些名字在我们的程序文本文件中是不可见的，除非我们显式地使它们可见。using 指示符告诉编译器使用在名字空间 std 中声明的名字。（在 2.7 和 2.8 节中将更进一步讨论名字空间与 using 指示符。）[1]

一旦程序已经被输入到文件中，如 prog1.c，接下来就要编译它。在 Unix 系统中，按以下步骤进行（$表示系统提示符）。

```
$ CC prog1.C
```

用来调用 C++编译器的命令的名字，在不同的实现中也不相同（在 Windows 中，通常通过选择菜单项来调用命令）。CC 是 Unix 工作站上使用 C++编译器的命令名。你可以通过参考手册或系统管理员获得系统的 C++命令名。

编译器的一部分工作是分析程序代码的正确性。编译器不能判断程序的语义是否正确，但能够判断出程序形式(form)上的错误，下面是两个常见的程序形式错误：

1．语法错误。程序员犯了 C++语言的语法错误。例如：

```
int main ( {   // 错误：缺少 ')'
    readIn():  // 错误：非法字符 ':'
    sort();
    compact();
    print();

    return 0   // 错误：缺少 ';'
```

2．类型错误。在 C++中，每个数据项都有一个相关联的数据类型，例如，10 是一个整型数值。由双引号括起来的词“ hello ”是一个字符串。如果为一个需要整型参数的函数提供了一个字符串，编译器就会报告类型错误。

错误消息包含一个行号以及编译器对错误的简要描述。按报告的顺序逐一修正错误，是个好的习惯。一个简单的错误常常会有很多影响，会引起编译器报告比实际多得多的错误，因此，一旦错误被改正后，应当马上重新编译。这个循环过程通常被称为*编辑—编译—调试*(*edit-compile-debug*)。

编译器的第二部分工作是转换正确的程序代码。这种转换被称为*代码生成*(*code generation*)。典型情况下，它生成目标代码或汇编指令代码。这些代码能够被运行该程序的计算机所理解。

成功编译的结果是一个可执行文件。前面的程序执行时，其输出结果如下：

```
readIn()
sort()
compact()
```

[1] 在本书写作时刻(大约指 97 年前后——译注)，并不是所有的 C++实现都支持名字空间。如果你使用的 C++实现不支持名字空间，那么 using 指示符必须要忽略掉。因为本书的许多例子都是用不支持名字空间的 C++实现来编译的，所以绝大多数的代码例子都省略了 using 指示符。

```
print()
```

C++定义了一组内置的基本数据类型：整数类型（int）、浮点数类型（float）、字符类型（char）、以及只有 false 和 true 两个值的布尔类型（boolean）。每种类型都与 C++语言中某一个*关键字(keyword)*相关联，程序中的每个对象都与一个特定的类型相关联。例如，下面的代码：

```
int    age = 10;
double  price = 19.99;
char    delimiter = ' ';
bool    found = false;
```

定义了四个对象——age、price、delimiter 和 found——分别是整数类型、双精度浮点数类型、字符类型和布尔类型。每个类型都被提供了一个字面常量初始值：整数 10、浮点数 19.99、空格字符、布尔值 false。

在内置类型之间经常发生隐式的类型*转换*(*conversion*)。例如，将一个 double 双精度型的常量赋给一个 int 型的 age

```
age = 33.333;
```

实际上，赋给 age 的是被截断后的整数值 33。（这些*标准转换*(*standard conversion*)以及一般类型的转换将在 4.14 节中详细讨论。）

C++标准库还提供了一组扩展的基本数据类型。其中包括字符串（string）、复数（complex number）、向量（vector）和列表（list）。例如：

```
// 为了使用string对象，下面的头文件是必需的
#include <string>
string current_chapter = "Getting Started";

// 为了使用vector对象，下面的头文件是必需的
#include <vector>
vector<string> chapter_titles( 20 );
```

current_chapter 是一个字符串对象，被初始化为字符串文字“ Getting Started ”。chapter_title 是一个向量，包含有 20 个字符串类型的元素。特殊的语法

```
vector<string>
```

指示编译器创建一个能够存放字符串元素的向量类型。要定义一个能够存放 20 个整数的向量对象，我们可以这样写：

```
vector<int> ivec( 20 );
```

（本书对向量还有更多的描述。）

无论是一种语言还是它的标准库，都不可能提供实际程序设计环境要求的所有数据类型，因此，现代语言都提供了类型定义工具设施，使我们能够在语言中引入新的类型，这些类型的用法与内置类型的用法一样方便。在 C++中，这种设施就是类机制。在 C++中，像

string、complex、vector、list 这样的标准库类型都被设计成类。实际上， 输入/输出库也是这样的。

类设施可能是 C++中最重要的组成部分，第 2 章对整个类机制作了基本的概述性介绍。

1.2.1　程序流程控制

缺省情况下，语句将按顺序执行。例如，在前面的程序中（重新列在下面），read()总是先被执行，然后是 sort()、compact()、write()。

```
int main()
{
 read();
 sort();
 compact();
 write();

 return 0;
}
```

然而，如果销售进展得很慢，例如，只有 0 或 1 项，那么就没有必要排序和压缩了，但是我们仍然需要输出这一项销售记录，或者指出没有销售记录产生。通过条件语句 if 我们可以完成这项工作。（假设已经重写了 readIn()函数，使其能够返回读入的项数。）

```
// read() returns the number of entries read
// its return value is of type int
int read() { ... }

// ...

int main()
{
 int count = read();

 // if number of entries read is greater than 1
 // then sort() and compact()

 if ( count > 1 ) {
      sort();
      compact();
 }

 if ( count == 0 )
      cout << "no sales for this month\n";
 else write();

 return 0;
}
```

第一个 if 语句给出了在括号中的条件表达式为真的情况下应该执行的动作。在这个被修改过的程序中，只有在 count 大于 1 的时候 sort()、compact()函数才会被调用。在第二个 if 语句中，有两个执行分支。如果条件为真——在这里，即如果 count 等于 0——则简单地输出没有销售产生，否则，只要 count 不等于 0，就调用 write()。（我们将在 5.3 节中详细讨论 if 语句。）

第二种非顺序执行的语句是迭代或*循环*(*loop*)语句。当条件保持为真的时候，循环重复执行一条或多条语句。例如：

```
int main()
{
  int iterations = 0;
  bool continue_loop = true;
  while ( continue_loop != false )
  {
     iterations++;

     cout << "the while loop has executed "
          << iterations << " times\n";

     if ( iterations == 5 )
        continue_loop = false;
  }

  return 0;
}
```

在这个看似人为构造的例子中，*while* 循环执行 5 次，直到 iterations 等于 5 并且 continum_loop 被赋值为 false。语句

```
iterations++;
```

对 iterations 加 1。在 1.5 节中将有更实际的 while 循环的例子。第 15 章将详细讲解循环语句。

1.3 预处理器指示符

头文件通过 *include* ***预处理器指示符***（*preprocessor include directive*）而成为我们程序的一部分。预处理器指示符由“ # ”号标示出来，并且这个符号放在程序中该行的最起始列上。处理这些指示符的程序被称做***预处理器***（*preprocessor*）（通常捆绑在编译器中）。

#include 指示符读入指定文件的内容，它有两种格式：

```
#include <some_file.h>
#include "my_file.h"
```

如果文件名用尖括号“〈，〉”括起来，表明这个文件是一个工程或标准头文件，查找过程会检查预定义的目录。我们可以通过设置搜索路径环境变量或命令行选项来修改这些目录。（在不同的平台上这些方法大不相同，建议你请教同事或查阅编译器手册以获得更进一步的信息）。如果文件名用一对引号括起来，则表明该文件是用户提供的头文件，查找该文件时将从当前文件目录开始。

被包含的文件还可以含有#include 指示符。由于嵌套包含文件的原因，一个头文件可能会被多次包含在一个源文件中。条件指示符可防止这种头文件的重复处理。例如：

```
#ifndef BOOKSTORE_H
#define BOOKSTORE_H
   /* Bookstore.h contents go here */
#endif
```

条件指示符

```
#ifndef
```

检查 BOOKSTORE_H 在前面是否已经被定义。这里，BOOKSTORE_H 是一个预编译器常量。（习惯上预编译器常量往往被写成大写字母）。如果 BOOKSTORE_H 在前面没有被定义，则条件指示符的值为真，于是从#ifndef 到#endif 之间的所有语句都被包含和处理。相反，如果#ifndef 指示符的值为假，则它与#endif 指示符之间的行将被忽略。

为了保证头文件只被处理一次，把#define 指示符

```
#define BOOKSTORE_H
```

放在#ifndef 后面，这样在头文件的内容第一次被处理时，BOOKSTORE_H 被定义，从而防止了在程序文本文件中以后#ifndef 指示符的值为真。

只要不存在“两个必须包含的头文件要检查一个同名的预处理器常量”的情形，这个策略就能够很好地运作。

#ifdef 指示符常被用来判断一个预处理器常量是否已被定义，以便有条件地包含程序代码。例如：

```
int main()
{
#ifdef DEBUG
  cout << "Beginning execution of main()\n";
#endif

  string word;
  vector< string > text;

  while ( cin >> word )
  {
#ifdef DEBUG
     cout << "word read: " << word << "\n";
#endif
     text.push_back( word );
```

```
  }

  // ...
```

本例中，如果没有定义 DEBUG，实际被编译的程序代码如下：

```
int main()
{
  string word;
  vector< string > text;

  while ( cin >> word )
  {
         text.push_back( word );
  }

  // ...
}
```

反之，如果定义了 DEBUG，则传给编译器的程序代码是：

```
int main()
{
  cout << "Beginning execution of main()\n";

  string word;
  vector< string > text;

  while ( cin >> word )
  {
      cout << "word read: " << word << "\n";
      text.push_back( word );
  }

  // ...
}
```

我们在编译程序时可以使用-D 选项，并且在后面写上预处理器常量的名字，这样就能在命令行中定义预处理器常量：[2]

```
$ CC -DDEBUG main.C
```

也可以在程序中用#define 指示符定义预处理器常量。

编译 C++程序时，编译器自动定义了一个预处理器名字__cplusplus（注意前面有两个下划线）。因此，我们可以根据它来判断该程序是否是 C++程序，以便有条件地包含一些代码。例如：

[2] 对于 UNIX 系统，确实是这样的。Windows 程序员应该检查一下编译器的用户指南。

```
#ifdef __cplusplus
  // ok: we're compiling C++
  // we'll explain extern "C" in Chapter 7!
  extern "C"
#endif
  int min( int, int );
```

在编译标准 C 时，编译器将自动定义名字__STDC__。当然，__cplusplus 与__STDC__不会同时被定义。

另外两个比较有用的预定义名字是：__LINE__和__FILE__。__LINE__记录文件已经被编译的行数，__FILE__包含正在被编译的文件的名字。可以这样使用它们：

```
if ( element_count == 0 )
     cerr << "Error: " << __FILE__
          << " : line " << __LINE__
          << "element_count must be non-zero.\n";
```

另外两个预定义名字分别包含当前被编译文件的编译时间(__TIME__)和日期（__DATE__）。时间格式为 hh:mm:ss，因此如果在上午 8 点 17 分编译一个文件，则时间表示为 08:17:05。如果这一天是 1996 年 10 月 31 日，星期四，则日期表示为

```
Oct  31  1996
```

若当前处理的行或文件发生变化，则__LINE__和__FILE__的值将分别被改变，其他四个预定义名字在编译期间保持不变。它们的值也不能被修改。

assert()是 C 语言标准库中提供的一个通用预处理器宏。在代码中常利用 assert()来判断一个必需的前提条件，以便程序能够正确执行。例如，假定我们要读入一个文本文件，并排序其中的词，必需的前提条件是文件名已经提供给我们，这样我们才能打开这个文件。为了使用 assert()，必须包含与之相关联的头文件：

```
#include <assert.h>
```

下面是一个简单的使用示例：

```
assert( filename != 0 );
```

assert()测试 filename 不等于 0 的条件是否满足。这表示，为了后面的程序能够正确执行，我们必须断言一个必需的前提条件。如果这个条件为假——即：filename 等于 0——断言失败，则程序将输出诊断消息，然后终止。

assert.h 是 C 库头文件的 C 名字，C++程序可以通过 C 库的 C 名字或 C++名字来使用它。这个头文件的 C++名字是 cassert。C 库头文件的 C++名字总是以字母 C 开头，后面是去掉后缀.h 的 C 名字。（正如前面所解释的，由于在各种 C++实现中，头文件的后缀各不相同，因此标准 C++头文件没有指定后缀）。

使用头文件的 C 名字，或者 C++名字，两种情况下头文件的#include 预处理器指示符的影响也会不同。下面的#include 指示符

```
#include <cassert>
```

导致 cassert 的内容被读入到我们的文本文件中。但是由于所有的 C++库名字是在名字空间 std 中被定义的，因而在我们的程序文本文件中，它们是不可见的，除非用下面的 using 指示符显式地使其可见：

```
using namespace std;
```

使用 C 头文件的#include 指示符，

```
#include <assert.h>
```

就可以直接在程序文本文件中使用名字 assert()，而无需使用 using 指示符。[3]（库文件厂商用名字空间来控制全局名字空间污染(即名字冲突)问题，以避免它们的库污染了用户程序的名字空间。8.5 节将讨论这些细节。）

1.4 注　　释

注释用来帮助程序员读程序，它是一种程序礼仪，可以用来概括程序的算法、标识变量的意义、或者阐明一段比较难懂的程序代码。注释不会增加程序的可执行代码的长度。在代码生成以前，编译器会将注释从程序中剔除掉。

C++中有两种注释符号，一种是注释对(/*，*/)，与 C 语言中的一样。注释的开始用/*标记，编译器会将/*与*/之间的代码当作注释。注释可以放在程序的任意位置，可以含有制表符（tab）、空格或换行，还可以跨越多行程序。例如：

```
/*
 * This is a first look at a C++ class definition.
 * Classes are used both in object-based and
 * object-oriented programming.  An implementation
 * of the Screen class is presented in Chapter 13.
 */

class Screen {
  /* This is referred to as the class body */
public:
  void home();   /* move cursor to 0,0 */
  void refresh();   /* redraw Screen     */
private:
  /* Classes support “information hiding”.     */
  /* Information hiding restricts a program's  */
  /* access to the internal representation of  */
  /* a class (its data).  This is done through */
  /* use of the “private:” label             */
  int height, width;
```

[3] 在本书写作时刻，并不是所有的 C++实现都支持 C 库头文件的 C++名字。因为本书的许多例子是在不支持 C++头文件名的实现中编译的，所以有时候例子代码会用 C 名字引用 C 库头文件，而有时候用 C++名字引用 C 库头文件。

```
};
```

在代码中混杂过多的注释会使程序更难于理解。例如，注释几乎淹没了 width 和 height 的声明。通常，把注释放在要描述的文本之上比较合适。与其他软件文档一样，考虑到有效性问题，注释必须随着软件的发展而升级。但是，注释与所描述的代码随时间推移而越离越远的情况却是经常发生的。

注释对不能嵌套，即一个注释对不能出现在另外一个注释对之中。请尝试在系统中编译下面的程序，它会使大多数编译器无法正常处理：

```
#include <iostream>

/*
 * comment pairs /* */ do not nest.
 * "do not nest" is considered source code,
 * as are both these lines and the next.
 */

int main() {
    cout << "hello, world\n";
}
```

解决这种嵌套注释的一个办法是在星号和斜线之间加一个空格。

```
/* * /
```

对于星号和斜线序列，只有当这两个字符之间没有被空格分割时，它们才被看作是注释符。

第二种注释符是双斜线（//），它可用来注释一个单行，程序行中注释符右边的内容都将被当作注释而被编译器忽略。例如，下面的 Screen 类使用了两种注释：

```
/*
 * This is a first look at a C++ class definition.
 * Classes are used both in object-based and
 * object-oriented programming.  An implementation
 * of the Screen class is presented in Chapter 13.
 */

class Screen {
  // This is referred to as the class body
public:
  void home(); // move cursor to 0,0
  void refresh();  // redraw Screen
private:
  /* Classes support “information hiding”.     */
  /* Information hiding restricts a program's  */
  /* access to the internal representation of  */
  /* a class (its data).  This is done through */
  /* use of the “private:” label              */
```

```
    // private data goes here ...
};
```

大多数程序往往包含两种格式的注释。多行的说明通常被放在注释对中，半行或单行的注释则由双斜线指出。

1.5 输入/输出初步

C++的输入/输出功能由输入/输出流（iostream）库提供。输入/输出流库是C++中一个面向对象的类层次结构，也是标准库的一部分。

终端输入，也被称为*标准输入*(*standard input*)，与预定义的 iostream 对象 cin（发音为 see-in）绑定在一起。直接向终端输出，也被称为*标准输出*(*standard output*)，与预定义的 iostream 对象 cout（发音为 see-out）绑定在一起。cerr（发音为 see-err）典型地用来产生给程序用户的警告或错误信息。

任何要想使用 iostream 库的程序必须包含相关的系统头文件：

```
#include <iostream>
```

输出操作符<<用来将一个值导向到标准输出或标准错误上。例如：

```
int v1, v2;
// ...
cout << "The sum of v1 + v2 = ";
cout << v1 + v2;
cout << '\n';
```

双字符序列“\n”表示换行符。输出换行符时，它结束当前的行，并将随后的输出导向到下一行。除了显式地使用换行符外，我们还可以使用预定义的 iostream *操纵符*(*manipulator*)endl。

操纵器在 iostream 上执行的是一个操作，而不只是简单地提供数据。例如，endl 在输出流中插入一个换行符，然后刷新输出缓冲区。我们一般不写：

```
cout << '\n';
```

而是写

```
cout << endl;
```

（预定义 iostream 操纵器将在第 20 章中讨论。）

连续出现的输出操作符可以连接在一起，例如：

```
cout << "The sum of v1 + v2 = " << v1 + v2 << endl;
```

连续的输出操作符按顺序应用在 cout 上。为了便于阅读，连接在一起的输出操作符，可以分写在几行。下面的三行组成一条输出语句：

```
cout << "The sum of "
     << v1 << " + "
```

```
    << v2 << " = " << v1 + v2 << endl;
```

类似地，输入操作符（>>）用来从标准输入读入一个值。例如：

```
string file_name;
// ...
cout << "Please enter the file to be opened: ";
cin >> file_name;
```

连续出现的输入操作符，也可以连接起来。例如：

```
string ifile, ofile;
// ...
cout << "Please enter input and output file names: ";
cin >> ifile >> ofile;
```

怎样读入未知个数的输入值呢？在 1.2 **节结束的时候，我们已经做过。请看下面的代码序列**

```
string word;
while ( cin >> word )
      // ...
```

在 while **循环中，每次迭代都从标准输入读入一个字符串，直到所有的串都读进来。当到达文件结束处**(end-of-file)**时，条件**

```
( cin >> word )
```

为假（第 20 **章将解释这是如何发生的）。下面是使用这段代码序列的一个例子：**

```
#include <iostream>
#include <string>

int main()
{
  string word;

  while ( cin >> word )
          cout << "word read is: " << word << '\n';

  cout << "ok: no more words to read: bye!\n";
  return 0;
}
```

下面是 Janmes Jouce **的小说** Finnegans Wake **的前五个词：**

```
riverrun, past Eve and Adam's
```

从键盘上输入这些词，程序的输出是：

```
word read is: riverrun,
word read is: past
word read is: Eve
word read is: and
word read is: Adam's
word read is: ok: no more words to read: bye!
```

（在第 6 章，我们将会看到怎样从各种输入字符串中删除标点符号。）

1.5.1 文件输入和输出

iostream **库也支持文件的输入和输出。所有能应用在标准输入和输出上的操作符，也都可以应用到已经被打开的输入或输出(或两者兼有)文件上。为了打开一个文件供输入或输出，除了** iostream **头文件外，还必须包含头文件**

```
#include <fstream>
```

为了打开一个输出文件，我们必须声明一个 ofstream **类型的对象：**

```
ofstream outfile( "name-of-file" );
```

为了测试是否已经成功地打开了一个文件，我们可以写出这样的代码：

```
// evaluates to false if file failed to open
if ( ! outfile )
   cerr << "Sorry! We were unable to open the file!\n";
```

类似地，为了打开一个文件供输入，我们必须声明一个 ifstream **类型的对象：**

```
ifstream infile( "name of file" );
if ( ! infile )
   cerr << "Sorry! We were unable to open the file!\n";
```

下面是一个简单的程序。它从一个名为 in_file **的文本文件中读取单词，然后把每个词写到一个名为** out_file **的输出文件中，并且每个词之间用空格分开。**

```
#include <iostream>
#include <fstream>
#include <string>
int main()
{
 ofstream outfile( "out_file" );
 ifstream infile( "in_file" );

 if ( ! infile ) {
      cerr << "error: unable to open input file!\n";
      return -1;
 }
```

```
    if ( ! outfile ) {
         cerr << "error: unable to open output file!\n";
         return -2;
     }

     string word;
     while ( infile >> word )
           outfile << word << ' ';

     return 0;
}
```

第 20 章将对 iostream 库进行全面的讨论，包括文件输入和输出。现在，我们对 C++提供的内容有了大致的了解，下一步，我们将通过使用类和模板设施把新的类型引入到语言中。