第20章　C++输入/输出系统基础

C++支持两个完备的I/O（输入/输出）系统：一个是从C继承而来的系统，另一个是C++定义的面向对象的I/O系统（后面简单地将其称为C++ I/O系统）。本书在第一部分讨论过基于C的I/O系统，这里将开始讨论C++ I/O系统。和基于C的I/O一样，C++的I/O系统是一个完全集成的系统，该系统的不同方面（例如，控制台I/O和磁盘I/O）实际上只是从不同角度看待同一种机制。本章将讨论C++ I/O系统的基本原理。虽然本章的例子使用的是“控制台”I/O系统，但涉及的内容同样适用于其他设备，其中也包括磁盘文件（详见第21章的讨论）。

由于从C继承的I/O系统的功能非常丰富、灵活和强大，所以你不免会对为什么C++还要定义另一个系统感到奇怪。这是因为C的I/O系统对对象一无所知，因此对C++而言，为了给面向对象的编程提供完全的支持，必须创建可以操作用户定义的对象的I/O系统。除了支持对象外，在并未使用用户定义的对象的程序中采用C++的I/O系统也有许多好处。坦白地说，应该在所有新代码中使用C++ I/O子系统，因为C++支持C I/O只是为了保持兼容性。

本章将讨论如何对数据进行格式化，如何重载<<和>> I/O运算符，使其可用于我们创建的类，以及如何创建称为操作算子的I/O函数，以提高程序效率。

20.1　老的C++ I/O与现代的C++ I/O

目前，有两种版本的面向对象的C++ I/O库被普遍使用，一种是基于最初的C++规范的老版本，另一种是标准C++定义的新版本。老的I/O库为头文件<iostream.h>所支持，新的I/O库为头文件<iostream>所支持。对编程人员来说，这两个库中的大部分内容看上去都相同，这是因为从本质上说，新的I/O库只是老I/O库的更新和改良版。事实上，二者之间的主要不同体现在内部的实现方法上，而不是体现在它们的用法上。

从编程者的角度看，新老两种版本的C++ I/O库之间有两个主要的不同：首先，新的I/O库包含少许附加功能并定义了一些新的数据类型，所以从本质上说，新的I/O库是老I/O库的超级。使用新库时，原来针对老的I/O库编写的所有程序几乎不必进行任何实质性的改变就可以被编译；其次，老的I/O库采用全局名字空间，新的I/O库采用std名字空间（回忆一下，所有标准C++库都使用std名字空间）。由于老的I/O库已经过时，所以本书只介绍新的I/O库，但所涉及的大部分内容同样适用于老的I/O库。

20.2　C++的流

像基于C的I/O系统一样，C++ I/O系统通过流进行操作。本书曾在第9章详细介绍过流，所以这里不再重复，然而我们可以把流归纳如下：一个流是一种既可以产生信息又可以消耗信息的逻辑设备，它通过I/O系统与一个物理设备相连。尽管流所连接的物理设备可以完全不同，但是所有的流以同样的方式运作。因为所有流的运作方式相同，所以实际上可以利用同样的I/O函数操作所有类型的物理设备。例如，可以利用同样的函数把信息写到文件或写到打印机和屏

幕。采用这种方法的优点是只需要掌握一种I/O系统。

20.3 C++的流类

正如前面讲到的那样，标准C++在头文件<iostream>中提供了对I/O系统的支持，在这个头文件中定义了一套相当复杂的类层次结构以支持I/O操作。I/O类以一个模板类的系统开始，本书曾在第18章介绍过：一个模板类定义一个类的形式，而不必指定将要操作的数据。一旦定义了一个模板类，就可以创建该类的具体实例。由于模板类与I/O库有关，所以标准C++创建了两种I/O模板类的说明：一个针对8位字符，另一个针对宽字符。本书只使用8位字符类，因为到目前为止，它们是最常用的类，但是同样的技术也适用于非8位的字符类。

C++ I/O系统建立在两个既相关又不同的模板类层次结构之上。第一个类层次结构是从低级I/O类派生而来的，称为basic_streambuf。这个类提供基本的低级输入/输出操作，而且还对整个C++ I/O系统提供底层支持。除非你正在进行高级的I/O编程，否则不必直接使用basic_streambuf。最常使用的类层次结构是从basic_ios派生而来的，它是一个高级I/O类，可以提供格式化、错误检查和与I/O流有关的状态信息（一个basic_ios的基类称为ios_base，它定义了几个为basic_ios所用的非模板特性）。basic_ios用做几个派生类的基类，这些派生类包括basic_istream，basic_ostream和basic_iostream，利用这些类可以分别创建输入流、输出流和输入/输出流。

前面曾经介绍过，I/O库创建了两个模板类层次结构的说明：一个用于8位字符，另一个用于宽字符。下面的列表反映了模板类名与它们的字符和宽字符之间的映射关系。

模板类	基于字符的类	基于宽字符的类
basic_streambuf	streambuf	wstreambuf
basic_ios	ios	wios
basic_istream	istream	wistream
basic_ostream	ostream	wostream
basic_iostream	iostream	wiostream
basic_fstream	fstream	wfstream
basic_ifstream	ifstream	wifstream
basic_ofstream	ofstream	wofstream

因为基于字符的名字在程序中经常用到，所以这些名字将在本书后面使用。由于老的I/O库也使用这些名字，所以新老I/O库在源代码级可以兼容。

最后要说明的一点是：ios类包含许多成员函数和变量，它们可以监控基本的流操作，而且将被频繁地引用。记住，如果在程序中包括了头文件<iostream>，就可以访问这个重要的类。

20.3.1 C++的预定义流

一个C++程序开始执行时将自动打开四个内置流，这些内置流是：

流	含义	默认设备
cin	标准输入	键盘
cout	标准输出	屏幕
cerr	标准错误输出	屏幕
clog	cerr的缓冲版本	屏幕

cin，cout 和 cerr 流与 C 的 stdin，stdout 和 stderr 相对应。

默认情况下，标准流用来与控制台通信。然而，在支持 I/O 重定向的环境中（例如，DOS，Unix，OS/2 和 Windows），标准流可以被重新定向到其他设备或文件。为简单起见，假设本章的例子没有出现 I/O 重定向的情况。

标准 C++ 还定义了四个附加流：win，wout，werr 和 wlog，它们都是宽字符版本的标准流。宽字符的类型是 wchar_t，一般为 16 位。宽字符用于存放与一些自然语言有关的大型字符集。

20.4　格式化的 I/O

C++ I/O 系统允许你对 I/O 操作进行格式化。例如，可以设置域宽度、指定数字基数以及决定显示小数点后面的数字位数。有两种相互关联但概念上不同的格式化数据的方式。第一种方式可以直接访问 ios 类的成员，确切地说，可以设置 ios 类内定义的各种格式状态标记或调用各种 ios 成员函数；第二种方式可以使用称为操纵算子（manipulator）的特殊函数，该函数是 I/O 表达式的一部分。

下面我们将讨论使用 ios 成员函数和标记的格式化的 I/O。

20.4.1　利用 ios 成员进行格式化

每一个流都与一组格式标记相关联，这组标记可以控制信息格式化的方式。ios 类声明了一个称为 fmtflags 的位屏蔽枚举，其中定义了下面一些值（从技术上讲，这些值是在 ios_base 中定义的，我们在前面曾经讲过，ios_base 是 ios 的基类）。

adjustfield	basefield	boolalpha	dec
fixed	floatfield	hex	internal
left	oct	right	scientific
showbase	showpoint	showpos	skipws
unitbuf	uppercase		

这些值用于设置或清除格式标记。如果使用的是老式的编译器，那么可能没有定义 fmtflags 枚举类型。在这种情况下，格式标记将被编码为一个长整型。

设置 skipws 时，开始的空白字符（空格、制表键和换行）在输入流的时候将被丢弃；清除 skipws 时，则不放弃空白字符。

当把标记设置为 left 时，输出呈左对齐；当设置为 right 时，输出呈右对齐；当设置为 internal 时，通过在正号（或负号）与数字之间插入空格，可以对数值进行填充以使其充满一个域；如果没有设置标记，输出默认为右对齐。

默认情况下，数值以十进制的形式输出，然而也可以改变数字的基数。通过设置 oct 标记可以使输出以八进制形式显示，设置 hex 标记可以使输出以十六进制形式显示。为了使输出返回到十进制，可以通过设置 dec 标记实现。

通过设置 showbase 可以显示数字的基数。例如，如果转换基数是十六进制，则数值 1F 将显示为 0x1F。

默认情况下，当按照科学计数法显示时，e 为小写；当按照十六进制值显示时，x 为小写。当设置为 uppercase 时，这些字符将以大写形式显示。

通过设置showpos可以显示正数前面的加号，设置showpoint可以在输出所有浮点数时显示十进制小数点和尾随的0，无论是否需要。

通过设置scientific标记，可以用科学计数法显示浮点数；当设置成fixed时，则用普通计数法显示浮点数；当两种标记都未被设置时，编译器会选择一种适当的方法显示。

当设置unitbuf时，缓冲区在每次插入操作之后都会被刷新。

当设置boolalpha时，可以利用关键字true和false输入或输出布尔值。

因为要经常用到oct，dec和hex域，所以可以将它们通称为basefield（基域）。同样，left，right和internal也可以称为adjustfield（对齐域）。最后，scientific和fixed域可以称为floatfield（浮点域）。

20.4.2 设置格式标记

为了设置一个标记，可以使用setf( )函数。这个函数是一个ios成员，其最常用的形式如下所示：

```
fmtflags setf(fmtflags flags);
```

这个函数返回格式标记先前的设置并开启由flags指定的那些标记。例如，为了开启showpos标记，可以使用下面的语句：

```
stream.setf(ios::showpos);
```

这里，stream是希望影响的流。注意用来限定showpos的ios::的用法。由于showpos是ios类定义的枚举常量，所以使用时必须用ios加以限定。

下面的程序利用开启的showpos和showpoint标记显示数值100：

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

int main()
{
  cout.setf(ios::showpoint);
  cout.setf(ios::showpos);

  cout << 100.0; // displays +100.000

  return 0;
}
```

setf( )是ios类的成员函数并且对该类创建的流产生影响，理解这一概念非常重要。对setf( )的所有调用都与一个特定的流相关，setf( )不能被自己调用，换句话说，在C++中没有全局格式状态的概念，每一个流都单独维护自己的格式状态信息。

尽管从技术上说，前面的程序没有任何错误，但是我们却可以采用一种更加有效的方法。我们可以简单地利用OR（或）运算符把想要设置的标记值放在一起，而不是多次调用setf( )。例如，下面仅通过一个调用就可以实现与上面程序同样的功能：

```
// You can OR together two or more flags,
cout.setf(ios::showpoint | ios::showpos);
```

注意： 因为格式标记是在 ios 类中定义的，所以必须使用 ios 和作用域运算符访问它们的值。例如，单独的 showbase 不能被识别，要想被识别，必须指定 ios::showbase。

20.4.3 清除格式标记

与 setf( )互补的是 unsetf( )，这个 ios 成员函数用于清除一个或多个格式标记。该函数的一般形式如下所示：

```
void unsetf(fmtflags flags);
```

flags 指定的标记将被清除（所有其他标记则不受影响）。

下面的程序说明了 unsetf( )的用法。该程序先是设置了 uppercase 和 scientific 标记，然后用科学计数法输出100.12，此时，科学计数法使用的是大写的“E”。接下来，该程序清除了uppercase 标记并再次以科学计数法用小写的 “e” 输出 100.12。

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

int main()
{
  cout.setf(ios::uppercase | ios::scientific);

  cout << 100.12; // displays 1.001200E+02

  cout.unsetf(ios::uppercase); // clear uppercase

  cout << " \n" << 100.12; // displays 1.001200e+02

  return 0;
}
```

20.4.4 setf( )的重载形式

setf( )有一种重载形式，其一般语法格式如下：

```
fmtflags setf(fmtflags flags1, fmtflags flags2);
```

在这个版本中，只有由flags2指定的标记会受到影响。这些标记先是被清除，然后根据flags1 指定的标记进行设置。注意，即使 flags1 含有其他标记，也只有被 flags2 指定的那些标记会受到影响。先前的标记设置将被返回。例如：

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

int main( )
{
  cout.setf(ios::showpoint | ios::showpos, ios::showpoint);

  cout << 100.0; // displays 100.000, not +100.000

  return 0;
}
```

这里，showpoint 被设置，而 showpos 不被设置，这是因为后者没有在第二个参数中指定。具有两个参数的 setf( )在设置基数、对齐方式和格式标记时最为常用。我们在前面讲过，

oct，dec和hex域可以统称为basefield。同样，left，right和internal域可以称为adjustfield。最后，scientific和fixed域可以称为floatfield。因为包含这些分组的标记互相排斥，因此在设置一个标记时必须关闭另一个标记。例如，下面的程序将输出设置为十六进制。为了以十六进制的形式输出，一些实现要求除hex标记开启之外，其他的基数标记都要关闭，为此，最简单的方法就是利用两个参数的setf()。

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

int main()
{
  cout.setf(ios::hex, ios::basefield);

  cout << 100; // this displays 64

  return 0;
}
```

这里，basefield标记（即：dec，oct和hex）先是被清除，然后hex标记被设置。

记住，只有在flags2中指定的标记可以被flags1指定的标记影响。例如，在下面的程序中，第一次设置showpos标记的尝试以失败告终。

```
// This program will not work.
#include <iostream>
using namespace std;

int main()
{
  cout.setf(ios::showpos, ios::hex); // error, showpos not set

  cout << 100 << '\n'; // displays 100, not +100

  cout.setf(ios::showpos, ios::showpos); // this is correct

  cout << 100; // now displays +100

  return 0;
}
```

记住，人们在大多数情况下还是希望用unsetf()清除标记，用只有一个参数的setf()（前面介绍过）设置标记。setf()的setf(fmtflags,fmtflags)版本只是在一些特殊情形下（例如，设置基数）才会使用。还有一种情况比较适于使用这种版本，即使用一个标记模板，该模板指定所有的标记状态，其中只希望改变一个或两个标记状态。在这种情况下，可以在flags1中指定模板并利用flags2指定哪些标记将受到影响。

20.4.5 检查格式标记

有时我们只是想知道当前的格式设置，而不是想修改任何设置。为了达到这个目的，ios将成员函数flags()包括在内，该函数只返回每个格式标记的当前设置。其原型如下所示：

```
fmtflags flags();
```

下面的程序利用flags()显示与cout相关的格式标记设置。这里，要格外注意showflags()函数，读者会发现它对编写程序很有帮助。

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

void showflags() ;

int main()
{
  // show default condition of format flags
  showflags();

  cout.setf(ios::right | ios::showpoint | ios::fixed);

  showflags();

  return 0;
}

// This function displays the status of the format flags.
void showflags()
{
  ios::fmtflags f;
  long i;

  f = (long) cout.flags(); // get flag settings

  // check each flag
  for(i=0x4000; i; i = i >> 1)
    if(i & f) cout << "1 ";
    else cout << "0 ";

  cout << " \n";
}
```

程序的输出如下所示（准确的输出结果因编译器的不同而不同）：

```
0 0 0 0 0 1 0 0 0 0 0 0 0 0 1
0 1 0 0 0 1 0 1 0 0 1 0 0 0 1
```

20.4.6　设置所有标记

flags( )函数还有一种形式，它可以使你设置与某个流相关的所有格式标记。这种flags( )版本的原型如下所示：

```
fmtflags flags(fmtflags f);
```

当使用这种版本时，f中的位模式用于设置与流相关的格式标记，因而所有格式标记都将受到影响。该函数返回先前的设置。

下面的程序说明了这种版本的 flags( )。它首先构造一个开启 showpos，showbase，oct 和 right的标记掩码并关闭所有其他标记，然后利用flags( )把与cout相关的格式标记按照这些设置设定。函数 showflags( )可以检验相应的标记（该函数与前面程序中使用的函数相同）。

```
#include <iostream>
using namespace std;
void showflags();

int main()
```

```
{
  // show default condition of format flags
  showflags();

  // showpos, showbase, oct, right are on, others off
  ios::fmtflags f = ios::showpos | ios::showbase | ios::oct | ios::right;
  cout.flags(f); // set all flags

  showflags();

  return 0;
}
```

20.4.7 使用 width( )，precision( )和 fill( )

除格式化标记之外，ios还定义了三个成员函数，这些函数可以设置以下格式参数：域宽、精度和填充字符，完成这些工作的函数分别是width( )，precision( )和fill( )。下面依次介绍这些函数。

默认情况下，当输出一个值时，它所占据的空间只是显示时占据的字符个数。然而，可以利用width( )函数指定一个最小域宽。该函数的原型如下所示：

```
streamsize width(streamsize w);
```

这里，w是将要改成的域宽，先前的域宽被返回。在一些具体实现中，域宽必须在每个输出之前被设置，否则将采用默认域宽。streamsize被编译器定义为某种整型形式。

设置了最小域宽之后，当某个值小于指定的宽度时，则域将用当前的填充字符（默认情况下为空格）填充以达到指定的域宽。如果某个值超出了最小域宽，域将会超出限度，此时不会截去该值。

当输出浮点值时，可以使用precision( )函数确定数字的精度位数。该函数的原型如下所示：

```
streamsize precision(streamsize p);
```

这里，精度设置为p，先前的值将被返回。精度的默认值为6。在一些具体实现中，精度必须在输出每个浮点值之前被设置，否则将采用默认精度。

默认情况下，当需要填充一个域时将使用空格填充。也可以利用fill( )函数指定填充字符。该函数的原型如下所示：

```
char fill(char ch);
```

在调用fill( )后，ch将变成新的填充字符，先前的填充字符被返回。下面的程序说明了这些函数的用法：

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

int main()
{
  cout.precision(4) ;
  cout.width(10);

  cout << 10.12345 << "\n"; // displays 10.12

  cout.fill('*');
```

```
    cout.width(10);
    cout << 10.12345 << "\n"; // displays *****10.12

    // field width applies to strings, too
    cout.width(10);
    cout << "Hi!" << "\n"; // displays *******Hi!
    cout.width(10);
    cout.setf(ios::left); // left justify
    cout << 10.12345; // displays 10.12*****

    return 0;
}
```

程序的输出如下所示：

```
     10.12
*****10.12
*******Hi!
10.12*****
```

width( )，precision( )和fill( )有各自的重载形式，它们可以获得当前设置，但不会改变当前设置。这些重载形式如下所示：

```
char fill( );
streamsize width( );
streamsize precision( );
```

20.4.8　利用操作算子格式化 I/O

改变流的格式参数的另一种方法是使用称为操作算子（manipulator）的一些特殊函数，这些函数可以包含在I/O表达式中。标准操作算子如表20.1所示。从表20.1中可以看出，许多I/O操作算子与ios类的成员函数相对应。最近，许多操作算子被加进了C++并且将不被老式编译器所支持。

表 20.1　C++ 操作算子

操作算子	用途	输入 / 输出
boolalpha	开启 boolapha 标记	输入 / 输出
dec	开启 dec 标记	输入 / 输出
endl	输出一个换行符并刷新流	输出
ends	输出一个 null	输出
fixed	开启 fixed 标记	输出
flush	刷新一个流	输出
hex	开启 hex 标记	输入 / 输出
internal	开启 internal 标记	输出
left	开启 left 标记	输出
nobooalpha	关闭 boolalpha 标记	输入 / 输出
noshowbase	关闭 showbase 标记	输出
noshowpoint	关闭 showpoint 标记	输出
noshowpos	关闭 showpos 标记	输出
noskipws	关闭 skipws 标记	输入
nounitbuf	关闭 unitbuf 标记	输出

（续表）

操作算子	用途	输入/输出
nouppercase	关闭 uppercase 标记	输出
oct	开启 oct 标记	输入/输出
resetiosflags (fmtflags *f*)	关闭 f 中指定的标记	输入/输出
right	开启 right 标记	输出
scientific	开启 scientific 标记	输出
setbase(int *base*)	将基数设置为 base	输入/输出
setfill(int *ch*)	将填充字符设置为 ch	输出
setiosflags(fmtflags *f*)	开启 f 中指定的标记	输入/输出
setprecision (int *p*)	设置数字精度	输出
setw(int *w*)	将域宽设置为 w	输出
showbase	开启 showbase 标记	输出
showpoint	开启 showpoint 标记	输出
showpos	开启 showpos 标记	输出
skipws	开启 skipws 标记	输入
unitbuf	开启 unitbuf 标记	输出
uppercase	开启 uppercase 标记	输出
ws	跳过开始的空格	输入

为了访问带参数的操作算子，例如 setw( )，必须在程序中包括<iomanip>。

下面是一个使用操作算子的例子：

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

int main()
{
  cout << hex << 100 << endl;

  cout << setfill('?') << setw(10) << 2343.0;

  return 0;
}
```

上面程序将显示如下信息：

```
64
??????2343
```

注意操作算子是如何出现在一个较大的I/O表达式中的，同时也要注意，当操作算子不带参数时（例如此例中的endl( )），后面将不跟括号，这是因为它是一个传递给重载运算符<< 的函数地址。

作为对照，下面的程序与前面的程序具有相同的功能，该程序利用ios成员函数获得了同样的结果：

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

```
int main()
{
  cout.setf(ios::hex, ios::basefield);
  cout << 100 << "\n"; // 100 in hex

  cout.fill('?');
  cout.width(10);
  cout << 2343.0;

  return 0;
}
```

从这个例子可以看出，使用操作算子而不是ios成员函数的主要优点是：它们常常使编写的代码更加紧凑。可以使用setiosflags( )操作算子直接设置各种与流相关的格式标记，例如，下面的程序使用setiosflags( )设置了showbase和showpos标记：

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

int main()
{
  cout << setiosflags(ios::showpos);
  cout << setiosflags(ios::showbase);
  cout << 123 << " " << hex << 123;

  return 0;
}
```

操作算子setiosflags( )执行的功能与成员函数setf( )的相同。

更有趣的操作算子之一是boolapha，它可以使用字词“true”和“false”（而不是使用数字）输入和输出真值和假值。例如：

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

int main()
{
  bool b;

  b = true;
  cout << b << " " << boolalpha << b << endl;

  cout << "Enter a Boolean value: ";
  cin >> boolalpha >> b;
  cout << "Here is what you entered: " << b;

  return 0;
}
```

该范例的运行结果如下所示：

```
1 true
Enter a Boolean value: false
Here is what you entered: false
```

20.5 重载<<和>>

我们知道,<<和>>运算符在C++中被重载，从而可以对C++的内置类型执行I/O操作。还可以对这些运算符进行重载，以使其对你所创建的类型执行I/O操作。

在C++中，因为<<可以把字符插入到一个流中，所以该输出运算符被称为插入运算符。同样,>>输入运算符被称为提取运算符，因为该运算符可以从一个流中提取字符。重载插入和提取运算符的函数通常称为插入器（inserter）和析取器（extractor）。

20.5.1 创建自己的插入器

为自己的类创建插入器非常简单。所有插入器函数的一般形式是：

```
ostream &operator<<(ostream &stream, class_type obj)
{
  // body of inserter
  return stream;
}
```

注意，该函数返回一个到ostream类型的流的引用（记住，ostream是支持输出的ios类的派生类）。此外，该函数的第一个参数是一个到输出流的引用，第二个参数是被插入的对象（第二个参数也可以是一个到被插入对象的引用）。插入器在退出之前必须要做的最后一件事是返回流，这使得可以在一个较大的I/O表达式中使用插入器。

在插入器函数中，可以放置你所需要的任何类型的过程或操作。也就是说，插入器的作用完全取决于你。然而，为了在编写程序时保持良好的编程习惯，应该将插入器的操作限定为向流输出信息。例如，让一个插入器计算出小数点后具有30位数字的p并使其作为插入操作的一种副作用或许并不可取！

为了说明如何定制插入器，我们将为phonebook类型的对象创建一个插入器，具体做法如下所示：

```
class phonebook {
public:
  char name[ 80] ;
  int areacode;
  int prefix;
  int num;
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
  }
};
```

这个类用来保存人名和电话号码。下面是一种为phonebook类型的对象创建插入器函数的一种方法：

```
// Display name and phone number
```

```
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // must return stream
}
```

下面是一个简单的说明 phonebook 插入器函数的程序：

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

class phonebook {
public:
  char name[80];
  int areacode;
  int prefix;
  int num;
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
  }
};

// Display name and phone number.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // must return stream
}

int main()
{
  phonebook a("Ted", 111, 555, 1234);
  phonebook b("Alice", 312, 555, 5768);
  phonebook c("Tom", 212, 555, 9991);

  cout << a << b << c;

  return 0;
}
```

这个程序的输出如下所示：

```
Ted (111) 555-1234
Alice (312) 555-5768
Tom (212) 555-9991
```

从前面的程序中可以看出，phonebook插入器不是phonebook的成员。尽管初看上去有些不可思议，但是这很容易理解。当一个任意类型的运算符函数是一个类成员时，左边的操作数是调用该运算符函数的对象。而且，这个对象是运算符函数作为其类成员的类的对象，这一点无法改变。如果一个重载运算符函数是一个类的对象，那么其左侧的操作数一定是该类的对象。然而，当重载插入器时，其左侧的操作数是一个流，右侧的操作数是一个类的对象。因此，重载的插入器不是重载它们的类的成员。前面程序中的变量name，areacode，prefix和num是公有变量，所以可以被插入器访问。

在C++中，插入器不是定义它们的类的成员，这似乎是一个严重的缺陷。既然重载的插入器不是成员，那么它们如何访问一个类的私有成分呢？在前面的程序中，所有成员都是公有的，然而，封装才是面向对象编程的一个基本组件。如果要求所有的输出数据都是公有数据，则将与这种原则相冲突。好在有一种方案可以化解这种两难的局面：将插入器作为类的友元（friend），此举既可以满足将重载插入器的第一个参数作为一个流的要求，又可以保证该函数可以访问重载它的类的私有成员。下面的程序与前面程序具有相同的功能，但却将插入器改成了一个friend函数：

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

class phonebook {
  // now private
  char name[80];
  int areacode;
  int prefix;
  int num;
public:
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
  }

friend ostream &operator<<(ostream &stream, phonebook o);
};

// Display name and phone number.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";

  stream << o.prefix << "-" << o.num << "\n";

  return stream; // must return stream
}

int main()
{
```

```
  phonebook a("Ted", 111, 555, 1234);
  phonebook b("Alice", 312, 555, 5768);
  phonebook c("Tom", 212, 555, 9991);

  cout << a << b << c;

  return 0;
}
```

当定义插入器函数体时，要记住应使其尽可能地保持普遍性。例如，前面例子中所示的插入器可以适用于任何流，因为这个函数体将输出定向到调用该插入器的流中。虽然从技术上讲将语句

```
stream << o.name << " ";
```

写为

```
cout << o.name << " ";
```

没有错误，但是该语句将有把硬编码的cout作为输出流的效果。最初的版本可以作用于任何流，包括那些连接到磁盘文件的流。虽然在某些情况下，特别是在涉及专用输出设备的情况下，你或许想要对输出流进行硬编码，但在大多数情况下不需要这样做。一般来说，插入器越灵活，它们就越有价值。

注意：除非遇到像0034这样的num值（在这种情况下，前面的0不被显示），否则phonebook类的插入器都能正常工作。要证明这一点，可以把num改成字符串，也可以把填充字符设置为0并利用width()格式函数生成前导0。这个问题作为练习留给读者解答。

在讲解析取器之前，让我们再看一个插入器函数的例子。插入器不必被限定为只能处理文本，它可以用来以任何形式输出有意义的数据。例如，对于CAD系统的部分类来说，它们所用的一种插入器可以输出绘图仪指令，另一种插入器可以生成图形图像。基于Windows程序的插入器可以显示对话框。为了举例说明非文本输出，我们看一看下面的程序，该程序可以在屏幕上画出方框（因为C++没有定义图形库，所以该程序利用字符绘制方框，但如果读者使用的系统支持图形，希望你们用图形代替字符）。

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

class box {
  int x, y;
public:
  box(int i, int j) { x=i; y=j; }
  friend ostream &operator<<(ostream &stream, box o);
};

// Output a box.
ostream &operator<<(ostream &stream, box o)
{
  register int i, j;

  for(i=0; i<o.x; i++)
    stream << "*";

  stream << "\n";
```

```
  for(j=1; j<o.y-1; j++) {
    for(i=0; i<o.x; i++)
      if(i==0 || i==o.x-1) stream << "*";
      else stream << " ";
    stream << "\n";
  }

  for(i=0; i<o.x; i++)
    stream << "*";
  stream << "\n";

  return stream;
}

int main()
{
  box a(14, 6), b(30, 7), c(40, 5);
  cout << "Here are some boxes:\n";
  cout << a << b << c;

  return 0;
}
```

程序显示的内容如下：

```
Here are some boxes:
**************
*            *
*            *
*            *
*            *
**************
********************************
*                              *
*                              *
*                              *
*                              *
*                              *
********************************
****************************************
*                                      *
*                                      *
*                                      *
****************************************
```

20.5.2 创建自己的析取器

析取器与插入器相反。析取器函数的一般形式如下：

```
istream &operator>>(istream &stream, class_type &obj)
{
  // body of extractor
  return stream;
}
```

析取器返回一个istream类型的流的引用，该流是一个输入流。第一个参数也必须是一个到istream类型的流的引用。注意，第二个参数必须是一个析取器为之重载的类的对象的引用，只有这样，该对象才可以被输入（析取）操作所修改。

我们仍以 phonebook 类为例，下面的程序说明了一种编写析取器的方法：

```
istream &operator>>(istream &stream, phonebook &o)
{
  cout << "Enter name: ";
  stream >> o.name;
  cout << "Enter area code: ";
  stream >> o.areacode;
  cout << "Enter prefix: ";
  stream >> o.prefix;
  cout << "Enter number: ";
  stream >> o.num;
  cout << "\n";

  return stream;
}
```

我们注意到，尽管这是一个输入函数，但它通过提示用户执行输入。关键是，尽管析取器的主要目的是输入，但也可以执行为达此目的所需的任何操作。然而，就插入器而言，最好是使由析取器执行的操作直接与输入相关。如果不这样做，将会有破坏结构和清晰度之险。

下面的程序可以说明 phonebook 析取器：

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

class phonebook {
  char name[80];
  int areacode;
  int prefix;
  int num;
public:
  phonebook() { };
  phonebook(char *n, int a, int p, int nm)
  {
    strcpy(name, n);
    areacode = a;
    prefix = p;
    num = nm;
  }
  friend ostream &operator<<(ostream &stream, phonebook o);
  friend istream &operator>>(istream &stream, phonebook &o);
};

// Display name and phone number.
ostream &operator<<(ostream &stream, phonebook o)
{
  stream << o.name << " ";
  stream << "(" << o.areacode << ") ";
```

```
  stream << o.prefix << "-" << o.num << "\n";

  return stream; // must return stream
}

// Input name and telephone number.
istream &operator>>(istream &stream, phonebook &o)
{
  cout << "Enter name: ";
  stream >> o.name;
  cout << "Enter area code: ";
  stream >> o.areacode;
  cout << "Enter prefix: ";
  stream >> o.prefix;
  cout << "Enter number: ";
  stream >> o.num;
  cout << "\n";

  return stream;
}

int main()
{
  phonebook a;

  cin >> a;

  cout << a;

  return 0;
}
```

实际上，因为只有当输入流被连接到像控制台这样的交互设备（即当输入流时 cin 时）时才需要 cout 语句，所以 phonebook 的析取器并不完美。例如，如果将析取器用于与磁盘文件相连的流，cout 语句将不再适用。为了好玩起见，你或许想试着取消 cout 语句（当输入流引用 cin 时除外）。例如，你或许会采用如下所示的 if 语句：

```
if(stream == cin) cout << "Enter name: ";
```

现在，只有当输出设备是屏幕时，才给出提示。

20.5.3 创建自己的操作算子函数

除重载插入和析取运算符之外，还可以通过创建自己的操作算子函数定制C++的I/O系统。定制操作算子之所以重要，主要出于以下两个原因：第一，可以把一系列独立的I/O操作合并进一个操作算子中。例如，有一种常见的情形是在一个程序中频繁出现同样的I/O操作序列，在这种情况下就可以利用定制操作算子执行这些操作，从而达到简化源代码和防止出现意外错误的目的。定制操作算子非常重要的第二个原因体现在需要对非标准设备执行I/O操作时。例如，可以使用操作算子给特殊类型的打印机或光识别系统发送控制代码。

定制操作算子虽然是支持OOP的C++的一个特征，但它也有助于非面向对象的编程。我们将会看到，定制操作算子可以使所有I/O密集型程序都更清晰和更有效。

我们知道，操作算子有两种基本类型：一种用来操作输入流，另一种用来操作输出流。除

了这两种广义的划分之外，还有另一种分类方法：即将其分为带一个参数的操作算子和不带参数的操作算子。坦白地说，创建带参数的操作算子的过程随编译器的不同而差别很大，这种差别甚至存在于同一个编译器的两个不同版本之间。由于这个原因，用户一定要查阅相关编译器的关于创建带参数的操作算子的指令文档。然而，要创建无参数的操作算子，则非常简单，而且对所有编译器而言全都相同，现说明如下。

所有无参数的操作算子输出函数具有以下架构：

```
ostream &manip-name(ostream &stream)
{
  // your code here
  return stream;
}
```

这里，manip-name 是操作算子的名字。我们注意到，该函数返回一个对 ostream 类型的流的引用。如果将一个操作算子用做较大的 I/O 表达式的一部分，这样做是必须的。即使操作算子有一个参数为到它正在操作的流的引用，在输出操作中插入操作算子时也不使用参数，这一点非常重要。

作为一个简单的例子，下面的程序创建一个称为 sethex( )的操作算子，该函数将开启 showbase 标记并将输出设置为十六进制形式：

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

// A simple output manipulator.
ostream &sethex(ostream &stream)
{
  stream.setf(ios::showbase);
  stream.setf(ios::hex, ios::basefield);

  return stream;
}

int main()
{
  cout << 256 << " " << sethex << 256;

  return 0;
}
```

这个程序显示 256 0x100。可以看出，sethex 像所有内置操作算子一样，以同样的方式用做 I/O 表达式的一部分。

定制操作算子的使用并不复杂。例如，以下是两个简单的操作算子 la( )和 ra( )，它们分别显示左箭头和右箭头：

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

// Right Arrow
ostream &ra(ostream &stream)
```

```
{
  stream << "-------> ";
  return stream;
}

// Left Arrow
ostream &la(ostream &stream)
{
  stream << " <-------";
  return stream;
}

int main()
{
  cout << "High balance " << ra << 1233.23 << "\n";
  cout << "Over draft " << ra << 567.66 << la;

  return 0;
}
```

上面的程序将显示以下内容：

```
High balance -------> 1233.23
Over draft -------> 567.66 <-------
```

如果使用频繁，这些简单的操作算子可以把你从单调乏味的录入工作中解放出来。

对往某个设备发送代码来说，使用输出操作算子非常实用。例如，一个打印机可能接受各种代码，这些代码可以被改变以便具有不同的字号或字体，或者设置打印头的位置。如果这些调整比较频繁，则应该利用操作算子。

所有无参数的输入操作算子函数具有以下架构：

```
istream &manip-name(istream &stream)
{
  // your code here
  return stream;
}
```

输入操作算子接收一个对调用它的流的引用，这个流必须被该操作算子返回。

下面的程序创建 getpass( )输入操作算子，它可以鸣铃并提示用户输入密码：

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

// A simple input manipulator.
istream &getpass(istream &stream)
{
  cout << '\a'; // sound bell
  cout << "Enter password: ";

  return stream;
}

int main()
{
```

```
  char pw[80];

  do {
    cin >> getpass >> pw;
  } while (strcmp(pw, "password"));

  cout << "Logon complete\n";

  return 0;
}
```

记住，至关重要的一点是：你的操作算子将返回流，否则将不能在一系列输入或输出操作中使用操作算子。