第 20 章

iostream 库

C++的输入/输出设施是由 *iostream 库（iostream library）*提供的，它是一个利用多继承和虚拟继承实现的面向对象类层次结构，是作为 C++标准库的一个组件而提供的。它为内置数据类型的输入输出提供了支持，也支持文件的输入输出。除此之外，类的设计者可以扩展 iostream 库，以读写新的类类型。

为了在我们的程序中使用 iostream 库，我们必须包含相关的头文件，如下：

```
#include <iostream>
```

输入输出操作是由 istream（输入流）和 ostream（输出流）类提供的（第三个类 iostream 类同时从 istream 和 ostream 派生，允许双向输入/输出）。为了方便，这个库定义了下列三个标准流对象。

1．cin，发音为 see-in，代表*标准输入（standard input）*的 istream 类对象。一般地，cin 使我们能够从用户终端读入数据。

2．cout，发音为 see-out，代表*标准输出（standard output）*的 ostream 类对象。一般地，cout 使我们能够向用户终端写数据。

3．cerr，发音为 see-err，代表*标准错误（standard error）*的 ostream 类对象。cerr 是导出程序错误消息的地方。

输出主要由重载的左移操作符（<<）来完成。类似地，输入主要由重载的右移操作符（>>）来完成。例如：

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

int main()
{
```

```
    string in_string;

    // 向用户终端写字符串
    cout << "Please enter your name: ";

    // 把用户输入读取到 in_string中
    cin >> in_string;

    if ( in_string.empty() )
       // 产生一个错误消息，输出到用户终端
         cerr << "error: input string is empty!\n";
    else cout << "hello, " << in_string << "!\n";
}
```

在考虑这两个操作符时，一种很有用的思考方式是，它们指出了数据移动的方向。例如，

```
>> x
```

把数据放入 x 中，而

```
<< x
```

从 x 中拿出数据。20.1 节将介绍 iostream 库为数据输入提供的支持。20.5 节将了解怎样扩展 iostream 库，以支持新类类型的数据输入。类似地，20.2 节将介绍 iostream 库为数据输出提供的支持，20.4 节我们将了解怎样扩展这个库，以允许新类类型的数据输出。

除了对用户终端的读写操作之外，iostream 库还支持对文件的读写。下列三种类类型提供了文件支持：

1．ifstream，从 istream 派生，把一个文件绑到程序上用来输入。

2．ofstream，从 ostream 派生，把一个文件绑到程序上用来输出。

3．fstream，从 iostream 派生，把一个文件绑到程序上用来输入和输出。

为了使用 iostream 库的文件流组件，我们必须包含相关的头文件：

```
#include <fstream>
```

（fstream 头文件包含了 iostream 头文件，所以我们不需要同时包含这两个文件。）C++为文件的输入/输出也支持同样的输入和输出操作符。例如：

```
#include <fstream>
#include <string>
#include <vector>
#include <algorithm>

int main()
{
    string ifile;

    cout << "Please enter file to sort: ";
    cin >> ifile;

    // 构造一个 ifstream 输入文件对象
```

```
    ifstream infile( ifile.c_str() );

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

    string ofile = ifile + ".sort";

    // 构造一个 ofstream 输出文件对象
    ofstream outfile( ofile.c_str() );
    if ( ! outfile) {
         cerr << "error: unable to open output file: "
                << ofile << endl;
         return -2;
    }

    string buffer;
    vector< string, allocator > text;

    int cnt = 1;
    while ( infile >> buffer ) {
             text.push_back( buffer );
             cout << buffer << ( cnt++ % 8 ? " " : "\n" );
    }

    sort( text.begin(), text.end() );

    // ok: 把排序后的词打印到 outfile
    vector<string,allocator>::iterator iter = text.begin();
    for ( cnt = 1; iter != text.end(); ++iter, ++cnt )
          outfile << *iter
                  << (cnt%8 ? " " : "\n" );

    return 0;
}
```

下面是运行该程序的一个例子：要求我们输入一个文件以便排序。我们键入 alice_emma **（我们的输入在示例输出中以黑体显示。）程序把读入的每个单词回显到标准输出上：**

```
Please enter file to sort: alice_emma
Alice Emma has long flowing red hair. Her
Daddy says when the wind blows through her
hair, it looks almost alive, like a fiery
bird in flight. A beautiful fiery bird, he
tells her, magical but untamed. "Daddy, shush, there
is no such thing," she tells him, at
the same time wanting him to tell her
```

```
more. Shyly, she asks, "I mean, Daddy, is
there?"
```

然后程序把排序后的字符串序列写到 outfile 中。当然，标点符号也会影响单词的顺序（我们将在下节修正它）：

```
"Daddy, "I A Alice Daddy Daddy, Emma Her
Shyly, a alive, almost asks, at beautiful bird
bird, blows but fiery fiery flight. flowing hair,
hair. has he her her her, him him,
in is is it like long looks magical
mean, more. no red same says she she
shush, such tell tells tells the the there
there?" thing," through time to untamed. wanting when
wind
```

20.6 节我们将详细介绍文件输入/输出。

iostream 库还支持*内存输入/输出（in-memory input/output）*，一个流被附着在程序的内存中的一个字符串上。然后，我们可以用 iostream 输入和输出操作符来对它进行读写。我们可以通过定义下列三种类类型的一个实例来定义一个 iostream 字符串对象：

1．istringstream，从 istream 派生，从一个字符串中读取数据；

2．ostringstream，从 ostream 派生，写入到一个字符串中；

3．stringstream，从 iostream 派生，从字符串中读取，或者写入到字符串中。

要使用这些类，我们必须包含相关的头文件：

```
#include <sstream>
```

（sstream 头文件包含了 iostream 头文件，因此我们无需同时包含这两个头文件。）在下面的代码段中，一个 ostringstream 被用来格式化一个错误信息。然后返回底层的字符串：

```
#include <sstream>

string program_name( "our_program" );
string version( "0.01" );

// ...

string mumble( int *array, int size )
{
   if ( ! array ) {
        ostringstream out_message;

      out_message << "error: "
                  << program_name << "--" << version
                   << ": " << __FILE__ << ": " << __LINE__
                  << " -- ptr is set to 0; "
                   << " must address some array.\n";
```

```
        // 返回底层 string 对象
        return out_message.str();
    }
    // ...
}
```

20.8 节将详细介绍 iostream 字符串对象。

在实践中，iostream 支持两种预定义的字符类型：char 和 wchar_t。目前我们所描述的 iostream 类（以及我们在本章余下部分要关注的）读写的是 char 型的流。与此互补的是另外一组支持 wchar_t 型的 iostream 对象和类。每个类与类对象都加了前缀"w"，以便与相应的 char 型区分开。因此，wchar_t 标准输入被命名为 wcin、标准输出为 wcout，以及标准错误 wcerr。然而，char 和 wchar_t 型的 stream 类和类对象所需要的头文件是相同的。

wchar_t 输入和输出类是 wistream、wostream、和 wiostream。文件输入和输出类是 wifstream、wofstream、和 wfstream。iostream 字符串输入输出类是 wistringstream、wostringstream、和 wstringstream。

20.1　输出操作符<<

最常用的输出方法是在 cout 上应用左移操作符(<<)。例如：

```
#include <iostream>

int main() {
    cout << "gossipaceous Anna Livia\n";
}
```

在用户终端上输出以下内容：

```
gossipaceous Anna Livia
```

输出操作符可以接受任何内置数据类型的实参，包括 const char*，以及标准库 string 和 complex 类类型。任何表达式，包括函数调用，都可以是输出操作符的实参，只要它的计算结果是一个能被输出操作符实例接受的数据类型即可。例如：

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

int main()
{
    cout << "The length of "ulysses" is:\t";
    cout << strlen( "ulysses" );
    cout << '\n';

    cout << "The size of "ulysses" is:\t";
```

```
    cout << sizeof( "ulysses" );
    cout << endl;
}
```

在用户终端上输出如下内容：

```
The length of "ulysses" is: 7
The size of "ulysses" is:8
```

(endl **是一个** ostream ***操纵符（****manipulator****）***，**它把一个换行符插入到输出流中，然后刷新** ostream **缓冲区。我们将在** 20.9 **节介绍有关缓冲的做法**)。

把输出操作符连接成一条语句常常会更方便一些。例如，上面的程序可以重写为：

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

int main()
{
    // 输出操作符可以被连接在一起

    cout << "The length of "ulysses" is:\t"
         << strlen( "ulysses" ) << 'n';

    cout << "The size of "ulysses" is:\t"
         << sizeof( "ulysses" ) << endl;
}
```

输出操作符序列（以及输入操作符序列）能够被连接的原因是，表达式

```
cout << "some string"
```

计算的结果是左边的 ostream **操作数；也就是说，表达式的结果是** cout **对象自己，于是，通过这个序列，它又被应用到下一个输出操作符上，等等（我们说操作符**<<**从左向右结合）。**

iostream **库还提供了指针类型的预定义输出操作符，允许显示对象的地址。缺省情况下，这些值以十六进制的形式显示。例如，**

```
#include <iostream>

int main()
{
    int i = 1024;
    int *pi = &i;

    cout << "i:   " << i
         << "\t&i:\t" << &i << '\n';

    cout << "*pi: " << *pi
         << "\tpi:\t"   << pi  << endl
         << "\t\t&pi:\t" << &pi << endl;
```

```
}
```

在终端上输出：

```
i:   1024  &i: 0x7ffff0b4
*pi: 1024   pi: 0x7ffff0b4
         &pi: 0x7ffff0b0
```

后面我们将会看到怎样用十进制数形式打印地址。
下面的程序展示了一种迷惑。我们的目的是输出 pstr **包含的地址值：**

```
#include <iostream>

const char *str = "vermeer";
int main()
{
    const char *pstr = str;
    cout << "The address of pstr is: "
         << pstr << endl;
}
```

但是，编译并运行程序，却产生了以下意料之外的输出：

```
The address of pstr is: vermeer
```

问题是，类型 const char* **没有被解释成地址值，而是** C **风格字符串。为了输出** pstr **包含的地址值，我们必须改变** const char* **的缺省处理。我们将分两步完成，首先把** const **强制转换掉，然后把** pstr **强制转换成** void* **类型：**

```
<< static_cast<void*>(const_cast<char*>(pstr))
```

编译并运行程序，现在产生期望的输出：

```
The address of pstr is: 0x116e8
```

下面是另一个迷惑。我们的目的是显示两个值中的较大值：

```
#include <iostream>

inline void
max_out( int val1, int val2 ) {
    cout << ( val1 > val2 ) ? val1 : val2;
}

int main()
{
    int ix = 10, jx = 20;

    cout << "The larger of " << ix;
    cout << ", " << jx << " is ";
```

```
    max_out( ix, jx );

    cout << endl;
}
```

但是，编译并运行程序，却生成如下不正确的结果：

```
The larger of 10, 20 is 0
```

问题在于输出操作符的优先级高于条件操作符，所以输出 val1 **和** val2 **比较结果的** true/false **值。即，表达式**

```
cout << ( val1 > val2 ) ? val1 : val2;
```

被计算为

```
(cout << ( val1 > val2 )) ? val1 : val2;
```

因为 val1 **不大于** val2**，所以计算结果为** false**，它被输出为** 0**。为了改变预定义的操作符优先顺序，整个条件操作符表达式必须被放在括号中：**

```
cout << (val1 > val2 ? val1 : val2);
```

这次产生了正确的输出：

```
The larger of 10, 20 is 20
```

如果 bool **文字值** true **和** false **以字符串的形式输出，而不是** 0 **或** 1 **—— 即，如果输出为：**

```
The larger of 10, 20 is false
```

则前面不正确的输出可能会更容易调试，从而不至于让程序员产生迷惑。

缺省情况下，false**文字值被输出为** 0**，**true **为** 1**。我们可以通过应用**boolalpha **操纵符来改变这种缺省行为。下面的程序正是这样做的：**

```
int main()
{
    cout << "default bool values: "
         << true << " " << false
         << "\nalpha bool values: "
         << boolalpha
         << true << " " << false
         << endl;
}
```

程序执行时产生下面的输出：

```
default bool values: 1 0
alpha bool values: true false
```

对于内置数组及容器类型（如 vector **或** map**）的输出，要求迭代一遍，并输出每个单独的元素。例如：**

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

string pooh_pals[] = {
      "Tigger", "Piglet", "Eeyore", "Rabbit"
};

int main()
{
   vector<string> ppals( pooh_pals, pooh_pals+4 );

   vector<string>::iterator iter = ppals.begin();
   vector<string>::iterator iter_end = ppals.end();

   cout << "These are Pooh's pals: ";
   for ( ; iter != iter_end; iter++ )
        cout << *iter << " ";

   cout << endl;
}
```

我们可以不用显式地“对容器中的元素进行迭代，并依次输出每个元素”，ostream_iterator**可以被用来实现同样的效果。例如，下面是一个等价的程序，它使用了** ostream_iterator**（关于** ostream_iterator **的详细讨论见** 12.4 **节）：**

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <string>

string pooh_pals[] = {
      "Tigger", "Piglet", "Eeyore", "Rabbit"
};

int main()
{
    vector<string> ppals( pooh_pals, pooh_pals+4 );

    vector<string>::iterator iter     = ppals.begin();
    vector<string>::iterator iter_end = ppals.end();

    cout << "These are Pooh's pals: ";

    // 把每个元素拷贝到cout ...
    ostream_iterator< string > output( cout, " " );
    copy( iter, iter_end, output );

    cout << endl;
```

```
}
```

编译并运行程序，产生如下输出：

```
These are Pooh's pals: Tigger Piglet Eeyore Rabbit
```

练习 20.1
已知下列对象定义：

```
string sa[4] = { "pooh", "tigger", "piglet", "eeyore" };
vector< string > svec( sa, sa+4 );
string robin( "christopher robin" );
const char *pc = robin.c_str();
int ival = 1024;
char blank = ' ';
double dval = 3.14159;
complex purei( 0, 7 );
```

a）**在标准输出上打印出每个对象的值。**
b）**输出** pc **的地址值。**
c）**利用以下条件操作符的结果，输出** ival **和** dval **的最小值。**

```
ival < dval ? ival : dval
```

20.2　输　　入

输入主要由右移操作符（>>**）来支持。例如，在下面的程序中，从标准输入读入一个** int **型的值序列，并把它放在一个** vector **中：**

```
#include <iostream>
#include <vector>

int main()
{
    vector<int> ivec;
    int ival;

    while ( cin >> ival )
          ivec.push_back( ival );

    // ...
}
```

子表达式

```
cin >> ival
```

从标准输入读入一个整数值，如果成功，则把该值拷贝到 ival 中。这个子表达式的结果是左边的 istream 对象——在这种情况下，即 cin 自己。（我们马上就会看到，这使得输入操作符能够连接起来。）

表达式

```
while ( cin >> ival )
```

从标准输入读入一个序列，一直到 cin 为 false 为止。有两种情况使一个 istream 对象被计算为 false：读到文件末尾（在这种情况下，我们已经正确地读完文件中所有的值）或遇到一个无效的值，比如 3.14159（小数点是非法的）、1e-1（字符文字 e 是非法的），或者一般的任意字符串文字。在读入一个无效值的情况下，istream 对象被放置到一种错误状态中，并且对于值的所有读入动作都将停止。（在 20.7 节，我们将详细地讨论错误条件。）

一组预定义的输入操作符可以接受任何的内置数据类型，包括 C 风格字符串以及标准库 string 和 complex 类类型。例如：

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

int main()
{
    int item_number;
    string item_name;
    double item_price;

    cout << "Please enter the item_number, item_name, and price: "
         << endl;

    cin >> item_number;
    cin >> item_name;
    cin >> item_price;

    cout << "The values entered are: item# "
         << item_number << " "
         << item_name << " @$"
         << item_price << endl;
}
```

下面是程序的执行示例情况：

```
Please enter the item_number, item_name, and price:
10247 widget 19.99
The values entered are: item# 10247 widget @$19.99
```

如果在独立的行上输入每个项会怎样？这不是问题。缺省情况下，输入操作符丢弃任何中间空白（空格、制表符、换行符、走纸、回车，关于如何改变这种缺省行为的讨论见 20.9 节）：

```
Please enter the item_number, item_name, and price:
10247
widget
19.99
The values entered are: item# 10247 widget @$19.99
```

“数值的读操作”比“数值的写操作”更可能导致 iostream **错误。例如，如果输入项的序列是**

```
// 错误：item_name应该在第二个位置上
BuzzLightyear 10009 8.99
```

语句

```
cin >> item_number;
```

导致输入错误，因为 BuzzLightyear **显然不是一个** int **类型的值。当出现输入错误时，如果对** istream **对象进行测试，则测试结果为** false**。例如，一种更为健壮的做法如下：**

```
cin >> item_number;
if ( ! cin )
     cerr << "error: invalid item_number type entered!\n";
```

尽管 iostream **库支持输入操作符的连接，但是这种做法使我们无法测试个别读操作的可能错误，因此，我们只能在确实没有错误机会的情况下才能使用连接形式的输入操作符。下面是改写之后的程序，用到了连接形式的输入操作符：**

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

int main()
{
    int item_number;
    string item_name;
    double item_price;

    cout << "Please enter the item_number, item_name, and price: "
        << endl;

    // ok: but more error prone
    cin >> item_number >> item_name >> item_price;

    cout << "The values entered are: item# "
 << item_number << " "
 << item_name << " @$"
 << item_price << endl;
}
```

字符序列

```
ab c
d    e
```

由下列九个字符构成：‘a’、‘b’、‘ ’（空格）、‘c’、‘\n’（换行符）、‘d’、‘\t’（制表符）、‘e’和‘\n’。然而，下面的程序用输入操作符只读取了五个字母字符：

```
#include <iostream>

int main()
{
char ch;

// 读入每个字符，然后输出
while ( cin >> ch )
         cout << ch;
cout << endl;

// ...
}
```

程序执行时输出如下内容：

```
abcde
```

缺省情况下，所有的空白字符都被抛弃掉。如果我们希望读入空白字符，或许是为了保留原始的输入格式，或许是为了处理空白字符（比如计算换行符的个数），一种方法是使用 istream 的 get()成员函数（ostream 的 put()成员函数一般与 get()配合使用——稍后我们将更详细地看看这些函数）。例如：

```
#include <iostream>

int main()
{
char ch;

// 获取每个字符，包括空白字符
while ( cin.get( ch ))
       cout.put( ch );
// ...
}
```

（第二种方法是使用 noskipws 操纵符。）

对于下面的两个字符串序列，如果由 const char* 或 string 输入操作符来读入，则它们都被视为“包含五个由空白字符分隔的字符串”：

```
A fine and private place
"A fine and private place"
```

引号的存在并没有导致内嵌的空白字符被当作一个扩展字符串的一部分。相反，这两个引号成为第一个词的首字符，以及最后一个词的末字符。

我们可以不是显式地从标准输入上依次读入每个单独的元素，istream_iterator 可以被用来达到相同的行为效果。例如；

```
#include <algorithm>
#include <string>
#include <vector>
#include <iostream>

int main()
{
   istream_iterator< string > in( cin ), eos ;
   vector< string > text ;

   // 从标准输入向 text 拷贝值
   copy( in , eos , back_inserter( text ) ) ;

   sort( text.begin() , text.end() ) ;

   // 删除所有重复的值
   vector< string >::iterator it ;
   it = unique( text.begin() , text.end() ) ;
   text.erase( it , text.end() ) ;

   // 显示结果 vector
   int line_cnt = 1 ;
   for ( vector< string >::iterator iter = text.begin();
         iter != text.end() ; ++iter , ++line_cnt )
         cout << *iter
              << ( line_cnt % 9 ? " " : "\n" ) ;
   cout << endl;
}
```

程序的输入是程序代码文本本身。这些代码文本已经被存储在名为 istream_iter.C 的文件中。在 UNIX 下，我们可以把文件重定向到标准输入（istream_iter 是程序的名字）：

```
istream_iter < istream_iter.C
```

（对于非 UNIX 系统，请查询程序员指南手册。）程序执行时产生以下输出：

```
!= " "\n" #include % ( ) *iter ++iter
++line_cnt , 1 9 : ; << <algorithm> <iostream.h>
<string> <vector> = > >::difference_type >::iterator ? allocator back_inserter(
cin copy( cout diff_type eos for in in( int
istream_iterator< it iter line_cnt main() sort( string text text.begin()
text.end() text.erase( typedef unique( vector< { }
```

（关于 iostream iterator 在 12.4 节讨论。）

除了预定义的输入操作符以外，重载的输入操作符可以支持读入用户定义的类类型。20.5 节将详细介绍重载的输入操作符。

20.2.1　字符串输入

我们既可以以 C 风格字符数组的形式读入字符串，也可以以 string 类类型的形式读入字符串。建议使用 string 类类型。主要好处是，与字符串相关的内存可被自动管理。例如，为了把字符串当作 C 风格字符数组，我们必须判断数组的长度——该长度应该足够容纳每一个可能的字符串。典型情况下，我们将每个字符串读入到一个数组缓冲区中，然后从空闲存储区中分配“正好可以存放这个字符串”的内存，并把缓冲区拷贝到这个按需分配的内存区中。例如：

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

char inBuf[ 1024 ];
try
{
  while ( cin >> inBuf ) {
          char *str = new char[ strlen( inBuf )+1 ];
          strcpy( str, inBuf );
          // ... do something to str
          delete [] str;
   }
}
catch( ... ) { delete [] str; throw; }
```

string 类型非常易于管理：

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

string str;
while ( cin >> str )
      // ... do something to string
```

在本小节的余下部分，我们将了解用 C 风格字符数组和类 string 输入操作符来读入字符串。作为输入文本，我们将回到“young Alice Emma”：

```
Alice Emma has long flowing red hair. Her Daddy says
when the wind blows through her hair, it looks almost
alive, like a fiery bird in flight. A beautiful fiery
bird, he tells her, magical but untamed. "Daddy, shush,
there is no such creature," she tells him, at the same time
wanting him to tell her more. Shyly, she asks, "I mean,
Daddy, is there?"
```

我们将把它键入到名为 alice_emma 的文本文件中，然后将其重定向到程序的标准输入。以后介绍文件输入时，我们将直接打开并读取它。以下程序将从标准输入中以 C 风格字符数组形式读取字符串序列，并确定哪一个字符串最长。

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

int main()
{
    const int bufSize = 24;
    char buf[ bufSize ], largest[ bufSize ];

    // hold statistics;
    int curLen, max = -1, cnt = 0;
    while ( cin >> buf )
   {
        curLen = strlen( buf );
        ++cnt;

        // new longest word? save it.
        if ( curLen > max ) {
            max = curLen;
            strcpy( largest, buf );
        }
    }

    cout << "The number of words read is "
         << cnt << endl;

    cout << "The longest word has a length of "
         << max << endl;

    cout << "The longest word is "
         << largest << endl;
}
```

编译并执行程序，产生以下输出：

```
The number of words read is 65
The longest word has a length of 10
The longest word is creature,"
```

实际上，这个结果是不正确的。beautiful 是文本中最长的词，长度为 9。但是，被选中的是 creature，因为有一个逗句和一个引号附在它的后面。为了使程序能像用户期望的那样解释字符串，我们需要过滤掉非字母元素。

但是，在开始之前，我们再仔细地看一看这个程序。在程序中，每个字符串被存储在 buf

中，它被声明为长度为 24 的数组。如果读入的字符串长度等于或超出 24，buf 就会溢出。例如，上个例子可能会被修改如下：

```
while ( cin >> setw( bufSize ) >> buf )
```

这里 bufSize 是字符数组 buf 的长度。setw()把长度等于或大于 bufSize 的字符串分成最大长度为

```
bufSize – 1
```

的两个或多个字符串。

一个空字符被放在每个新串的末尾。为了使用 setw()，要求程序包含 iomanip 头文件：

```
#include <iomanip>
```

如果可见的 buf 声明没有指定维数，

```
char buf[] = "An unrealistic example";
```

程序员可以应用 sizeof 操作符——只要标识符是一个数组的名字，并且在表达式可见的域中：

```
while ( cin >> setw(sizeof( buf )) >> buf );
```

在下面的程序中，使用 sizeof 操作符导致没有预料到的程序行为：

```
#include <iostream>
#include <iomanip>

main()
{
    const int bufSize = 24;
    char buf[ bufSize ];

    char *pbuf = buf;

    // 每个大于sizeof(char*) 的字符串
    // 被分成两个或多个字符串
    while ( cin >> setw(sizeof(pbuf)) >> pbuf )
           cout << pbuf << endl;
}
```

编译并执行该程序，产生下列未预料的结果：

```
$ a.out
The winter of our discontent

The
win
ter
of
our
```

```
dis
con
ten
t
```

问题在于，传递给 setw()的是字符指针的大小而不是其指向的字符数组的大小。在这台特定的机器上，字符指针是四字节，所以原始输入被分成长度为 3 的字符串序列。

下面的代码试图修正这个错误，实际上导致更严重的错误：

```
while ( cin >> setw(sizeof(*pbuf)) >> pbuf )
```

意图是向 setw()传递 pbuf 指向的数组的大小。但是，符号

```
*pbuf
```

只产生一个 char。在这种情况下，被传递给 setw()的是 1。while 循环每次执行都会把一个空字符读入到 pbuf 指向的数组中。所以从来不会读取标准输入；该循环会无限进行下去。

使用 string 类类型，所有这些内存分配的问题都不存在了，string 会自动管理内存。下面是用 string 重写的程序：

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

int main()
{
string buf, largest;

// hold statistics;
int curLen, max = -1, cnt = 0;
while ( cin >> buf ) {
 curLen = buf.size();
 ++cnt;

 // new longest word? save it.
 if ( curLen > max ) {
 max = curLen;
 largest = buf;
 }
}

// ... rest the same
}
```

由于逗号和引号被解释成字符串的一部分，所以输出仍然不正确。让我们来写一个函数从字符串中滤掉这些元素：

```
#include <string>
void filter_string( string &str )
{
```

```
// 过滤元素
string filt_elems( "\",?.");
     string::size_type pos = 0;

     while (( pos = str.find_first_of( filt_elems, pos ))
               != string::npos )
               str.erase( pos, 1 );
}
```

这样做能工作得很好，但是，我们希望去掉的元素被固定在代码中了。较好的策略是，允许用户传递一个包含这些元素的字符串。如果用户希望使用这些缺省的元素，则他们可以传递一个空字符串。

```
#include <string>

void filter_string( string &str,
                  string filt_elems = string("\",?."))
{
   string::size_type pos = 0;

   while (( pos = str.find_first_of( filt_elems, pos ))
        != string::npos )
   str.erase( pos, 1 );
}
```

以下是 filter_string()的更一般化的版本，它接受一对 iterator，由它们标记出需要过滤的元素范围：

```
template <class InputIterator>
void filter_string( InputIterator first, InputIterator last,
                  string filt_elems = string("\",?."))
{
      for ( ; first != last; first++ )
      {
          string::size_type pos = 0;
          while (( pos = (*first).find_first_of( filt_elems, pos ))
             != string::npos )
                (*first).erase( pos, 1 );
      }
}
```

使用该函数的程序可能会这样：

```
#include <string>
#include <algorithm>
#include <iterator>
#include <vector>
#include <iostream>
```

```
bool length_less( string s1, string s2 )
      { return s1.size() < s2.size(); }

int main()
{
    istream_iterator< string > input( cin ), eos;

    vector< string > text;
    // copy 是一个泛型算法
    copy( input, eos, back_inserter( text ));

    string filt_elems( "\",.?;:");
    filter_string( text.begin(), text.end(), filt_elems );

    int cnt = text.size();
    // max_element是一个泛型算法
    string *max = max_element( text.begin(), text.end(),
                               length_less );
    int len = max->size();

    cout << "The number of words read is "
         << cnt << endl;

    cout << "The longest word has a length of "
         << len << endl;

    cout << "The longest word is "
         << *max << endl;
}
```

当我们在 max_element()**中使用缺省的** string **小于操作符时，程序的输出令我们吃惊：**

```
The number of words read is 65
The longest word has a length of 4
The longest word is wind
```

wind **显然不是最长的元素。对这个结果迷惑了一会儿之后，我们意识到，**string **的小于操作符计算的不是字符串的长度而是其字母顺序关系。在那种意义上，**wind **是文本中的最大字符串。为了找到最大长度的字符串，我们需要提供另外一个小于操作符：**length_less()**：**

```
The number of words read is 65
The longest word has a length of 9
The longest word is beautiful
```

练习 20.2

从标准输入读入类型为：string、double、string、int、string **的一个序列。检查是否有输入错误发生。**

练习 20.3

从标准输入读入未知数目的字符串，把它们存储在 list 中。并且判断最长和最短的字符串。

20.3　其他输入/输出操作符

在某些场合下，我们需要把输入流当作一个未经解释的字节序列来读取，而不是特定数据类型（如 char、int、srting 等等）的序列。istream 成员函数 get()一次读入一个字节。getline()一次读入一块字节，或者由一个换行符作为结束，或者由某个用户定义的终止字符作为结束。成员函数 get()有三种形式：

1．get(char& ch)**从输入流中提取一个字符，包括空白字符，并将它存储在 ch 中。它返回被应用的 istream 对象。例如，以下程序收集了在输入流上的各种统计信息，然后直接将其拷贝到输出流上。**

```
#include <iostream>

int main()
{
   char ch;
   int  tab_cnt = 0, nl_cnt = 0, space_cnt = 0,
        period_cnt = 0, comma_cnt = 0;

   while ( cin.get( ch )) {
         switch( ch ) {
               case ' ':  space_cnt++;  break;
                case '\t': tab_cnt++;    break;
                case '\n': nl_cnt++;     break;
                case '.':  period_cnt++; break;
                case ',':  comma_cnt++;  break;
          }
         cout.put( ch );
   }

   cout << "\nour statistics:\n\t"
        << "spaces: "    << space_cnt  << '\t'
        << "new lines: " << nl_cnt     << '\t'
        << "tabs: "      << tab_cnt    << "\n\t"
        << "periods: "   << period_cnt << '\t'
        << "commas: "    << comma_cnt  << endl;
}
```

ostream **成员函数** put()**提供了另外一种方法，用来将字符输出到输出流中。**put()**接受** char **型的实参并返回被调用的** ostream **类对象。**

编译并执行程序，产生下列输出：

```
Alice Emma has long flowing red hair. Her Daddy says
when the wind blows through her hair, it looks almost alive,
like a fiery bird in flight. A beautiful fiery bird, he tells her,
magical but untamed. "Daddy, shush, there is no such creature,"
she tells him, at the same time wanting him to tell her more.
Shyly, she asks, "I mean, Daddy, is there?"

our statistics:
     spaces: 59     new lines: 6    tabs: 0
     periods: 4     commas: 12
```

2．Get()的第二个版本也从输入流读入一个字符。区别是，它返回该字符值而不是被应用的 istream 对象。它的返回类型是 int 而不是 char，因为它也返回文件尾的标志(end-of-file)，该标志通常用-1 来表示以便与字符集区分开。为测试返回值是否为文件尾，我们将它与 iostream 头文件中定义的常量 EOF 作比较。被指定用来存放 get()返回值的变量，应该被声明为 int 类型，以便包含字符值和 EOF。下面是一个简单的例子：

```
#include <iostream>

int main()
{
   int ch;

  // alternatively:
  // while (( ch = cin.get()) && ch != EOF )
  while (( ch = cin.get()) != EOF )
         cout.put(ch);
  return 0;

}
```

使用前两个 get()中任何一个，读取下列字符序列需要七次迭代：

```
a b c
d
```

读入的七个字符是（‘a’、空格、‘b’、空格、‘c’、换行、‘d’）。第八次迭代遇到 EOF。对于输入操作符(>>)，因为它在缺省情况下跳过空白字符，所以它读取这个序列只需四次迭代，依次返回‘a’、‘b’、‘c’、‘d’。get()函数的下一种形式可以用两次迭代读取这个序列。

3．get()的第三个版本具有下列原型：

```
get(char *sink, streamsize size, char delimiter='\n')
```

sink 代表一个字符数组，用来存放被读取到的字符。size 代表可以从 istream 中读入的字符的最大数目。delimiter 表示，如果遇到它就结束读取字符的动作。delimiter 字符本身不会被读入，

而是留在 istream 中，作为 istream 的下一个字符。一个常见的错误是，在执行第二个 get()之前忘记了去掉 delimiter。在下面的程序例子中，我们用 istream 成员函数 ignore()来去掉 delimiter。缺省情况下，换行符被用作 delimiter。

字符读取过程一直进行，直到以下任何一个条件发生。在发生了任何一个条件之后，一个空字符被放在数组中的下一个位置上。

- size-1 个字符被读入；
- 遇到文件结束符（end-of-file）；
- 遇到 delimiter 字符(再次说明,它不会被放在数组中,而是留作 istream 的下一个字符)。

get()的返回值是被调用的 istream 对象。(gcount()返回实际被读入的字符个数。)下面是其用法的一个简单示例：

```
#include <iostream>

int main()
{
       const int max_line = 1024;
       char line[ max_line ];

      while ( cin.get( line, max_line ))
      {
            // maximum read is max_line - 1 to allow for null
            int get_count = cin.gcount();
            cout << "characters actually read: "
                 << get_count << endl;

            // do something with line

            // 如果遇到换行符
           // 在读下一行之前去掉它
           if ( get_count < max_line-1 )
               cin.ignore();
      }
}
```

针对 young Alice Emma 执行程序时，产生以下输出：

```
characters actually read: 52
characters actually read: 60
characters actually read: 66
characters actually read: 63
characters actually read: 61
characters actually read: 43
```

为了更好地测试它的行为，我们创建了一行，超过了 max_line 个字符，然后把它放在包含 Alice Emma 的文件的最前面：

```
characters actually read: 1023
characters actually read: 528
characters actually read: 52
characters actually read: 60
characters actually read: 66
characters actually read: 63
characters actually read: 61
characters actually read: 43
```

缺省情况下，ignore()从被调用的 istream 对象中读入一个字符并丢弃掉，但是我们也可以指定显式的长度和 delimiter。它的原型如下：

```
ignore( streamsize length = 1, int delim = traits::eof )
```

ignore ()从 istream 中读入并丢弃 length 个字符，或者遇到 delimiter 之前包含 delimiter 在内的所有字符，或者直到文件结尾。它返回当前被应用的 istream 对象。

因为程序员常常忘了在应用 get()之前丢弃 delimiter，所以使用成员函数 getline()要比 get()更好，因为它丢弃 delimiter 而不是将其留作 istream 的下一个字符。getline()的语法与 get()的三参数形式相同（它也返回被调用的 istream 对象）：

```
getline(char *sink, streamsize size, char delimiter='\n')
```

因为 getline()和 get()的三参数形式都可以读入 size 个或少于 size 个字符，所以我们有必要查询 istream 以确定实际读入了多少个字符。istream 成员函数 gcount()正好提供了这样的信息，它返回由最后的 get()或 getline()调用实际提取的字符数。

ostream 成员函数 write()提供了另外一种方法，可以输出字符数组。它不是输出“直到终止空字符为止的所有字符”，而是输出某个长度的字符序列，包括内含的空字符。它的函数原型如下：

```
write( const char *sink, streamsize length )
```

length 指定要显示的字符个数。write()返回当前被调用的 ostream 类对象。

与 ostream 的 write()函数相对应的函数是 istream 的 read()函数，它的原型被定义如下：

```
read( char* addr, streamsize size )
```

read()从输入流中提取 size 个连续的字节，并将其放在地址从 addr 开始的内存中。gcount()返回由最后一个 read()调用提取的字节数。read()返回当前被调用的 istream 类对象。下面是使用 getline()、gcount()和 write()的例子：

```
#include <iostream>

int main()
{
    const int lineSize = 1024;
    int lcnt = 0; // 读入多少行
    int max = -1; // 最长行的长度
    char inBuf[ lineSize ];
```

```
    // 读取 1024 个字符或者遇到换行符
    while (cin.getline( inBuf, lineSize ))
    {
        // 实际读入多少字符
        int readin = cin.gcount();

        // 统计：行数、最长行
        ++lcnt;
        if ( readin > max )
             max = readin;

        cout << "Line #" << lcnt
             << "\tChars read: " << readin << endl;

        cout.write( inBuf, readin).put('\n').put('\n');
    }

    cout << "Total lines read: " << lcnt << endl;
    cout << "Longest line read: " << max << endl;
}
```

当对 Moby Dick **的前几个句子执行这个程序时，程序产生下列输出：**

```
Line #1 Chars read: 45
Call me Ishmael.  Some years ago, never mind

Line #2 Chars read: 46
how long precisely, having little or no money

Line #3 Chars read: 48
in my purse, and nothing particular to interest

Line #4 Chars read: 51
me on shore, I thought I would sail about a little

Line #5 Chars read: 47
and see the watery part of the world.  It is a

Line #6 Chars read: 43
way I have of driving off the spleen, and

Line #7 Chars read: 28
regulating the circulation.

Total lines read: 7
Longest line read: 51
```

istream **的** getline()**函数只支持输入到一个字符数组中。但是，标准库给出了非成员的** getline() **实例，它可以输入到一个** string **对象中，原型如下：**

```
getline( istream &is, string str, char delimiter );
```

这个 getline()**实例的行为如下：读入最大数目为** str::max_size-1 **个字符。如果输入序列超出这个限制，则读操作失败，并且** istream **对象被设置为错误状态；否则，当读到** delimiter**（它被从** istream **中丢弃，但没有被插入到** string **中）或遇到文件结束符时，输入结束。**

另外还有三个 istream **操作符：**

```
// push character back into the iostream
putback( char c );

// resets pointer to 'next' istream item backward by one
unget();

// returns next character (or EOF)
// but does not extract it
peek();
```

下面代码段说明了怎样使用这些操作符：

```
char ch, next, lookahead;

while ( cin.get( ch ))
{
    switch (ch) {
    case '/':
        // 是注释行吗？用 peek() 看一看:
        // 是的？ ignore() 余下的行
        next = cin.peek();
        if ( next == '/' )
            cin.ignore( lineSize, '\n' );
        break;
    case '>':
        // 查找 >>=
        next = cin.peek();
        if ( next == '>' ) {
            lookahead = cin.get();
            next = cin.peek();
            if ( next != '=' )
                cin.putback( lookahead );
        }
    // ...
}
```

练习 20.4

请从标准输入读入以下字符序列，包括所有空白字符，并依次回显在标准输出上：

```
a b c
d    e
```

```
f
```

练习 20.5

请读入句子“ riverrun**，**from bend of bay to swerve of shore**”，将它当作**(a)**九个字符串的序列，**(b)**一个单个字符串。**

练习 20.6

请用 getline()**和** gcount()**从标准输入读入一系列文字行。确定读入的最长行（如果在读入一行时，要求应用多个** getline()**，则确保这一行仍被看作单独的一行）。**

20.4　重载输出操作符<<

实现一个类类型时，如果我们希望这个类支持输入和输出操作，那么我们必须提供重载的输入和输出操作符的实例。本节我们将了解怎样重载输出操作符。重载输入操作符是下一节的主题。下面是 WordCount **类的输出操作符重载实例：**

```
#include <iostream>

class WordCount {
    friend ostream&
          operator<<(ostream&, const WordCount&);
public:
    WordCount( string word, int cnt=1 );
    // ...
private:
    string word;
    int occurs;
};

ostream&
operator <<( ostream& os, const WordCount& wd )
{    // format: <occurs> word
     os << "< " << wd.occurs << " > "
     << wd.word;
     return os;
}
```

一个设计问题是，类的输出操作符是否应该产生尾部的换行符。由于内置数据类型的输出操作符并不产生这样的换行符，所以用户一般不会期望一个类的实例会提供换行符。因此，对于一个类的输出操作符而言，较好的设计选择是不产生尾部的换行符。

一旦定义了 WordCount **的输出操作符，我们现在就可以将它与其他输出操作符自由地混合使用。例如：**

```
#include <iostream>
```

```
#include "WordCount.h"

int main()
{
    WordCount wd( "sadness", 12 );
    cout << "wd:\n" << wd << endl;
    return 0;
}
```

在用户终端上输出如下：

```
wd:
< 12 > sadness
```

输出操作符是一个双目操作符，它返回一个 ostream **引用。重载定义的一般化框架如下：**

```
// general skeleton of the overloaded output operator
ostream&
operator <<( ostream& os, const ClassType &object )
{
    // any special logic to prepare object

    // actual output of members
    os <<  // ...

    // return ostream object
    return os;
}
```

它的第一个实参是一个 ostream **对象的引用。第二个一般是一个特定类类型的** const **引用。返回类型是一个** ostream **引用。它的值总是该输出操作符被应用的** ostream **对象。**

因为第一个实参是一个 ostream **引用，所以输出操作符必须被定义为非成员函数。（详细讨论见** 15.1 **节。）当输出操作符要求访问非公有成员时，它必须被声明为该类的友元。（关于友元的讨论见** 15.2 **节。）**

Location **是一个类，它包含一个单词每次出现所在的行列数。下面是它的定义：**

```
#include <iostream>

class Location {
    friend ostream& operator<<( ostream&, const Location& );
public:
    Location( int line=0, int col=0 )
           : _line( line ), _col( col ) {}
private:
    short _line;
    short _col;
};
```

```
ostream& operator <<( ostream& os, const Location& lc )
{
    // Location 对象的输出:  < 10,37 >
    os << "<" << lc._line
       << "," << lc._col << "> ";

    return os;
}
```

我们来重新定义 WordCount**，使它包含一个** Location **类对象的** vector**：**_occurList**，以及一个** string **类对象**_word**：**

```
#include <vector>
#include <string>
#include <iostream>
#include "Location.h"

class WordCount {
    friend ostream& operator<<(ostream&, const WordCount&);

public:
    WordCount(){}
    WordCount( const string &word ) : _word(word){}
    WordCount( const string &word, int ln, int col )
       : _word( word ){ insert_location( ln, col ); }

    string word()   const { return _word; }
    int    occurs() const { return _occurList.size(); }
    void   found( int ln, int col )
           { insert_location( ln, col ); }

private:
    void insert_location( int ln, int col )
        { _occurList.push_back( Location( ln, col )); }

    string           _word;
    vector< Location > _occurList;
};
```

string **类和** Location **类都定义了** operator<<()**的实例。下面是** WordCount **输出操作符的新定义：**

```
ostream&
operator <<( ostream& os, const WordCount& wd )
{
    os  << "<" << wd._occurList.size() << "> "
        << wd._word << endl;
```

```
    int cnt = 0, onLine = 6;
    vector< Location >::const_iterator first =
        wd._occurList.begin();

    vector< Location >::const_iterator last =
        wd._occurList.end();

    for ( ; first != last, ++first )
    {
        // os << Location
        os << *first << " ";

        // formatting: 6 to a line
        if ( ++cnt == onLine )
           { os << "\n"; cnt = 0;  }

    }
    return os;
}
```

下面的程序利用了 WordCount 的新定义。为简单起见，单词的出现位置被手工编在代码中。

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

int main()
{
    WordCount search( "rosebud" );

    // for simplicity, hand-code occurrences
    search.found(11,3);  search.found(11,8);
    search.found(14,2);  search.found(34,6);
    search.found(49,7);  search.found(67,5);
    search.found(81,2);  search.found(82,3);
    search.found(91,4);  search.found(97,8);

    cout << "Occurrences: " << "\n"
         << search << endl;

    return 0;
}
```

编译并执行该程序，它产生下列输出：

```
Occurrences:
<10> rosebud
<11,3>  <11,8>  <14,2>  <34,6>  <49,7>  <67,5>
<81,2>  <82,3>  <91,4>  <97,8>
```

该程序的输出被保存在名为 output 的文件中，我们接下来的努力目标是定义一个输入操作符将它读回来。

练习 20.7

已知下面的 Date 类定义，

```
class Date {
public:
  // ...
private:
  int month, day, year;
};
```

请给出输出操作符的重载实例，

（a）　产生以下格式

```
// spell the month out
September 8th, 1997
```

（b）　产生以下格式

```
9 / 8 / 97
```

（c）　哪一个更好？为什么？

（d）　Date 的输出操作符应该是一个友元函数吗？为什么？

练习 20.8

请为下面的 CheckoutRecord 类定义输出操作符：

```
class CheckoutRecord {
public:
  // ...
private:
  double book_id;
  string title;
  Date date_borrowed;
  Date date_due;
  pair<string,string> borrower;
  vector< pair<string,string>* > wait_list;
};
```

20.5　重载输入操作符>>

重载输入操作符(>>)与重载输出操作符类似，只不过出错的可能性更大。例如，下面是 WordCount 输入操作符的实现：

```
#include <iostream>
#include “WordCount.h”

/* 必须修改WordCount，指定输入操作符为友元
   class WordCount {
    friend ostream& operator<<( ostream&, const WordCount& );
    friend istream& operator>>( istream&, WordCount& );
*/

istream&
operator>>( istream &is, WordCount &wd )
{
    /* WordCount对象被读入的格式：
 ●   <2> string
 ●   <7,3> <12,36>
     */

    int ch;

    /* 读入小于符号，如果不存在
 ●   则设置istream 为失败状态并退出
     */
    if ((ch = is.get()) != ‘<’ )
    {
    is.setstate( ios_base::failbit );
    return is;
    }

    // 读入有多少个
    int occurs;
    is >> occurs;

    // 取 >; 不检查错误
    while ( is && (ch = is.get()) != ‘>’ ) ;

    is >> wd._word;

    // 读入位置;
    // 每个位置的格式: < line, col >
    for ( int ix = 0; ix < occurs; ++ix )
    {
        int line, col;

        // extract values
        while (is && (ch = is.get())!= ‘<’ ) ;
        is >> line;

        while (is && (ch = is.get())!= ‘,’ ) ;
```

```
        is >> col;

        while (is && (ch = is.get())!= '>' ) ;

        wd.occurList.push_back( Location( line, col ));
    }
    return is;
}
```

这个例子说明了一些可能的、与 iostream **错误状态相关的话题：**

1．**由于不正确的格式而导致失败，**istream **应该把状态标记为** fail**：**

```
is.setstate( ios_base::failbit )
```

2．**对于错误状态中的** iostream**，插入和提取操作没有影响。例如：**

```
while (( ch = is.get() ) != lbrace)
```

如果 istream **对象** is **处于一种错误状态，则该循环将永远进行下去。这就是为什么在每次调用** get()**之前必须测试条件的原因：**

```
// 测试 "is" 是否处于好的状态
while ( is && (ch = is.get()) != lbrace)
```

如果 istream **对象没有处于好的状态，则它被测试为** false**。我们将在** 20.7 **节更详细地了解** iostream **对象的条件状态。）**

下面的程序读取一个 WordCount **类对象，其内容正是通过上节定义的重载输出操作符写入的。**

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

int main()
{
    WordCount readIn;

    // operator>>( cin, readIn )
    cin >> readIn;

    if ( !cin ) {
        cerr << "WordCount input error" << endl;
        return -1;
    }

    // operator<<( cout, readIn )
    cout << readIn << endl;
}
```

编译并执行该程序，产生下列输出：

```
<10> rosebud
<11,3>  <11,8>  <14,2>  <34,6>  <49,7>  <67,5>
<81,2>  <82,3>  <91,4>  <97,8>
```

练习 20.9

WordCount 输入操作符直接处理单独的 Location 项的输入。请把这部分代码抽取到一个独立的 Location 输入操作符中。

练习 20.10

请为 20.4 节的练习 20.7 中定义的 Date 类提供一个输入操作符。

练习 20.11

请为 20.4 节的练习 20.8 中定义的 CheckoutRecord 类提供一个输入操作符。

20.6　文件输入和输出

如果一个用户希望把一个文件连接到程序上，以便用来输入或输出，则必须包含 fstream 头文件（它又包含了 iostream 头文件）：

```
#include <fstream>
```

为了打开一个仅被用于输出的文件，我们可以定义一个 ofstream（输出文件流）类对象。例如：

```
ofstream outfile( "copy.out", ios_base::out );
```

传递给 ofstream 构造函数的实参分别指定了要打开的文件名和打开模式。一个 ofstream 文件可以被打开为输出模式(ios_base::out)或附加模式(ios_base::app)。(在缺省情况下，一个 ostream 文件以输出模式打开。)outfile2 的定义与 outfile 的定义等价：

```
// 缺省情况下，以输出模式打开
ofstream outfile2( "copy.out" );
```

如果在输出模式下打开已经存在的文件，则所有存储在该文件中的数据都将被丢弃。如果我们希望增加而不是替换现有文件中的数据，则我们应该以附加模式打开文件。于是，新写到文件中的数据将被添加到文件尾部。在这两种模式下，如果文件不存在，则程序都会创建一个新文件。

在试图读写文件之前，先判断它是否已被成功打开，这总是一个不错的主意，我们可以按如下方式测试 outfile：

```
if ( ! outFile ) { // 打开失败
     cerr << "cannot open "copy.out" for output\n";
     exit( -1 );
}
```

ofstream 从 ostream 类派生。所有 ostream 操作都可以被应用到一个 ofstream 类对象上，例如：

```
char ch = ' ';
outFile.put( '1' ).put( ')' ).put( ch );
outFile << "1 + 1 = " << (1 + 1) << endl;
```

向outfile中插入：

```
1) 1 + 1 = 2
```

以下的程序从标准输入获取字符，并将它输出到 copy.out 中：

```
#include <fstream>

int main()
{
    // 打开文件copy.out用于输出
    ofstream outFile( "copy.out" );

    if ( ! outFile ) {
         cerr << "Cannot open "copy.out" for output\n";
        return -1 ;
    }

    char ch;
    while ( cin.get( ch ) )
         outFile.put( ch );
}
```

用户定义的输出操作符实例也可以被应用到 ofstream 类对象上。下面的程序调用了上节定义的 WordCount 输出操作符：

```
#include <fstream>
#include "WordCount.h"

int main()
{
    // 打开文件word.out用于输出
    ofstream oFile( "word.out" );

    // 在这里测试是否打开成功...

    // 手工创建和设置WordCount对象
    WordCount artist( "Renoir" );
    artist.found( 7, 12 ); artist.found( 34, 18 );
```

```
    // 调用 operator <<(ostream&, const WordCount&);
    oFile << artist;
}
```

为了打开一个仅被用于输入的文件，我们可以使用 ifstream **类对象。**ifstream **类从** istream **类派生。下面的程序读取一个由用户指定的文件，并将内容写入到标准输出上：**

```
#include <fstream>

int main()
{
    cout << "filename: ";
    string file_name;

    cin >> file_name;

    // 打开文件 copy.out用于输入
    ifstream inFile( file_name.c_str() );

    if ( !inFile ) {
         cerr << "unable to open input file: "
             << file_name << " -- bailing out!\n";
        return -1;
    }

    char ch;
    while ( inFile.get( ch ))
           cout.put( ch );
}
```

下面的程序读入我们的 Alice Emma **文本文件、用** filter_string()**过滤它（**Alice Emma **文本以及** filter_string()**的定义见** 20.2.1 **节）、排序字符串、去掉重复单词，然后把结果文本写入到一个输出文件中：**

```
#include <fstream>
#include <iterator>
#include <vector>
#include <algorithm>

template <class InputIterator>
void filter_string( InputIterator first, InputIterator last,
                    string filt_elems = string("\",?."))
{
    for ( ; first != last; first++ )
    {
         string::size_type pos = 0;
         while ((pos=(*first).find_first_of(filt_elems,pos))
                    != string::npos )
```

```
                            (*first).erase( pos, 1 );
        }
    }

    int main()
    {
         ifstream infile( "alice_emma" );

         istream_iterator<string> ifile( infile );
         istream_iterator<string> eos;

         vector< string > text;
         copy( ifile, eos, inserter( text, text.begin() ));

         string filt_elems( "\",.?;:" );
         filter_string( text.begin(), text.end(), filt_elems );

         vector<string>::iterator iter;

         sort( text.begin(), text.end() );
         iter = unique( text.begin(), text.end() );
         text.erase( iter, text.end() );

         ofstream outfile( "alice_emma_sort" );

         iter = text.begin();
         for ( int line_cnt = 1; iter != text.end();
               ++iter, ++line_cnt )
        {
             outfile << *iter << " ";
             if ( ! ( line_cnt % 8 ))
                 outfile << '\n';
         }
         outfile << endl;
    }
```

编译并运行该程序，产生如下输出：

```
A Alice Daddy Emma Her I Shyly a
alive almost asks at beautiful bird blows but
creature fiery flight flowing hair has he her
him in is it like long looks magical
mean more no red same says she shush
such tell tells the there through time to
untamed wanting when wind
```

在定义 ifstream **和** ofstream **类对象时，我们也可以不指定文件。以后通过成员函数** open() **显式地把一个文件连接到一个类对象上。例如：**

```
ifstream curFile;
// ...
curFile.open( filename.c_str() );
if ( ! curFile ) // open failed?
    // ...
```

我们可以通过成员函数 close()**断开一个文件与程序的连接。例如：**

```
curFile.close();
```

在下面的程序中，用同一个 ifstream **类对象依次打开和关闭了五个文件：**

```
#include <fstream>

const int fileCnt = 5;
string fileTabl[ fileCnt ] = {
      "Melville","Joyce","Musil","Proust","Kafka"
};

int main()
{
    ifstream inFile; // 没有连接任何文件

    for ( int ix = 0; ix < fileCnt; ++ix )
    {
         inFile.open( fileTabl[ix].c_str() );
         // ... 判断是否打开成功
         // ... 处理文件
         inFile.close();
    }
}
```

一个 fstream **类对象可以打开一个被用于输出** ***或者*** **输入的文件。**fstream **类从** iostream **类派生而来。在下面的例子中，使用** fstream **类对象文件，对** word.out **先读后写。文件** word.out**，在本节开始时被创建，它包含一个** WordCount **对象。**

```
#include <fstream>
#include "WordCount.h"

int main()
{
    WordCount wd;
    fstream file;

    file.open( "word.out", ios_base::in );
    file >> wd;
    file.close();

    cout << "Read in: " << wd << endl;
```

```
    // ios_base::out 将丢弃当前的数据
    file.open( "word.out", ios_base::app );
    file << endl << wd << endl;
    file.close();
}
```

一个 fstream 类对象也可以打开一个同时被用于输入*和*输出的文件。例如，下面的定义以输入和输出模式打开 word.out：

```
fstream io( "word.out", ios_base::in|ios_base::app );
```

按位或（OR）操作符被用来指定一种以上的模式。通过 seekg()或 seekp()成员函数，我们可以对 fstream 类对象重新定位（g 表示为了获取（getting）字符而定位[用于 ifstream 类对象]，而 p 表示为放置字符（putting）而定位[用于 ofstream 类对象]）。这些函数移动到文件中的一个“绝对”地址，或者从特定位置移动一个偏移。seekg()和 seekp()有以下两种形式：

```
// 设置到文件中固定的位置上
seekg( pos_type current_position );

// 从当前位置向某个方向进行偏移
seekg( off_type offset_position, ios_base::seekdir dir );
```

在第一个版本中，当前位置被设置为由 current_position 指定的某个固定的位置，这里 0 是文件的开始。例如，如果一个文件由下列字符

```
abc def ghi jkl
```

构成，则下列调用

```
io.seekg( 6 );
```

把 io 重新定位到字符位置 6，在我们的例子中即字符 f。第二种形式使用一个偏移来重新定位文件，该偏移值或者是从当前位置开始计算，或者是到文件开始处的偏移，或者是从文件尾部倒退向后计算；这由第二个实参来指定。dir 可以被设置为以下选项之一：

1．ios_base::beg，文件的开始。

2．ios_base::cur，文件的当前位置。

3．ios_base::end，文件的结尾。

在下面的例子中，在每次迭代时，seekg()的每次调用都将文件重新定位在第 i 个记录项上：

```
for ( int i = 0; i < recordCnt; ++i )
      readFile.seekg( i * sizeof(Record), ios_base::beg );
```

第一个实参可以被指定为负数。例如，下面语句从当前位置向后移动 10 个字节：

```
readFile.seekg( -10, ios_base::cur );
```

fstream 文件中的当前位置由下面两个成员函数之一返回：tellg()或 tellp()（‘p’表示 putting——用在 ofstream 上，‘g’表示 getting——用在 ifstream 上)。例如：

```
// 标记出当前位置
ios_base::pos_type mark = writeFile.tellp();

// ...
if ( cancelEntry )
     // 返回到原先标记的位置上
     writeFile.seekp( mark );
```

如果程序员希望从文件的当前位置向前移动一个 Record**，则可以有以下两种写法：**

```
// 等价的做法：用于重新定位的seek调用
readFile.seekg( readFile.tellg() + sizeof(Record) );

// 这种方法被认为效率更高
readFile.seekg( sizeof(Record), ios_base::cur );
```

我们来更详细地看一个实际的程序设计示例。下面是问题：给定一个要读取的文本文件。我们将计算文件的字节大小，并将它存储在文件尾部。另外，每次遇到一个换行符，我们都将当前的字节大小（包括换行符）存储在文件末尾。例如，已知文本文件：

```
abcd
efg
hi
j
```

程序应该生成下面修改之后的文本文件：

```
abcd
efg
hi
j
5 9 12 14 24
```

下面是我们的原始实现：

```
#include <iostream>
#include <fstream>

main()
{
    // 以输入和附加模式打开
    fstream inOut( "copy.out", ios_base::in|ios_base::app );
    int cnt = 0;   // byte count
    char ch;

    while ( inOut.get( ch ) )
    {
```

```
            cout.put( ch ); // 在终端回显
            ++cnt;
            if ( ch == '\n' ) {
                 inOut << cnt ;
                inOut.put( ' ' ); // 空格
            }
        }

        // 输出最终的字节数
        inOut << cnt << endl;
        cout << "[ " << cnt << " ]" << endl;
        return 0;
    }
```

inOut 是附在文件 copy.out 上的 fstream 类对象，它以输入和附加(append)两种模式打开。以附加模式打开的文件将把数据写到文件尾部。

每次读入一个字符时，包括空白字符但不包括文件结束符，我们把 cnt 加 1 并在用户终端上回显这个字符。将输入的字符回显到终端上，目的是我们可以看清程序是否像期望的那样工作。

每次遇到换行符时，我们都把 cnt 当前值写到 inQut 中。读到文件结束符则终止循环。我们把 cnt 的最后值写到 inQut 和屏幕上。

我们编译程序。它似乎是正确的。我们用的文件含有 Moby Dick 的前几个句子，这是十九世纪美国小说家 Herman Melville 写的作品：

```
Call me Ishmael.  Some years ago, never mind
how long precisely, having little or no money
in my purse, and nothing particular to interest
me on shore, I thought I would sail about a little
and see the watery part of the world.  It is a
way I have of driving off the spleen, and
regulating the circulation.
```

程序执行时，产生如下输出：

```
[ 0 ]
```

没有任何字符被显示出来，程序认为文本文件是空的。这显然不正确。我们对某些基本的概念理解有误。请别着急，也别沮丧，现在要做的事情就是仔细地想一想。问题在于，文件是以附加（append）模式打开的，所以它一开始就被定位在文件尾。当

```
inOut.get( ch )
```

执行时，遇到了文件结束符，while 循环终止，因此 cnt 的值是 0。

虽然程序执行的结果很糟糕，但是，一旦我们找到了问题，解决的方案也就很简单了。我们所要做的，就是在开始读之前，把文件重新定位到文件的开始处。语句

```
inOut.seekg( 0 );
```

实现的正是这样的效果。重新编译并运行程序。这次产生如下输出：

```
Call me Ishmael.  Some years ago, never mind
[ 45 ]
```

它只为文本文件的第一行产生了显示和计数结果。余下的六行都被忽略了。唉，谁说程序设计很容易？这是程序员成长过程的一部分，尤其是当我们正在学习新东西的时候。（有时候记录下我们误解或做错的事情非常有用——尤其是以后当我们面对比我们更没有经验的人而失去耐心的时候。）现在我们需要做一个深呼吸，想想我们正在努力做什么事情，以及我们显然做了什么事情。你能发现问题吗？

问题在于，文件以附加（append）模式被打开。第一次写 cnt 时，文件被重新定位到文件末尾，后面的 get()遇到文件结束符，因此结束了 while 循环。

这次的解决方案是把文件重新定位到写 cnt 之前的位置上，这可以用下面两条语句来完成：

```
// 标记出当前位置
ios_base::pos_type mark = inOut.tellg();
inOut << cnt << sp;
inOut.seekg( mark ); // 恢复位置
```

重新编译并执行程序，终端上的输出结果是正确的。但是，检查输出文件，发现它仍然不对：最终字节数虽然已经被写到终端上，但是*没有*写到文件中。while 循环后面的输出操作符没有被执行。

这次的问题是，inOut 是处于“遇到文件结束符”的状态。只要 inOut 处于这种状态，就*不能*再执行输入和输出操作。方案是调用 clear()，清除文件的这种状态。这可以用以下语句来完成：

```
inOut.clear(); // zero out state flags
```

完整的程序如下：

```
#include <iostream>
#include <fstream>

int main()
{
    fstream inOut( "copy.out", ios_base::in|ios_base::app );
    int cnt=0;
    char ch;

    inOut.seekg(0);
    while ( inOut.get( ch ) )
    {
        cout.put( ch );
        cnt++;
        if ( ch == '\n' )
        {
            // mark current position
            ios_base::pos_type mark = inOut.tellg();
            inOut << cnt << ' ';
            inOut.seekg( mark ); // restore position
        }
```

```
        }
        inOut.clear();
        inOut << cnt << endl;

        cout << "[ " << cnt << " ]\n";
        return 0;
    }
```

重新编译并执行程序，终于产生正确的输出。我们在实现程序时的一个错误是，没有为需要支持的行为提供一条显式的语句。每一个后续的方案都是针对每一个出现的问题，而不是先分析整个问题，然后提出完整的方案。虽然我们得到了同样的结果，但是与“开始时就全部认真地考虑问题”相比较，我们付出了更多的劳动和代价。

练习 20.12

请使用练习 20.10 的 Date 类或练习 20.11 的 CheckoutRecord 类定义的输出操作（都在 20.5 节），写一个程序来创建一个输出文件，并将数据写入其中。

练习 20.13

请写一个程序打开并读取练习 20.12 中创建的文件，在标准输出上显示文件的内容。

练习 20.14

请写一个程序，打开练习 20.12 创建的文件，同时用于输入和输出。并且分别在以下位置输出一个 Date 类或 CheckoutRecord 类的实例：（a）在文件的开始处，（b）在第二个已有的对象之后，（c）在文件末尾。

20.7 条 件 状 态

一般来说，作为 iostream 库的用户，我们主要关心的是，一个流是否处于非错误状态。例如，如果我们写：

```
int ival;
cin >> ival;
```

并键入“Borges”，那么，在试图把一个字符串文字赋值给一个整型变量失败后，cin 被设置为一种错误状态。如果我们键入 1024，则读入成功，cin 仍处于好的状态。只有在好的状态时，输入流才执行读操作。

为了判断流对象是否处于好的状态，我们通常只是测试它的真值：

```
if ( !cin )
    // 读操作失败或遇到文件尾
```

为了读入未知数目的元素，通常我们写 while 循环如下：

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

```
// ok: 读操作成功
```

当到达文件结束符，或者在读操作期间发生错误条件时，while 循环的条件测试为 false。大多数情况下，对于流对象，这种形式的 true/false 结果已经足够了。但是，在实现 20.5 节中的 Word 输入操作符中，我们需要更细致地访问流的条件状态。

每个流对象都维护了一组条件标志，通过这些条件标志，我们可以监视流的当前状态。我们可以调用以下四个谓词成员函数：

1．如果一个流遇到文件结束符，则 eof()返回 true。例如：

```
if ( inOut.eof() )
  // ok, everything read in ...
```

2．如果试图做一个无效的操作，比如 seeking 重定位操作超出了文件尾，则 bad()返回 true。一般地，这表示该流由于某种未定义的方式而被破坏了。

3．如果操作不成功，比如打开一个文件流对象失败或遇到一种无效的输入格式，则 fail()返回 true。例如：

```
ifstream iFile( filename, ios_base::in );

if ( iFile.fail() ) // 不能打开
      error_message( ... );
```

4．如果其他条件都不为 true，则 good()返回 true。例如：

```
if ( inOut.good() )
```

显式地修改一个流对象的条件状态有两种方式。第一，使用 clear()成员函数，我们可以把条件状态复位到一个显式的值。第二，使用 setstate()成员函数，我们可以不复位条件状态，而是在对象现有条件状态的基础上再增加一个条件。例如，在 WordCount 类的输入操作符中，当遇到一种无效格式时，我们用 setstate()为 istream 对象增加一个失败的条件：

```
if ((ch = is.get()) != '<' )
{
    is.setstate( ios_base::failbit );
    return is;
}
```

所有可用的条件值如下：

```
ios_base::badbit
ios_base::eofbit
ios_base::failbit
ios_base::goodbit
```

为了设置多个条件状态，我们可以如下使用按位 OR 操作符：

```
is.setstate( ios_base::badbit | ios_base::failbit );
```

在测试 20.5 节中的 WordCount 输入操作符时，我们写过

```
if ( !( cin >> readIn ))
{
     cerr << "WordCount input error" << endl;
     exit( -1 );
}
```

作为其他的选择方案，我们或许会希望继续我们的程序，或许警告用户发生了输入错误，并要求再次输入。为了从 cin 中读取其他的输入，我们必须将它重新置于好的状态，我们可以用 clear()成员函数来完成：

```
cin.clear(); // resets cin to good
```

更一般地，clear()被用来清除一个流对象的现有条件状态，并且设置 0 个或多个新的条件状态。例如：

```
cin.clear( ios_base::goodbit );
```

这可以显式地使 cin 恢复为好的状态。（上面这两个调用是等价的，因为 clear()调用的缺省值是 goodbit 值。）

rdstate()成员函数使我们能够显式地访问一个 iostream 类对象的状态，例如：

```
ios_base::iostate old_state = cin.rdstate();

cin.clear();
process_input();

// 现在cin被重置为原来的状态
cin.clear( old_state );
```

练习 20.15

请改写练习 20.7 中的 Date 类和练习 20.8 中的 CheckoutRecord 类的输入操作符，以便设置 istream 对象的条件状态。也请修改用来练习操作符的程序，以便检查显式设置的条件状态，并且条件状态一旦被报告，就复位 istream 对象的条件状态。通过提供正常的和错误的格式，来练习修改后的程序。

20.8　string 流（字符串流）

iostream 库支持在 string 对象上的内存操作。ostringstream 类向一个 string 插入字符。istringstream 类从一个 string 对象读取字符。stringstream 类可以用来支持读和写两种操作。为了使用 string 流，我们必须包含相关的头文件：

```
#include <sstream>
```

例如，下面的函数把整个文件 alice_emma 读到一个 ostringstream 类对象 buf 中。buf 随输入的字符而增长，以便容纳所有的字符：

```
#include <string>
#include <fstream>
#include <sstream>

string read_file_into_string()
{
    ifstream ifile( "alice_emma" );
    ostringstream buf;

    char ch;
    while ( buf && ifile.get( ch ))
             buf.put( ch );

    return buf.str();
}
```

成员函数 str()返回与 ostringstream 类对象相关联的 string 对象。我们可以用与处理普通 string 对象一样的方式，来操纵这个 string 对象。例如，在下面的程序中，我们用与 buf 关联的 string 对象，按成员初始化 text：

```
int main()
{
    string text = read_file_into_string();

    // 标记出文本中每个换行符的位置
    vector< string::size_type > lines_of_text;
    string::size_type pos = 0;

    while ( pos != string::npos )
    {
        pos = text.find( '\n', pos );
        lines_of_text.push_back( pos );
    }

    // ...
}
```

ostringstream 对象也可以用来支持复合 string 的自动格式化；即由多种数据类型构成的字符串。例如，输出操作符自动地将类型转换成相应的字符串表示，而无需担心所需的存储区大小：

```
#include <iostream>
#include <sstream>

int main()
{
```

```
    int ival    = 1024;    int *pival    = &ival;
    double dval = 3.14159; double *pdval = &dval;

    ostringstream format_message;

   // ok: 把值转换成 string 表示
    format_message << "ival: " << ival
         << " ival's address: " << pival << '\n'
         << "dval: " << dval
         << " dval's address: " << pdval << endl;

    string msg = format_message.str();
    cout << " size of message string: " << msg.size()
         << " message: " << msg << endl;

}
```

在某些环境下，把非致命的诊断错误和警告集中在一起而不是在遇到的地方显示出来，前者要更受欢迎。一种简单的办法可以做到这一点，那就是提供一组一般形式的格式化重载函数：

```
string
format( string msg, int expected, int received )
{
    ostringstream message;
    message << msg << " expected: " << expected
           << " received: " << received << "\n";
    return message.str();
}

string format( string msg, vector<int> *values );
// ... and so on
```

于是，应用程序可以存储这些字符串，便于以后显示，或许可以按严重程度进行分类。一般地，它们可能会被分为 Notify、Log 或 Error 类。

istringstream 由一个 string 对象构造而来，它可以读取该 string 对象。istringstream 的一个用法是，将数值字符串转换成算术值。例如：

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

int main()
{
    int ival    = 1024;    int *pival    = &ival;
    double dval = 3.14159; double *pdval = &dval;

    // 创建一个字符串，存储每个值
    // 用空格作为分割
    ostringstream format_string;
```

```
    format_string << ival << " " << pival << " "
                  << dval << " " << pdval << endl;

    // 提取出被存储起来的ascii值
    // 把它们依次放在四个对象中
    istringstream input_istring( format_string.str() );
    input_istring >> ival >> pival
                  >> dval >> pdval;
}
```

练习 20.16

在 C **中，一个输出消息的格式化是使用标准** C **的** printf()**函数族。例如，下列代码段**

```
int    ival = 1024;
double dval = 3.14159;
char   cval = 'a';
char  *sval = "the end";

printf( "ival: %d\tdval: %g\tcval: %c\tsval: %s",
        ival, dval, cval, sval );
```

产生：

```
ival: 1024dval: 3.14159 cval: a   sval: the end
```

printf()**的第一个实参是一个格式字符串。每个**%**字符表示，它将被一个实参值替代；**%**后面的字符表示它的类型。下面是** printf()**支持的某些可能的类型：**

```
%d     integer
%g     floating point
%c     char
%s     C-style string
```

（完整的讨论见[KERNIGHAN88]**。）**

printf()**的其他实参与每个“** %**格式对”按位置一一匹配，格式字符串的其他字符被视为文字常量，直接被输出。**

printf()**函数族的两个主要缺点是：（**1**）格式化字符串不能被扩展，因而不能识别用户定义的类型，（**2**）如果实参的类型和数目与格式字符串不匹配，则这样的错误不能被检测出来，输出将很难看。**printf()**函数族的主要好处是，格式字符串非常紧凑。**

（a**） 请用一个** ostringstream **对象，产生等价的格式化输出。**

（b**） 请比较两种方法的优缺点。**

20.9 格　式　状　态

每一个 iostream **库对象都维护了一个*格式状态*（*format state*），它控制格式化操作的细节，**

比如整型值的进制基数或浮点数值的精度。C++为程序员提供了一组预定义的操纵符，可用来修改一个对象的格式状态。[1]

操纵符被应用在流对象上的方式，就好像它们是数据一样。但是，操纵符不导致读写数据，而是修改流对象的内部状态。例如，缺省情况下，true 值的 bool 对象被写成整数值 1：

```
#include <iostream>

int main()
{
    bool illustrate = true;;
    cout << "bool object illustrate set to true: "
       << illustrate << '\n';
}
```

为了修改 cout，使它能够把 illustrate 显示为 true，我们应用 boolalpha 操纵符：

```
#include <iostream>

int main()
{
    bool illustrate = true;;
    cout << "bool object illustrate set to true: ";

    // 改变cout的状态
    // 用字符串true和false输出bool值
    cout << boolalpha;
    cout << illustrate << '\n';
}
```

因为操纵符被应用之后，仍然返回原来被应用的流对象，所以我们可以把它的应用与数据的应用连接起来（或者与其他操纵符的应用连接起来）。下面是重写之后的小程序，它混合了数据和操纵符：

```
#include <iostream>

int main()
{
    bool illustrate = true;
    cout << "bool object illustrate: "
        << illustrate
        << "\nwith boolalpha applied: "
         << boolalpha << illustrate << '\n';

    // ...
```

[1] 另外，程序员还可以使用成员函数 setf() 和 unsetf()，直接设置或解除格式状态标志。我们没有介绍这些操作。关于这种方法的讨论见[STROUSTRUP97].

```
}
```

像这样，把操纵符和数据混合起来容易产生误导作用。应用操纵符之后，不只改变了后面输出值的表示形式，而且修改了 ostream 的内部格式状态。在我们的例子中，整个程序的余下部分都将把 bool 值显示为 true 或 false。

为了消除对 cout 的修改，我们必须应用 noboolalpha 操纵符：

```
cout << boolalpha   // 设置cout的内部状态
     << illustrate
     << noboolalpha // 解除cout内部状态
```

我们将会看到，许多操纵符都有类似的“设置/消除（set/unset）”对 。

缺省情况下，算术值以十进制形式被读写。程序员可以通过使用 hex、oct 和 dec 操纵符，把整数值的进制基数改为八进制或十六进制，或改回十进制（浮点值的表示不受影响）。例如：

```
#include <iostream>
int main()
{
int ival = 16;
double dval = 16.0;

cout << "ival: " << ival
     << " oct set: " << oct << ival << "\n";

cout << "dval: " << dval
     << " hex set: " << hex << dval << "\n";

cout << "ival: " << ival
     << " dec set: " << dec << ival << "\n";
}
```

编译并执行程序，产生下列输出：

```
ival: 16 oct set: 20
dval: 16 hex set: 16
ival: 10 dec set: 16
```

我们这个程序的一个问题是，我们在看到一个值的时候无法知道它的进制基数。例如，20 是真正的 20，还是 16 的八进制表示？操纵符 showbase 可以让一个整数值在输出时指明它的基数，形式如下：

1．0x 开头表明是十六进制数（如果希望显示为大写字母，则可以应用 uppercase 操纵符；为了转回小写的 x，我们可以应用 nouppercase 操纵符）。

2．以 0 开头表示八进制数。

3．没有任何前导字符，表示十进制数。

下面是用 showbase 改写过的程序：

```
#include <iostream>
int main()
{
    int ival = 16;
    double dval = 16.0;

    cout << showbase;

    cout << "ival: " << ival
         << " oct set: " << oct << ival << "\n";

    cout << "dval: " << dval
         << " hex set: " << hex << dval << "\n";

    cout << "ival: " << ival  << " dec set: "
         << dec << ival << "\n";

    cout << noshowbase;
}
```

下面是修改后的输出：

```
ival: 16 oct set: 020
dval: 16 hex set: 16
ival: 0x10 dec set: 16
```

noshowcase **操纵符重新设置** cout**，使它不再显示整数值的进制基数。**

缺省情况下，浮点值有 6 **位的精度。这个值可以用成员函数** precision(int)**或流操纵符** setprecision()**来修改（若使用后者，则必须包含** iomanip **头文件）。**precision()**返回当前的精度值。例如：**

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

int main()
{
    cout  << "Precision: "
          << cout.precision() << endl
          << sqrt(2.0) << endl;

    cout.precision(12);
    cout  << "\nPrecision: "
    << cout.precision() << endl
    << sqrt(2.0) << endl;

    cout  << "\nPrecision: " << setprecision(3)
    << cout.precision() << endl
```

```
    << sqrt(2.0) << endl;

    return 0;
}
```

编译并执行程序，产生以下输出：

```
Precision: 6
1.41421

Precision: 12
1.41421356237

Precision: 3
1.41
```

带有一个实参的操纵符，比如前面见到的 setprecision()**和** setw()**，要求包含** iomanip **头文件：**

```
#include <iomanip>
```

我们的例子没有说明 setprecision()**的两个更深入的方面：**1**）整数值不受影响，**2**）浮点值被四舍五入而不是被截取。因此当精度为** 4 **时，**3.14159 **变成** 3.142**，精度为** 3 **时变成** 3.14**。**

缺省情况下，当小数部分为 0 **时，不显示小数点。例如：**

```
cout << 10.00
```

输出为

```
10
```

为了强制显示小数点，我们使用 showpoint **操纵符：**

```
cout << showpoint
     << 10.0
     << noshowpoint << '\n';
```

noshowpoint **操纵符重新设置缺省行为。**

缺省情况下，浮点值以定点小数法显示。为了改变为科学计数法，我们使用 scientific **操纵符。为了改回到定点小数法，我们使用** fixed **操纵符：**

```
cout << "scientific: " << scientific
     << 10.0
     << "fixed decimal: " << fixed
     << 10.0 << '\n';
```

这产生

```
scientific: 1.0e+01
fixed decimal: 10
```

如果希望把‘e’输出为‘E’，我们可以使用 uppercase 操纵符。要转回小写字母，我们使用 nouppercase 操纵符。(uppercase 操纵符不会使所有字母字符都显示为大写！)。

缺省情况下，重载的输入操作符跳过空白字符（空格、制表符、换行符、走纸、回车）。已知序列

```
a b  c
d
```

循环

```
char ch;
while ( cin >> ch )
       // ...
```

执行四次，以读入从 a 到 d 的四个字符，跳过中间的空格、可能的制表符和换行符。操纵符 noskipws 使输入操作符不跳过空白字符：

```
char ch;
cin >> noskipws;
while ( cin >> ch )
       // ...
cin >> skipws;
```

现在 while 循环需要迭代七次，才能读入字符 a 到 d。为了转回到缺省行为，我们在 cin 上应用操纵符 skipws。

当我们写

```
cout << "please enter a value: ";
```

文字字符串被存储在与 cout 相关联的缓冲区中。有许多种情况可以引起缓冲区被刷新——即，清空——在我们的例子中，也就是将缓冲区写到标准输出上：

1．缓冲区可能会满，在这种情况下，它必须被刷新，以便读取后面的值。

2．我们可通过显式地使用 flush、ends 或 endl 操纵符来刷新缓冲区。

```
// 清空缓冲区
cout << "hi!" << flush;

// 插入一个空字符然后刷新缓冲区
char ch[2]; ch[0] = 'a'; ch[1] = 'b';
cout << ch << ends;

// 插入一个换行符然后刷新缓冲区
cout << "hi!" << endl;
```

3．unitbuf，一个内部的流状态变量，若它被设置，则每次输出操作后都会清空缓冲区。

4．一个 ostream 对象可以被*捆绑*到一个 istream 上，在这种情况下，当 istream 从输入流读取

数据时，ostream 的缓冲区就会被刷新。cout 被预定义为“捆绑”在 cin 上：

```
cin.tie( &cout );
```

语句

```
cin >> ival;
```

使得与 cout 相关联的缓冲区被刷新。

一个 ostream 对象一次只能被捆绑到一个 istream 对象上，为了打破现有的捆绑，我们可以传递一个实参 0。例如：

```
istream is;
ostream new_os;

// ...

// tie() 返回现有的捆绑
ostream *old_tie = is.tie();

is.tie( 0 ); // 打破现有的捆绑
is.tie( &new_os ); // 设置新的捆绑

// ...

is.tie( 0 ); // 打破现有的捆绑
is.tie( old_tie ); // 重新建立原来的捆绑
```

我们可以用 setw()操纵符来控制数字或字符串值的宽度。例如程序

```
#include <iostream>
#include <iomanip>

int main()
{
   int ival = 16;
   double dval = 3.14159;

   cout << "ival: " << setw(12) << ival << '\n'
        << "dval: " << setw(12) << dval << '\n';
}
```

产生以下输出：

```
ival:         16
dval:     3.14159
```

第二个 setw()是必需的，因为不像其他操纵符，setw()不修改 ostream 对象的格式状态。要想使输出的值左对齐，我们可以应用 left 操纵符（通过 right 操纵符可以重新设回到缺省

状态）。如果我们希望产生

```
   16
-   3
```

我们可以应用 internal 操纵符，它使得正负符号左对齐，而值右对齐，中间添加空格。如果希望用其他字符填充中间的空白，则可以应用 setfill()操纵符。

```
cout << setw(6) << setfill('%') << 100 << endl;
```

产生

```
%%%100
```

预定义的所有操纵符都被列在表 20.1 中。

表 20.1　操纵符

操　纵　符	含　　义
boolalpha	把true 和 false 表示为字符串
*noboolalpha	把true 和 false 表示为0、1
showbase	产生前缀，指示数值的进制基数
*noshowbase	不产生进制基数前缀
showpoint	总是显示小数点
*noshowpoint	只有当小数部分存在时才显示小数点
Showpos	在非负数值中显示 +
*noshowpos	在非负数值中不显示 +
*skipws	输入操作符跳过空白字符
noskipws	输入操作符不跳过空白字符
uppercase	在十六进制下显示 0X，科学计数法中显示E
*nouppercase	在十六进制下显示 0x，科学计数法中显示e
*dec	以十进制显示
hex	以十六进制显示
oct	以八进制显示
left	将填充字符加到数值的右边
right	将填充字符加到数值的左边

续表

操　纵　符	含　　义
Internal	将填充字符加到符号和数值的中间
*fixed	以小数形式显示浮点数
scientific	以科学计数法形式显示浮点数
flush	刷新ostream缓冲区
ends	插入空字符，然后刷新ostream缓冲区
endl	插入换行符，然后刷新ostream缓冲区
ws	“吃掉”空白字符
// 以下这些要求 #include <iomanip>	
setfill(ch)	用ch填充空白字符
setprecision(n)	将浮点精度设置为 n
setw(w)	按照w个字符来读或者写数值
setbase(b)	以进制基数 b 输出整数值
* 表示缺省的流状态	

20.10　一个强类型库

iostream 库是强类型的。例如，试图从一个 ostream 读数据，或者写数据到一个 istream，都会在编译时刻被捕获到，并标记为类型违例。例如，已知下列声明

```
#include <iostream>
#include <fstream>
class Screen;

extern istream& operator>>( istream&, const Screen& );
extern void print( ostream& );
ifstream inFile;
```

下面的两条语句都将导致编译时刻类型违例：

```
int main()
{
    Screen myScreen;
```

```
    // 错误：期望一个 ostream&
    print( cin >> myScreen );

    // 错误：期望 >> operator
    inFile << "error: output operator";
}
```

输入/输出设施是 C++**标准库的一个组件。第** 20 **章没有描述整个** iostream **库——特别是，像“创建用户定义的操纵符和缓冲区类”这样的主题超出了本书的范围。我们关注的是** iostream **库的基础部分，通过这些基础部分程序员可以提供基本的输入/输出功能。**