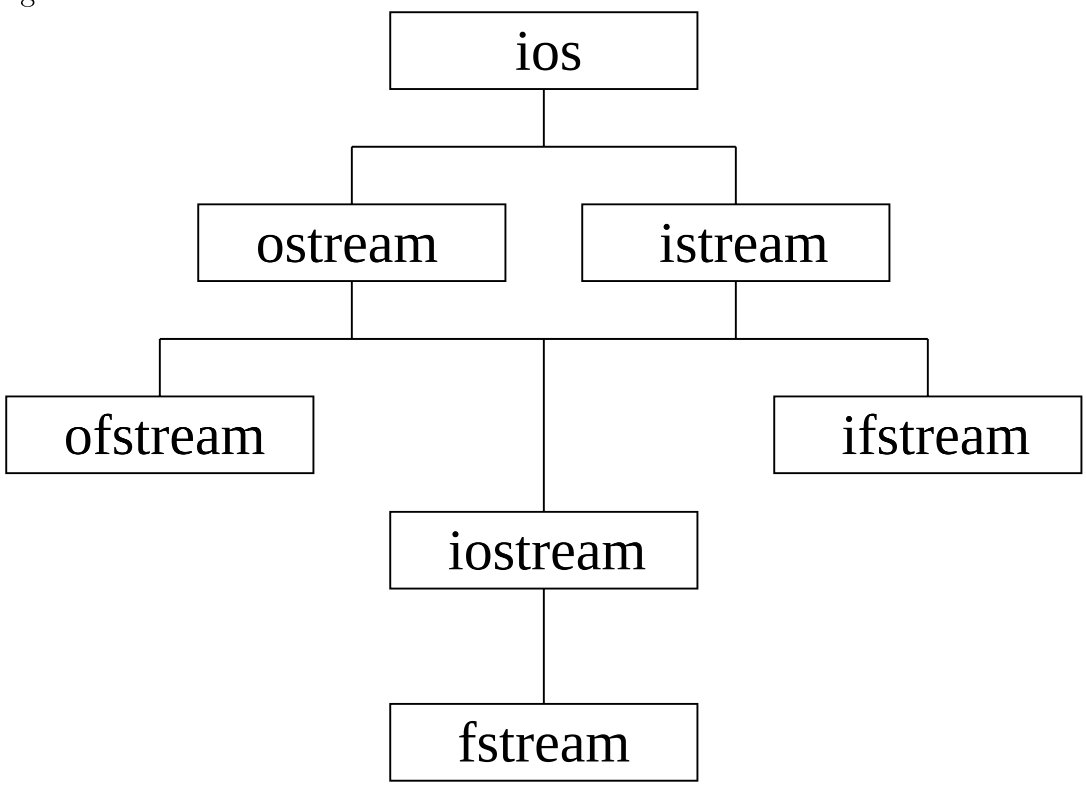
LECTURE 14
Input and Output Streams

Classes dealing with the input and output of data are called *stream* classes. We have dealt with these classes in a slightly haphazard way. I'd like to talk about them in a more systematic way to give you a better idea of how input and output works. This won't be a complete discussion since that would take too long. There is a chapter on streams in Stroustrup and the first third of *More C++ for Dummies* deals with streams. Most of this lecture comes from *More C++ for Dummies*. The class hierarchy is shown in the figure.



The mother of all base classes is `ios`. (In more recent compilers, `ios` has been replaced by `ios_base`.) The class `ios` contains most of the actual I/O code. It is `ios` that keeps track of the error state of the stream. The error flags are an enumerated type within `ios`. In addition the `ios` class converts data for display. It understands the format of the different types of numbers, how to output character strings, and how to convert an ASCII string to and from an integer or a floating–point number.

Standard output, `cout`, is an object of the class `ostream`, as is `cerr`, the standard error output. Standard input, `cin`, is an object of the class `istream`. `cout`, `cin`, and `cerr` are automatically constructed as global objects at program start–up. Objects of `iostream` deal with both input and output. Objects of `ifstream` deal with input files and objects of the class `ofstream` deal with output files. Objects `fstream` deal with files that can one can write to and read from. `ofstream`, `ifstream`, and `fstream` are subclasses that are defined in the include file `fstream.h`. Notice that `fstream.h` deals with file stream classes.

The overloaded right shift operator `operator>>()` is called the extractor. It is a member function of the class `istream`. The overloaded left shift operator `operator<<()` is called the inserter. It is a member function of the class `ostream`. Thus we have

```
//for input
istream& operator>>(istream& source, char *pDest);
istream& operator>>(istream& source, int &dest);
istream& operator>>(istream& source, char &dest);
//...etc...
//for output
ostream& operator<<(ostream& dest, char *pSource);
ostream& operator<<(ostream& dest, int source);
ostream& operator<<(ostream& dest, char source);
```

So when we type

```
#include <iostream.h>

void fn()
{
  cout<<"Hello, world\n";
}
```

First, C++ determines that the left–hand argument is of type `ostream` and the right–hand argument is of type `char*`. Armed with this knowledge, it finds the prototype `operator<<(ostream&, char*)` in `iostream.h`. C++ generates a call to this function, the `char*` inserter, passing the function the string `"Hello, world\n"` and the object `cout` as the two arguments. That is, it calls `operator<<(cout, "Hello, world\n")`. The `char*` inserter function, which is part of the standard C++ library, performs the requested output.

Input and Output Files

We have already seen that to open files to read from and write to:

```
  #include <fstream.h>

   int main() {
    ifstream infile("input.dat");     //input.dat is the name of the file
                                      //in your directory
    ofstream outfile("output.dat");   //output.dat is the name of the file
                                      //that will be created in your directory
    float x;
      while(infile >> x)              //detects end-of-file and exits loop
         { outfile << "x = " << x << endl; }
```

```
   infile.close();
   outfile.close();
   return 0;
  }
```

The statement

```
    ofstream outfile("output.dat");
```

calls one of the constructors of `ofstream`. It constructs the object `outfile` using the argument `"output.dat"`. The constructor that's called is

```
ofstream::ofstream(const char *pFileName,
                   int mode = ios::out,
                   int prot = filebuff::openprot);
```

The first argument is a pointer to the name of the file to open. The second and third arguments specify how the file will be opened. Since we didn't specify the second and third arguments, the default values take effect. Similarly, the statement

```
    ifstream infile("input.dat");
```

calls the following constructor of the class `ifstream`:

```
ifstream::ifstream(const char *pFileName,
                   int mode = ios::in,
                   int prot = filebuff::openprot);
```

The legal values for *mode* are listed in the following table. The Application column indicates which modes are valid for which file types.

Available Modes for Opening a File

Flag	Application	Meaning
ios::app	out	Always append output to the end of the file
ios::ate	out	Open and seek to end-of-file ("at end")
ios::binary	in, out	Open file in binary mode (as opposed to text mode)
ios::in	in	Open a file for input
ios::nocreate	in, out	If file doesn't exist, don't create it
ios::noreplace	out	Don't delete the file (open fails if file

		exists unless you specify app or ate)
ios::out	out	Open file for output
ios::trunc	out	Truncate file to zero length if it already exists (default if file exists and app or ate is not specified)

These modes are bit fields that are enumerated members of a bit vector in the class `ios`. That is why they are referred to as `ios::out` and `ios::app` and not simply as `out` and `app`. Let me give an example of what I mean by "bit fields". `ios::app` might equal 00000001, `ios::ate` might equal 00000010, `ios::out` might equal 00000100, etc. So each mode corresponds to one bit which can be 0 or 1. This means that more than one mode's value can be set at the same time using the arithmetic OR. For example, to open an output file with append, you could use the following:

```
ofstream out("outfile", ios::out | ios::app)
```

The `|` operator takes the union of the two arguments. Once you specify any part of the mode, you must specify the entire mode. Thus, when I specified `ios::app`, I also had to specify `ios::out`, because it was no longer specified by default. The `ios::nocreate` flag says "If file doesn't exist, don't create it." This is especially useful for input. For example, it is the only way to test for the existence of a file.

Let me explain the difference between opening a file in text mode versus binary mode. When a file is in text mode, newline characters are converted into a carriage–return/line–feed combination on output. The reverse process occurs on file input: carriage–return/line–feed pairs are converted into a single newline character. This process doesn't occur when the file is opened in binary mode. The default for opening a file is text mode.

The third argument to the `fstream` constructors specifies the type of file sharing allowed between applications. The possible values are:

File–Sharing Flags

Sharing Flag	Meaning
filebuf::openprot filebuf::sh_compat	Compatibility sharing mode
filebuf::sh_none	Exclusive mode; no sharing allowed
filebuf::sh_read	Other read opens allowed
filebuf::sh_write	Other write opens allowed

These are also bit fields. So the union of `filebuf::sh_read` and `filebuf::sh_write` allows complete sharing of files between applications.

As we have seen in our earlier discussion of input and output, it is also possible to open a file for both input and output. This is handled by the `fstream` class, which inherits from both the `ifstream` and `ofstream` classes. The constructor for the `fstream` class looks the same as those for the `ifstream` and `ofstream` classes except the mode argument is not defaulted:

```
fstream::fstream(const char *pFileName,
                 int mode,
                 int prot = filebuf::openprot);
```

To open such a file, the mode should be set to `ios::in|ios::out`. For example,

```
#include <fstream.h>

 int main() {
  fstream inout("input.dat",ios::in|ios::out);
  float x;
    inout >> x;
    inout<< endl << "x = " << x << endl;
  inout.close();
  return 0;
 }
```

Error Flags

If something goes wrong with the input/output (I/O) operations, the error state is set. Once the error state is set, all subsequent requests for I/O are ignored. The error state stays set until it is reset by the application. This allows the application to perform several I/O operations in row before checking–perhaps at the end of the function–whether the I/O operations succeeded. The error flag consists of a set of bits, each of which can be set independently. These bits are defined as follows. (Note the values are provided to satisfy your curiosity. Don't rely on them. Always use the name of the flag instead.)

Flag	Binary Value	Meaning
ios::eofbit	001	End-of-file encountered
ios::failbit	010	Last I/O operation failed
ios::badbit	100	Invalid operation attempted

When a read operation encounters the end–of–file, it sets the `ios::eofbit` in the error flag for the `ifstream` object. The `ios::failbit` is set when an I/O operation fails.

This could happen if the format of the data is improper. For example, attempting to extract a string of ASCII text into a number sets the `failbit`. In other words, if the program expects a number to be input and you give it a string of letters, `ios::failbit` is set. Attempting to read beyond the end–of–file also sets the `failbit`. Similar to the `ios::failbit`, the `ios::badbit` is set when an operation fails. The two flags differ in that the `badbit` indicates an unrecoverable error, whereas you might be able to recover from the `failbit` (then again, maybe not, but at least you have a chance). For example, attempting to open a file that doesn't exist sets the `badbit`. Although it's interesting to see which bits make up the error flag, you don't actually ever check or set these bits directly. Instead, you use the following access functions:

Functions that Read or Set the Stream Error State

Function	Purpose
`ios::bad()`	Returns TRUE if the badbit is set
`ios::clear(int=0)`	Sets the error flag
`ios::eof()`	Returns TRUE if the eofbit is set
`ios::fail()`	Returns TRUE if either the failbit or the badbit is set
`ios::good()`	Returns TRUE if no error bits are set
`ios::rdstate()`	Returns the error flag
`ios::operator!()`	Same as ios::fail()
`ios::operator void*()`	Same as ios::good(); cast operator

The check functions are primarily `ios::eof()`, `ios::fail()`, and `ios::bad()`. Each of these return TRUE if their respective bit is set (except `ios::fail()`, which returns a TRUE if either the failbit or the badbit is set). The `ios::good()` function returns the inverse of `ios::fail()`. Just about the worst named function in the entire C++ library is `ios::clear()`. This function gets its name from the fact that you use it to clear the error flag. However, you also use it to set an error flag. In fact, `clear()` allows you to set or clear any of the error bits you want. The two overloaded operators are just cute ways of calling `ios::good()` and `ios::fail()`. For example,

```
void fn(istream& in)
{
```

```
    //stuff
    if(!in)      //invokes operator!(), which calls in.fail()
    {
                 //operation failed
      return;
    }
  }
```

Alternatively, we could write the following:

```
void fn(istream& in)
{
   //stuff
   if(in)      //invokes operator void*(), which calls in.good()
   {
                //operation succeeded
    //stuff
   }
  }
```

(Explanation of `operator void*()`. If we had a pointer `istream *pIn` to an `istream` object, then `*pIn` would refer to the object pointed to by `pIn`. In our case `in` is an `istream` object, so we don't need the indirection operator `*`.) Some programmers prefer this form of calling `ios::good()` and `ios::fail()`. A simple example of error checking is

```
#include <fstream.h>

 int main() {
  ifstream infile("input.dat");     //input.dat is the name of the file
                                    //in your directory
  ofstream outfile("output.dat");   //output.dat is the name of the file
                                    //that will be created in your directory
  float x;
    while(!infile.eof() && infile.good()) //makes sure that end-of-file
                                          //hasn't been reached and that
                                          //infile is in good shape
       {
        infile >> x;
        outfile << "x = " << x << endl;
       }
  infile.close();
  outfile.close();
  return 0;
 }
```

(This program outputs the last `x` twice because the end–of–file is not reached until it goes beyond the last item in `input.dat`.)

Other Member Functions of istream and ostream

The `istream` and `ostream` classes support a number of member I/O functions in addition to the overloaded insertion and extraction operators. Some of these follow:

The get() Function

The `get()` function comes in two flavors. The simplest version inputs a single character. For example, in the following program snippet, `get()` is used to read input from the input file input.txt:

```
#include <fstream.h>

 int main() {
  ifstream in("input.txt");
  char c;
  while(!in.eof())
   {
     c = in.get();
     cout << c;
   }
  cout << endl;
 return 0;
 }
```

The program starts by opening the file `input.txt`. The program then loops until the file is empty. On each loop, the program fetches another character and outputs it to the standard output. Notice that `get()` is not quite the same as `operator>>(istream&,char&)`, which also fetches a single character from the input stream. The difference lies in the fact that `operator>>()`, by default, skips any white–space characters (e.g., space, tab, newline, etc.) found in the file, whereas `get()` does not. In fact this program also spits out a strange end–of–file character at the end.

The second version of `get()` carries the following prototype:

```
istream& istream::get(char* pszTarget, int nCount, char delim='\n');
```

This version inputs a series of characters terminated either by the appearance of a terminator character in the input stream or by a character count. Notice that you can have any character you want terminate the input. The count character solves the potential bug of inputting more characters than the buffer can hold. For example, the following code is inherently unsafe because it is entirely possible that the string extracted by `operator>>()` is longer than the 80 characters the buffer can hold:

```
istream in("input.txt");
char buffer[80];
in >> buffer;
```

A safer alternative would be

```
istream in("input.txt");
char buffer[80];
in.get(buffer, 80);
```

Because `get()` now knows the length of the receiving buffer, it will make sure not to extract more characters than the buffer can handle. `get()` gets not more than 80 characters and puts them into the array `buffer`. You can have `sizeof` calculate the buffer for you:

```
istream in("input.txt");
char buffer[80];
in.get(buffer, sizeof buffer);
```

The getline() Function

The prototype declaration for the `getline()` function is identical to that of the `get()` function:

```
istream& istream::getline(char* pszTarget, int nCount, char delim='\n');
```

In execution, `getline()` is identical to the second form of `get()`. The sole exception is that `get()` extracts characters from the input stream up to, but not including, the delimiter, whereas `getline()` extracts the delimiter as well. Neither function stores the delimiter into the `pszTarget` buffer. This makes `getline()` ideal for reading an entire line of input at a time. It reads this line of input from the input file and puts it in the array pointed to by `pszTarget`.

```
#include <fstream.h>

 int main() {
  ifstream in("input.txt");
  char szTarget[256];
     in.getline(szTarget, sizeof szTarget);
     cout << szTarget << "\n";
 return 0;
 }
```

This program reads an entire line at a time. Notice that when the line that was just read is output to `cout`, the program must replace the delimiter that was stripped out by `getline()`.

The read() Function

The prototype declaration for the read function is

```
istream& read(char* pszTarget, int nCount);
```

This function reads a fixed number of characters from the input stream without regard to any type of delimiter. In addition, `read()` doesn't tack a NULL character to the end of the `pszTarget` buffer, nor does it attempt to interpret `'\n'` characters.

The put() Function

The `put()` function carries the following simple prototype:

```
ostream& ostream::put(char ch);
```

This function does nothing more than output the specified character to the output stream. This is functionally identical to the `operator<<(ostream&, char&)` inserter.

The putback(Ch c) function

The `putback(Ch c)` function allows a program to put an unwanted character back to be read some other time, as shown in the class of complex numbers in lecture 6. `Ch` is a template type, i.e., it's any type you specify.

The write() Function

The `write()` function outputs a fixed number of characters from the source character string to the output stream. This function carries the following prototype

```
ostream& ostream::write(const char* pszSource, int nCount);
```

This is also a block–oriented transfer.

The Buffer

We said earlier that the base class `ios` does most of the input/output work. But it needs another class called `streambuf` which acts as a server to the `ios` class. `streambuf` is an intermediatry between `ios` and the physical media, e.g., the screen, the disk, etc. The class `streambuf` performs the actual I/O to the outside world. The class `streambuf` has several subclasses, each of which specializes in its own particular type of media. For example, `filebuf` handles file I/O for the `ios` class. Look at the following code:

```
ofstream out("ofile.txt");
int nAnInt = 10;
out << nAnInt;
```

The constructor for `ofstream` first creates an `ios` object. It then constructs a `filebuf` object for output to the file `ofile.txt`. During output, the `ios` object converts the number 10 into the character 1 followed by the character 0. The `ios` object passes the string "10" to the `filebuf` object for output to the file. This is a nice division of labor. When you create a different type of output object–for example an `ostream` object that outputs to the display–you get the same `ios` base class object (all formatting is the same, after all) but a different subclass of `streambuf` (outputting to a display is not at all the same as outputting to a file).

It's worth taking a moment to understand what disk buffering is. This is one of the functions performed by `streambuf`. If `streambuf` went to the disk every time `ios` wanted to write a few characters to disk, performance would be really slow. In fact, when you read or write to the disk, you must read an entire block of data at a time. (It's like Lay's Potato Chips–you can't eat just one "byte".) The size of a block depends on the disk, but it's usually 512 bytes or more. Therefore, on output, the `streambuf` class collects output requests in the buffer until it has several blocks worth. It then writes the entire buffer to the disk at once. Writing the output buffer to disk is called "flushing the buffer." For example, `endl` ends the line (`"\n"`) and then flushes the buffer.

On input, the situation is reversed. When the `ios` class asks for the first character from the input stream, the input buffer is empty. Rather than read a single character (even if that were possible), the `streambuf` reads several blocks of data into the input buffer. Then `streambuf` returns only the first character to `ios` and keeps the rest. When the next input request comes in, `streambuf` returns the next character from the input buffer without bothering to read from the disk. The `streambuf` class doesn't read from the disk again until the input buffer has been emptied by input requests.

A few conditions cause the output buffer to be flushed to disk early. For example, closing the file causes any remaining data that might be hanging around in the buffer to be flushed to disk. The application program can also force the output buffer to be flushed by calling `ostream::flush()`. For example,

```
out << student;
out.flush()
```

This assures your data is safely on the disk in case the program or the system crashes later on. Finally an output stream can be tied to an input stream so that a request for I/O from the input stream immediately flushes the output stream. For example,

```
char szname[80]'
cout << "Enter your name";
cin >> szName;
```

Things wouldn't work so well if "Enter your name" didn't make it to the display because it was cooling its heels in the output buffer. Tying `cout` to `cin` flushes the output buffer so that "Enter your name" can appear on the screen. Other `iostream` objects are not automatically tied. However, you can tie an `ostream` object to an `iostream` object in such a way that the `ostream` is automatically flushed when an I/O operation is performed on the `iostream`. For example,

```
#include <fstream.h>

int main()
{
  ifstream in("input.dat");
  ofstream out("output.dat");

//By tying out to in, out.flush() will be called whenever an I/O
//operation is performed on in

  in.tie(&out);

// Do some stuff ...

// Now untie out from in

  in.tie(0);
  return 0;
}
```

Notice that you can untie an object by passing a NULL to `tie()`.

Formatting

It is often desirable to format your output by setting the precision or the width, etc. You can do this using the following `ios` member functions:

```
    Function              Purpose
---------------------------------------------------------------
    fill(char)            Set the fill character for padding
                          during output.
---------------------------------------------------------------
    flags(long)           Set the formatting flags.
---------------------------------------------------------------
    precision(int)        Set the precision when outputting a
                          floating-point value
---------------------------------------------------------------
    width(int)            Set the minimum width. Restricts
```

```
                              the number of characters that are
                              input. If fewer characters are required
                              on output, the remaining space is filled
                              with the fill character.
```

Each of these functions also has a `void` argument list version, which simply queries the current setting. The `fill()` function sets the fill character. The default for the fill character is the blank space. The `precision()` function sets the precision. During the display of floating–point numbers, this setting determines the number of digits displayed to the right of the decimal point.

On output, the `width()` function specifies the minimum field width to be used in displaying the next field inserted. If the value being output requires more characters than are specified by `width()`, the width is ignored. If the output field is smaller than the specified width, the difference is made up by repeated application of the fill character. If the width is zero, the minimum number of characters necessary to contain the field are used for output. Zero is the default for the width. The `width()` function is strange in one respect. When you set a data member within a structure to a particular value, you usually expect it to stay set. But this is not so for the width. Each time you set the width, that setting applies only to the next operation. After that, the width is reset to zero.

The `width()` function also has an effect on the input. Setting the width restricts the number of characters extracted by the `char*` extractor. This is important because the `operator>>(istream&, char*)`, unlike the `getline()` function, has no place to indicate the size of the buffer receiving the character string. Settting the width to the size of the buffer ensures that the buffer boundaries are not exceeded.

Here is an example of how to use a few of these format control functions:

```
#include <fstream.h>

int main()
{
  ifstream infile("input.dat");
  if(infile.fail())
   {
    cout << "Couldn't open input.dat" << endl;
    return -1;
   }

  char buffer[5];                    //short array of characters
  infile.width(sizeof buffer);
  infile >> buffer;
  cout << buffer << endl;
```

```
    infile.width(sizeof buffer);      //have to repeat width()
    infile >> buffer;
    cout.width(15);
    cout.fill('*');
    cout << buffer << endl;
    return 0;
  }
```

The program starts by opening the input file in the normal fashion. Before extracting from the input file object, however, the program sets the input width to match the size of the buffer. The program then extracts a few characters into `buffer` and displays them to `cout` so you can see what you have. Suppose the input file consists of 30 characters, all in a row.

```
//input.dat file
123456789012345678901234567890
```

If we just had an `in >> buffer` statement, the program would try to put all 30 characters into the buffer which has length 5. This would crash the program and give a "bus error". The output from running the program is

```
1234
***********5678
```

Notice that width the input stream width set to 5 characters, the extractor reads only four characters, cleverly leaving a space for the NULL in the final position of the buffer.

The remaining formatting features of `ios` are hidden in a protected data member called `x_flags`. This data member consists of a series of 1–bit fields, some of which work together. Because these are single–bit fields, however, they can be (and are) set in different combinations to produce the desired effect. They are listed on pages 89–90 of *More C++ for Dummies*. The only ones that you would probably use with any frequency are those dealing with floating–point format. By setting either the `ios::fixed` or `ios::scientific` flag, you specify whether floating–point numbers are displayed in fixed or scientific format. If neither bit is set, the stream is in automatic mode, meaning use whatever is most applicable. In automatic mode, the precision referred to previously specifies the number of digits to be displayed in the number. In either fixed or scientific mode, precision specifies the number of digits after the decimal point. The default is automatic. Several functions access the flag bits. The function `long ios::flags()` reads the current flag word. The function `long ios::flags(long lNewFlag)` sets the flag word to the value contained in `lNewFlag`, and returns the previous value. An example of how to use this is:

```
  long lPreviousFlags;
  lPreviousFlags = cout.flags();          //record current flag word
```

```
cout.flags(cout.flags() | ios::fixed);      //fixed floating point notation
cout << "7 = " << 7.0 << endl;
cout.flags(lPreviousFlags);                 //reset flags to previous value
cout.flags(cout.flags() | ios::scientific); //scientific notation
cout << "7 = " << 7.0 << endl;
```

We write

```
cout.flags(cout.flags() | ios::fixed);
```

so that we have the union of fixed floating point notation with the flags that have already been set. If we had written

```
cout.flags(ios::fixed);
```

we would have wiped out the all the other format flags that had been set, even the default ones. There are several other functions that you might find more convenient than `flags()`. For example, the `ios::setf(long)` function sets a particular flag, and the `ios::unsetf(long)` function clears that flag. So to set the scientific notation flag, I could write:

```
cout.setf(ios::scientific);
```

To clear it, I could write:

```
cout.unsetf(ios::scientific);
```

Manipulators

Using `ios` member functions breaks up the flow of the output line. It's not very elegant to write

```
cout << "I = ";
cout.width(10);
cout << i << endl;
```

It's prettier to write

```
cout << "I = " << i << endl;
```

To overcome this problem, one can use a manipulator. *Manipulators* are objects defined in the include file `iomanip.h` to have the same effect as the member functions calls. In fact, they call the functions. The only advantage to manipulators is that the program can insert them directly into the stream rather than resort to a separate function call. Some manipulators take arguments and some do not. Technically speaking, you only have to include the header file `iomanip.h` in order to use the manipulators with arguments. Some of the manipulators are listed here. A more complete list can be found on page 100 of *More C++ for Dummies*.

Manipulator	Member Functions	Purpose
endl	ostream::flush()	Insert \n and flush the buffer
flush	ostream::flush()	Flush the stream
setfill()	ios::fill()	Set the fill character
setiosflags()	ios::setf()	Set the flags
resetiosflags()	ios::unsetf()	Undo the flags
setprecision()	ios::precision()	Set the floating-point precision
setw()	ios::width()	Set the width of the next field

Just like the `width()` function, the `setw()` manipulator must be included in the stream for each object whose width is not the default, zero. The only advantage that function calls have over manipulators is that the functions return the previous settings while the manipulators don't return anything. So if you want to store the previous setting, do something, then restore the previous setting, you can easily do it with function calls. With manipulators, you can only undo the flags with `resetiosflags()`. Here is an example using manipulators:

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

void fn()
{
 cout << setw(8) << 10 << setw(8) << 20 << endl;  //keep setting width
 cout << setiosflags(ios::scientific)
      << "7.0 = " << 7.0 << endl;        //scientific notation
 cout << resetiosflags(ios::scientific); //turn off scientific notation
}
```

Custom Inserters

The fact that C++ overloads the left shift operator to perform output means that you can overload the same operator to perform output on classes you define. We have seen examples of this. Recall that the class of complex numbers in lecture 6 had a friend function that overloaded the inserter:

```
ostream & operator << (ostream &, const complex &);
```

Let's give another example with the class `USDollar`:

```
#include <iostream.h>
#include <iomanip.h>
class USDollar
{
  public:
   USDollar(double v = 0.0)
   {
      dollars = v;
      cents = int((v - dollars) * 100.0 + 0.5);
   }
   operator double()
   {
      return dollars + cents / 100.0;
   }
   void display(ostream& out)
   {
      out << '$' << dollars << '.'
          //set fill to 0's for cents
          << setfill('0') << setw(2) << cents
          //now put it back to spaces
          << setfill(' ');
   }
  protected:
   unsigned int dollars;
   unsigned int cents;
};

//operator<< - overload the inserter for our class
ostream& operator<< (ostream& o, const USDollar& d)
{
   d.display(o);
   return o;
}
int main()
{
   USDollar usd(1.50);
   cout << "Initially usd = " << usd << "\n";
   usd = 2.0 * usd;
   cout << "then usd = " << usd << "\n";
   return 0;
}
```

The `display()` function starts by displaying `$`, the dollar amount, and the obligatory decimal point. Notice that output is to whatever `ostream` object it is passed and not necessarily just to `cout`. This allows the same function to be used on objects of `ostream` and its subclasses such as `fstream`. When it comes time to display the cents amount, `display()` sets the width to 2 positions and the leading character to 0. This ensures that numbers smaller than 10 display properly. Notice how the class `USDollar`, instead of accessing the `display()` function directly, also utilizes an `operator<<(ostream&, USDollar&)`. The programmer can now output `USDollar` objects as easily as intrinsic types, as the example in `main()` demonstrates. The output from the program is:

```
Initially usd = $1.50
then usd = $3.00
```

Notice that the `operator<<()` returns the `ostream` passed to it. This allows the operator to be chained with other inserters in a single expression, i.e, this allows us to string output operators together:

```
   complex c, d, x;
   ostream s;
   s << c << d << x;
```

Because the `operator<<()` binds left to right, the expression

```
      s << c << d << x;
```

can be interpreted as

```
      (((s << c) << d) << x);
```

The first insertion outputs the complex number `c` to `s`. The result of this expression is the object `s`, which is then passed to `operator<<(ostream &, const complex &)`. It is important that this operator return its `ostream` object so that the object can be passed to the next inserter in turn.

Smart Inserters

Many times, you would like to make the inserter smart. That is, you would like to say `cout << baseClassObject` and let C++ choose the proper subclass inserter in the same way that it choose the proper virtual member function. Because the inserter is not a member function, you cannot declare it virtual directly. There is a clever way to get around this:

```
#include <iostream.h>
#include <iomanip.h>
class Currency
{
  public:
   Currency(double v = 0.0)
```

```
   {
      unit = v;
      cent = int((v - unit) * 100.0 + 0.5);
   }
   virtual void display(ostream& out) = 0;

  protected:
   unsigned int unit;
   unsigned int cent;
};
class USDollar : public Currency
{
  public:
   USDollar(double v = 0.0) : Currency(v)
   {
   }
   //display $123.00
   virtual void display(ostream& out)
   {
      out << '$' << unit << '.'
          << setfill('0') << setw(2) << cent
          << setfill(' ');
   }
};
class DMark : public Currency
{
  public:
   DMark(double v = 0.0) : Currency(v)
   {
   }
   //display 123.00DM
   virtual void display(ostream& out)
   {
      out << unit << '.'
          //set fill to 0's for cents
          << setfill('0') << setw(2) << cent
          //now put it back to spaces
          << setfill(' ')
          << " DM";
   }
};
ostream& operator<< (ostream& o, Currency& c)
```

```
{
   c.display(o);
   return o;
}
void fn(Currency& c)
{
   // the following output is polymorphic because the
   // operator<<(ostream&, Currency&) is defined through a virtual
   // member function
   cout << "Deposit was " << c
        << "\n";
}
int main()
{
   //create a dollar and output it using the
   //proper format for a dollar
   USDollar usd(1.50);
   fn(usd);

   //now create a DMark and output it using its own format
   DMark d(3.00);
   fn(d);
   return 0;
}
```

The class `Currency` has two subclasses, `USDollar` and `DMark`. In `Currency`, the `display()` function is declared pure virtual. In each of the two subclasses, this function is overloaded with a `display()` function to output the object in the proper format for that type. The call to `display()` in `operator<<()` is now a virtual call. Thus, when `operator<<()` is passed `USDollar`, it outputs the object as a dollar. When passed `DMark`, it outputs the object as a deutsche mark. Thus, although `operator<<()` is not virtual, because it invokes a virtual function, it acts like a virtual function and the result is:

```
Deposit was $1.50
Deposit was 3.00 DM
```

This is another reason why it is better to perform the work of output in a member function, and let the non-member function refer to that function.