SESSION

29

Stream I/O

Session Checklist

- ✔ Rediscovering stream I/O as an overloaded operator
- ✔ Using stream file I/O
- ✔ Using stream buffer I/O
- ✔ Writing your own inserters and extractors
- ✔ Behind the scenes with manipulators

30 Min. To Go

So far, our programs have performed all input from the `cin` input object and output through the `cout` output object. Perhaps you haven't really thought about it much, but this input/output technique is a subset of what is known as stream I/O.

This session explains stream I/O in more detail. I must warn you that stream I/O is too large a topic to be covered completely in a single session — entire books are devoted to this one topic. I can get you started, though, so that you can perform the main operations.

How Does Stream I/O Work?

Stream I/O is based on overloaded versions of `operator>()` and `operator<<()`. The declaration of these overloaded operators is found in the include file iostream.h, which we have included in our programs since Session 2. The code for these functions is included in the standard library, which your C++ program links with.

The following shows just a few of the prototypes appearing in iostream.h:

```
//for input we have:
istream& operator>(istream& source, char *pDest);
istream& operator>(istream& source, int  &dest);
istream& operator>(istream& source, char &dest);
//...and so forth...

//for output we have:
ostream& operator<<(ostream& dest, char *pSource);
ostream& operator<<(ostream& dest, int   source);
ostream& operator<<(ostream& dest, char  source);
//...and so it goes...
```

When overloaded to perform I/O, `operator>()` is called the *extractor* and `operator<<()` is called the *inserter*.

Let's look in detail at what happens when I write the following:

```
#include <iostream.h>
void fn()
{
   cout << "My name is Randy\n";
}
```

The `cout` is an object of class `ostream` (more on this later). Thus, C++ determines that the best match is the `operator<<(ostream&, char*)` function. C++ generates a call to this function, the so-called `char*` inserter, passing the function the `ostream` object `cout` and the string `"My name is Randy\n"` as arguments. That is, it makes the call `operator<<(cout, "My name is Randy\n")`. The `char*` inserter function, which is part of the standard C++ library, performs the requested output.

The `ostream` and `istream` classes form the base of a set of classes that connect the application code with the outside world including input from and output to the file system. How did the compiler know that `cout` is of class `ostream`? This and a few other global objects are also declared in iostream.h. A list is shown in

Table 29-1. These objects are constructed automatically at program startup, before `main()` gets control.

Table 29-1
Standard Stream I/O Objects

Object	Class	Purpose
`cin`	`istream`	Standard input
`cout`	`ostream`	Standard output
`cerr`	`ostream`	Standard error output
`clog`	`ostream`	Standard printer output

Subclasses of `ostream` and `istream` are used for input and output to files and internal buffers.

The fstream Subclasses

The subclasses `ofstream`, `ifstream`, and `fstream` are defined in the include file fstream.h to perform stream input and output to a disk file. These three classes offer a large number of member functions. A complete list is provided with your compiler documentation, but let me get you started.

Class `ofstream`, which is used to perform file output, has several constructors, the most useful of which is the following:

```
ofstream::ofstream(char *pszFileName,
                   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. The legal values for `mode` are listed in Table 29-2 and those for `prot` in Table 29-3. These values are bit fields that are ORed together (the classes `ios` and `filebuff` are both parent classes of `ostream`).

Tip

The expression `ios::out` refers to a static data member of the class `ios`.

Table 29-2
Constants Defined in ios to Control How Files Are Opened

Flag	Meaning
`ios::ate`	Append to the end of the file, if it exists
`ios::in`	Open file for input (implied for `istream`)
`ios::out`	Open file for output (implied for `ostream`)
`ios::trunc`	Truncate file if it exists (default)
`ios::nocreate`	If file doesn't already exist, return error
`ios::noreplace`	If file does exist, return error
`ios::binary`	Open file in binary mode (alternative is text mode)

Table 29-3
Values for prot in the ofstream Constructor

Flag	Meaning
`filebuf::openprot`	Compatibility sharing mode
`filebuf::sh_none`	Exclusive; no sharing
`filebuf::sh_read`	Read sharing allowed
`filebuf::sh_write`	Write sharing allowed

For example, the following program opens the file MYNAME and then writes some important and absolutely true information to that file:

```
#include <fstream.h>
void fn()
{
   //open the text file MYNAME for writing - truncate
   //whatever's there now
   ofstream myn("MYNAME");
   myn << "Randy Davis is suave and handsome\n"
       << "and definitely not balding prematurely\n";
}
```

The constructor `ofstream::ofstream(char*)` expects only a filename and provides defaults for the other file modes. If the file MYNAME already exists, it is truncated; otherwise, MYNAME is created. In addition, the file is opened in compatibility sharing mode.

Referring to Table 29-2, if I wanted to open the file in binary mode and append to the end of the file if the file already exists, I would create the `ostream` object as follows. (In binary mode, newlines are not converted to carriage returns and line feeds on output nor are carriage returns and line feeds converted back to newlines on input.)

```
void fn()
{
   //open the binary file BINFILE for writing; if it
   //exists, append to end of whatever's already there

   ofstream bfile("BINFILE", ios::binary | ios::ate);
   //...continue on as before...
}
```

The stream objects maintain state information about the I/O process. The member function `bad()` returns an error flag which is maintained within the stream classes. This flag is nonzero if the file object has an error.

Note

Stream output predates the exception-based error-handling technique explained in Session 30.

To check whether the MYNAME and BINFILE files were opened properly in the earlier examples, I would have coded the following:

```
#include <fstream.h>
void fn()
{
   ofstream myn("MYNAME");
   if (myn.bad())        //if the open didn't work...
   {
      cerr << "Error opening file MYNAME\n";
      return;            //...output error and quit
   }
```

```
    myn << "Randy Davis is suave and handsome\n"
        << "and definitely not balding prematurely\n";
}
```

All attempts to output to an `ofstream` object that has an error have no effect until the error has been cleared by calling the member function `clear()`.

Tip

This last paragraph is meant quite literally — no output is possible as long as the error flag is nonzero.

The destructor for class `ofstream` automatically closes the file. In the preceding example, the file was closed when the function exited.

Class `ifstream` works much the same way for input, as the following example demonstrates:

```
#include <fstream.h>
void fn()
{
    //open file for reading; don't create the file
    //if it isn't there

    ifstream bankStatement("STATEMNT", ios::nocreate);
    if (bankStatement.bad())
    {
        cerr << "Couldn't find bank statement\n";
        return;
    }
    while (!bankStatement.eof())
    {
        bankStatement > nAccountNumber > amount;
        //...process this withdrawal
    }
}
```

The function opens the file STATEMNT by constructing the object `bankStatement`. If the file does not exist, it is not created. (We assume that the file has information for us, so it wouldn't make much sense to create a new, empty file.) If the object is bad (for example, if the object was not created), the function outputs an error message and exits. Otherwise, the function loops, reading the `nAccountNumber` and withdrawal `amount` until the file is empty (end-of-file is true).

An attempt to read an `ifstream` object that has the error flag set, indicating a previous error, returns immediately without reading anything.

Tip

Let me warn you one more time. Not only is nothing returned from reading an input stream that has an error, but the buffer comes back unchanged. This program can easily come to the false conclusion that it has just read the same value as previously. Further, `eof()` **will never return a true on an input stream which has an error.**

The class `fstream` is like an `ifstream` and an `ofstream` combined (in fact, it inherits from both). An object of class `fstream` can be created for input or output, or both.

20 Min.
To Go

The strstream Subclasses

The classes `istrstream`, `ostrstream`, and `strstream` are defined in the include file strstrea.h. (The file name appears truncated on the PC because MS-DOS allowed no more than 8 characters for a file name; GNU C++ uses the full file name strstream.h.) These classes enable the operations defined for files by the `fstream` classes to be applied to buffers in memory.

For example, the following code snippet parses the data in a character string using stream input:

```
#include <strstrea.h>
//Change to <strstream.h> for GNU C++
char* parseString(char *pszString)
{
   //associate an istrstream object with the input
   //character string
   istrstream inp(pszString, 0);

   //now input from that object
   int nAccountNumber;
   float dBalance;
   inp > nAccountNumber > dBalance;

   //allocate a buffer and associate an
   //ostrstream object with it
```

```
    char* pszBuffer = new char[128];
    ostrstream out(pszBuffer, 128);

    //output to that object
    out << "account number = " << nAccountNumber
        << ", dBalance = $" << dBalance
        << ends;

    return pszBuffer;
}
```

This function appears to be much more complicated than it needs to be, however, `parseString()` is easy to code but very robust. The `parseString()` function can handle any type of messing input that the C++ extractor can handle and it has all of the formatting capability of the C++ inserter. In addition, the function is actually simple once you understand what it's doing.

For example, let's assume that `pszString` pointed to the following string:

```
"1234 100.0"
```

The function `parseString()` associates the object `inp` is with the input string by passing that value to the constructor for `istrstream`. The second argument to the constructor is the length of the string. In this example, the argument is 0, which means "read until you get to the terminating NULL."

The extractor statement `inp >` first extracts the account number, 1234, into the `int` variable `nAccountNumber` exactly as if it were reading from the keyboard or a file. The second half extracts the value 100.0 into the variable `dDBalance`.

On the output side, the object `out` is associated with the 128 character buffer pointed to by `pszBuffer`. Here again, the second argument to the constructor is the length of the buffer — this value cannot be defaulted because `ofstrstream` has no way of determining the size of the buffer (there is no terminating NULL at this point). A third argument, which corresponds to the mode, defaults to `ios::out`. You can set this argument to `ios::ate`, however, if you want the output to append to the end of whatever is already in the buffer rather than overwrite it.

The function then outputs to the out object - this generates the formatted output in the 128 character buffer. Finally, the parseString() function returns the buffer. The locally defined `inp` and `out` objects are destructed when the function returns.

Note: **The constant `ends` tacked on to the end of the inserter command is necessary to add the `null` terminator to the end of the buffer string.**

The buffer returned in the preceding code snippet given the example input contains the string.

```
"account number = 1234, dBalance = $100.00"
```

Comparison of string-handling techniques

The string stream classes represent an extremely powerful concept. This becomes clear in even a simple example. Suppose I have a function whose purpose is to create a descriptive string from a `USDollar` object.

My solution without using `ostrstream` appears in Listing 29-1.

Listing 29-1
Converting USDollar to a String for Output

```
// ToStringWOStream - convert USDollar to a string
//                    displaying the amount

#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <string.h>

// USDollar - represent the greenback
class USDollar
{
  public:
    // construct a dollar object with an initial
    // dollar and cent value
    USDollar(int d = 0, int c = 0);

    // rationalize - normalize the number of nCents by
    //               adding a dollar for every 100 nCents
void rationalize()
```

Continued

Listing 29-1 *Continued*

```
    {
        nDollars += (nCents / 100);
        nCents   %= 100;
    }

// output- return as description of the
    //        current object
    char* output();

  protected:
    int nDollars;
    int nCents;
};

USDollar::USDollar(int d, int c)
{
    // store of the initial values locally
    nDollars = d;
    nCents = c;

    rationalize();
}

// output- return as description of the
//        current object
char* USDollar::output()
{
    // allocate a buffer
    char* pszBuffer = new char[128];

    // convert the nDollar and nCents values
    // into strings
    char cDollarBuffer[128];
    char cCentsBuffer[128];
    ltoa((long)nDollars, cDollarBuffer, 10);
    ltoa((long)nCents,   cCentsBuffer,  10);

    // make sure that the cents uses 2 digits
```

```
if (strlen(cCentsBuffer) != 2)
    {
        char c = cCentsBuffer[0];
        cCentsBuffer[0] = '0';
        cCentsBuffer[1] = c;
cCentsBuffer[2] = '\0';
    }

    // now tack the strings together

    strcpy(pszBuffer, "$");
    strcat(pszBuffer, cDollarBuffer);
    strcat(pszBuffer, ".");
    strcat(pszBuffer, cCentsBuffer);

    return pszBuffer;
}

int main(int nArgc, char* pszArgs[])
{
   USDollar d1(1, 60);
   char* pszD1 = d1.output();
   cout << "Dollar d1 = " << pszD1 << "\n";
   delete pszD1;

   USDollar d2(1, 5);
   char* pszD2 = d2.output();
   cout << "Dollar d2 = " << pszD2 << "\n";
   delete pszD2;

   return 0;
}
```

Output

```
Dollar d1 = $1.60
Dollar d2 = $1.05
```

The `ToStringWOStream` program does not rely on stream routines to generate the text version of a `USDollar` object. The function `USDollar::output()` makes heavy use of the `ltoa()` function, which converts a long into a string, and of the `strcpy()` and `strcat()` functions to perform the direct string manipulation. The function must itself handle the case in which the number of cents is less than 10 and, therefore, occupies only a single digit. The output from this program is shown at the end of the listing.

The following represents a version of `USDollar::output()` that does use the `ostrstream` class.

This version is included in the ToStringWStreams program on the accompanying CD-ROM.

```
char* USDollar::output()
{
    // allocate a buffer
    char* pszBuffer = new char[128];

    // attach an ostream to the buffer
    ostrstream out(pszBuffer, 128);

    // convert into strings (setting the width
    // insures that the number of cents digit is
    // no less than 2
    out << “$” << nDollars << “.”;
    out.fill(‘0’);out.width(2);
    out << nCents << ends;

    return pszBuffer;
}
```

This version associates the output stream object `out` with a locally defined buffer. It then writes the necessary values using the common stream inserter and returns the buffer. Setting the width to 2 insures that the number of cents uses 2 digits when its value is less than 10. The output from this version is identical to the output shown in Listing 29-1. The `out` object is destructed when control exits the `output()` function.

I find the stream version of `output()`much easier to follow and less tedious than the earlier nonstream version.

Manipulators

So far, we have seen how to use stream I/O to output numbers and character strings using default formats. Usually the defaults are fine, but sometimes they don't cut it. True to form, C++ provides two ways to control the format of output.

First, invoking a series of member functions on the stream object can control the format. You saw this in the earlier `display()` member function where `fill('0')` and `width(2)` set the minimum width and left fill character of `ostrstream`.

Note

The argument `out` represents an ostream object. Because `ostream` is a base class for both `ofstream` and `ostrstream`, this function works equally well for output to a file or to a buffer maintained within the program!

A second approach is through *manipulators*. Manipulators are objects defined in the include file iomanip.h to have the same effect as the member function calls. The only advantage to manipulators is that the program can insert them directly in the stream rather than having to resort to a separate function call.

The `display()` function rewritten to use manipulators appears as follows:

```
char* USDollar::output()
{
    // allocate a buffer
    char* pszBuffer = new char[128];

    // attach an ostream to the buffer
    ostrstream out(pszBuffer, 128);

    // convert into strings; this version uses
    // manipulators to set the fill and width
    out << "$" << nDollars << "."
        << setfill('0') << setw(2)
        << nCents << ends;

    return pszBuffer;
}
```

The most common manipulators and their corresponding meanings are listed in Table 29-4.

Table 29-4
Common Manipulators and Stream Format Control Functions

Manipulator	Member function	Description
`dec`	`flags(10)`	Set radix to 10
`hex`	`flags(16)`	Set radix to 16
`Oct`	`flags(8)`	Set radix to 8
`setfill(c)`	`fill(c)`	Set the fill character to `c`
`setprecision(c)`	`precision(c)`	Set display precision to `c`
`setw(n)`	`width(n)`	Set width of field to `n` characters *

Watch out for the width parameter (`width()` function and `setw()` manipulator). Most parameters retain their value until they are specifically reset by a subsequent call, but the width parameter does not. The width parameter is reset to its default value as soon as the next output is performed. For example, you might expect the following to produce two eight-character integers:

```
#include <iostream.h>
#include <iomanip.h>
void fn()
{
   cout << setw(8)       //width is 8...
        << 10            //...for the 10, but...
        << 20            //...default for the 20
        << "\n";
}
```

What you get, however, is an eight-character integer followed by a two-character integer. To get two eight-character output fields, the following is necessary:

```
#include <iostream.h>
#include <iomanip.h>
void fn()
{
   cout << setw(8)       //set the width...
        << 10
        << setw(8)       //...now reset it
<< 20
```

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

Which way is better, manipulators or member function calls? Member functions provide a bit more control because there are more of them. In addition, the member functions always return the previous setting so you know how to restore it (if you want). Finally, each function has a version without any arguments to return the current value, should you want to restore the setting later.

Even with all these features, the manipulators are the more common, probably because they look neat. Use whichever you prefer, but be prepared to see both in other people's code.

10 Min. To Go

Custom Inserters

The fact that C++ overloads the left-shift operator to perform output is neat because you are free to overload the same operator to perform output on classes you define.

Consider the `USDollar` class once again. The following version of the class includes an inserter that generates the same output as the `display()` versions prior:

```
// Inserter - provide an inserter for USDollar

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

// USDollar - represent the greenback
class USDollar
{
    friend ostream& operator<<(ostream& out, USDollar& d);
  public:
    // ...no change...
};

// inserter - output a string description
//            (this version handles the case of cents
//            less than 10)
ostream& operator<<(ostream& out, USDollar& d)
{
```

```
    char old = out.fill();
    out << "$"
        << d.nDollars
        << "."
        << setfill('0') << setw(2)
        << d.nCents;

    // replace the old fill character
    out.fill(old);

    return out;
}

int main(int nArgc, char* pszArgs[])
{
   USDollar d1(1, 60);
   cout << "Dollar d1 = " << d1 << "\n";

   USDollar d2(1, 5);
   cout << "Dollar d2 = " << d2 << "\n";

   return 0;
}
```

The inserter performs the same basic operations as the earlier `display()` functions outputting this time directly to the `ostream` out object passed to it. However, the `main()` function is even more straightforward than the earlier versions. This time the `USDollar` object can be inserted directly into the output stream.

You may wonder why the `operator<<()` returns the `ostream` object passed to it. This is what enables the insertion operations to be chained. Because `operator<<()` binds from left to right, the following expression

```
USDollar d1(1, 60);
cout << "Dollar d1 = " << d1 << "\n";
```

is interpreted as

```
USDollar d1(1, 60);
((cout << "Dollar d1 = ") << d1) << "\n";
```

The first insertion outputs the string "Dollar d1 = " to `cout`. The result of this expression is the object `cout`, which is then passed to `operator<<(ostream&,`

USDollar&). It is important that this operator return its ostream object so that the object can be passed to the next inserter, which outputs the newline character "\n".

Smart Inserters

We often would like to make the inserter smart. That is, we would like to say cout << baseClassObject and let C++ choose the proper subclass inserter in the same way that it chooses the proper virtual member function. Because the inserter is not a member function, we cannot declare it virtual directly.

We can easily sidestep the problem by making the inserter depend on a virtual display() member function as demonstrated by the VirtualInserter program in Listing 29-2.

Listing 29-2
VirtualInserter Program

```
// VirtualInserter - base USDollar on the base class
//                   Currency. Make the inserter virtual
//                   by having it rely on a virtual
//                   display() routine

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

// Currency - represent any currency
class Currency
{
    friend ostream& operator<<(ostream& out, Currency& d);
  public:
    Currency(int p = 0, int s = 0)
    {
        nPrimary = p;
        nSecondary = s;
    }

    // rationalize - normalize the number of nCents by
    //               adding a dollar for every 100 nCents
```

Continued

Listing 29-2 *Continued*

```
    void rationalize()
    {
        nPrimary   += (nSecondary / 100);
        nSecondary %= 100;
    }

    // display - display the object to the
    //           given ostream object
    virtual ostream& display(ostream&) = 0;

  protected:
    int nPrimary;
    int nSecondary;
};

// inserter - output a string description
//            (this version handles the case of cents
//            less than 10)
ostream& operator<<(ostream& out, Currency& c)
{
    return c.display(out);
}

// define dollar to be a subclass of currency
class USDollar : public Currency
{
  public:
    USDollar(int d, int c) : Currency(d, c)
    {
    }

    // supply the display routine
    virtual ostream& display(ostream& out)
    {
        char old = out.fill();
        out << "$"
            << nPrimary
            << "."
```

```
                << setfill('0') << setw(2)
                << nSecondary;

            // replace the old fill character
            out.fill(old);

            return out;
        }
};

void fn(Currency& c, char* pszDescriptor)
{
    cout << pszDescriptor << c << "\n";
}

int main(int nArgc, char* pszArgs[])
{
   // invoke USDollar::display() directly
   USDollar d1(1, 60);
   cout << "Dollar d1 = " << d1 << "\n";

   // invoke the same function virtually
   // through the fn() function
   USDollar d2(1, 5);
   fn(d2, "Dollare d2 = ");

   return 0;
}
```

The class `Currency` defines a nonmember, and therefore nonpolymorphic, inserter function. However, rather than perform any real work, this inserter relies on a virtual member function `display()` to perform all the real work. The subclass `USDollar` need only provide the `display()` function to complete the task. This version of the program produces the same output shown at the bottom of Listing 29-1.

That the insertion operation is, indeed, polymorphic is demonstrated by creating the output function `fn(Currency&, char*)`. The `fn()` function does not know what type of currency it is receiving and, yet, displays the currency passed it using the rules for a `USDollar`. `main()` outputs `d1` directly and `d2` through this added function `fn()`. The virtual output from `fn()` appears the same as its nonpolymorphic brethren.

Other subclasses of `Currency`, such as `DMMark` or `FFranc`, can be created even though they have different display rules by simply providing the corresponding `display()` function. The base code could continue to use `Currency` with impunity.

But Why the Shift Operators?

You might ask, “Why use the shift operators for stream I/O? Why not use another operator?”

The left-shift operator was chosen for several reasons. First, it’s a binary operator. This means the `ostream` object can be made the left-hand argument and the output object the right-hand argument. Second, left shift is a very low priority operator. Thus, expressions such as the following work as expected because addition is performed before insertion:

```
cout << “a + b” << a + b << “\n”;
```

Third, the left-shift operator binds from left to right. This is what enables us to string output statements together. For example, the previous function is interpreted as follows:

```
#include <iostream.h>
void fn(int a, int b)
{
   ((cout << “a + b”) << a + b) << “\n”;
}
```

Done!

But having said all this, the real reason is probably just that it looks really neat. The double less than, <<, looks like something moving out of the code, and the double greater than, >, looks like something coming in. And, hey, why not?

REVIEW

I began this session warning you that stream I/O is too complex to cover in a single chapter of any book, but this introduction should get you started. You can refer to your compiler documentation for a complete listing of the various member functions you can call. In addition, the relevant include files, such as iostream.h and iomanip.h, contain prototypes with explanatory comments for all the functions.

- Stream output is based on the classes `istream` and `ostream`.
- The include file iostream.h overloads the left-shift operator to perform output to `ostream` and overloads the right-shift operator to perform input from `istream`.
- The `fstream` subclass is used to perform file I/O.
- The `strstream` subclass performs I/O to internal memory buffers using the same insertion and extraction operators.
- The programmer may overload the insertion and extraction operators for the programmer's own classes. These operators can be made polymorphic through the use of intermediate functions.
- The manipulator objects defined in iomanip.h may be used to invoke `stream` format functions.

QUIZ YOURSELF

1. What are the two operators << and > called when used for stream I/O? (See "How Does String I/O Work?")
2. What is the base class of the two default I/O objects `cout` and `cin`? (See "How Does String I/O Work?")
3. What is the class `fstream` used for? (See "The fstream Subclasses.")
4. What is the class `strstream` used for? (See "The strstream Subclasses.")
5. What manipulator sets numerical output into hexadecimal mode? What is the corresponding member function? (See "Manipulators.")