Lecture 10: Advanced Input and Output Considerations

This lecture discusses advanced input/output (I/O) and file processing issues in C++. You will learn about the C++ I/O streams, about formatting I/O, about reading/writing text files as well as binary files, and other related topics. The lecture proceeds under the following captions:

- Introduction
- C++ Streams
- Overloading Input and Output Operators
- Formatting Input and Output
- Introduction to File Management
- Opening and Closing a File
- Processing Text Files
- Processing Binary Files
- Check for End-of-File
- Additional Formats for Binary I/O
- Random Access Files
- Renaming and Removing Files
- Summary and Concluding Remarks

Copyright © 2000 – 2017 by Elvis C. Foster

All rights reserved. No part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, or otherwise, without prior written permission of the author.

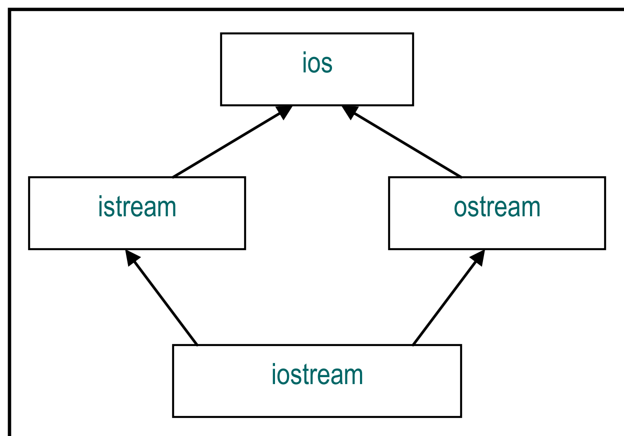
10.1 Introduction

From your introduction to C++ in lecture 1 (where **cin**, **cout**, **getchar**, **gets**, **scanf**, **printf** were discussed); you have been using the C++ I/O system. However, since up until lecture 6, you were not familiar with classes — an essential requirement for fully understanding the C++ I/O system — you were spared a formal discussion. We now embark on that discussion.

As you are aware, there are two I/O systems provided by the C++ programming environment, represented in the two header files **<stdio.h>** and **<iostream.h**> the older system includes the **printf** and **scanf** functions, and other functions. The newer system is an enhancement of the older one, and contains the **cin** and **cout** functions. It also has overloaded operators << and >> (shift left and shift right). Our focus here is on the new system. The older system is a subset of the newer, so that programs that were written for the old will also compile and run in the new.

When you include the header file **<iostream.h>** in your program, you are actually including a file that has the **iostream** class. This is in fact a derived class in (the simplified version of) the class hierarchy illustrated below.

Figure 10-1: Basic C++ I/O System



Each class in the I/O system has an associated *template class* (**basic_ios**, **basic_istream**, **basic_ostream** and **basic_iostream**).

The **istream** class handles inputs and includes a definition of the extraction operator, >>. The **ostream** class handles outputs and includes a definition of the insertion operator, <<. The **iostream** inherits from both **istream** and **ostream**, both of which inherit from **ios**.

The upcoming sections discuss the C++ I/O streams in more detail, how to overload the << and >> operations, how to format input/output.

10.2 C++ Streams

The C++ I/O system operates on *streams*. A *stream* is a common logical interface to the various physical devices of a computer system. The physical devices may be the keyboard (input), the monitor (output), a printer, or a disk file, a port, or a tape file, etc. Of course, not all devices will support all operations; for instance random access does not apply to a printer file.

All streams are treated the same; a C++ file may be implemented as any of the above mentioned devices, but irrespective of the device, it is treated the same way. A stream is linked to a file through an **open** operation, and disengaged from the file via the **close** operation.

There are two types of streams: text and binary.

- A text stream relates to the use of characters only. In some cases, there is not a one-to-one correspondence between what is sent to the stream and what is written to the file. For instance the **newline** character (for output) may mean carriage-return or line-feed, depending on the output device.
- A binary stream can apply to any type of data. There is a one-to-one correspondence between what is sent and what is written to a file.

A stream is really an object instance of one of the above mentioned classes. It is typically activated to manage access of a file. The *current location* (position) in a file is the point where the next file access will occur.

The following C++ predefined streams automatically open when your C++ program begins execution: **cin**, **cout**, **cerr**, **and clog**.

- You are already familiar with **cin** and **cout**, the streams for standard input and standard output, respectively.
- The **clog** and **cerr** streams are also linked to the standard output. The former is buffered and the latter is not. This means that any output to **cerr** is immediately written, while output to **clog** is written only when the buffer is full. Typically, program debugging and error information are written to **clog** and **cerr**.
- The 16-bit-wide versions of these predefined streams are **wcin, wcout, wcerr**, and **wclog**. These streams support languages such as Chinese, which require large character sets.
- By default, the standard streams are linked to the console, but they can be redirected to other devices, files, or the operating system.

10.3 Overloading Input and Output Operators

In our discussion of classes so far, you have been familiarized with the techniques of defining a member function to take care of inputting data associated with a class instance, and a member function to take care of outputting data from an instance. As an alternative to this, C++ allows you to overload the << and >> I/O operators.

The << operator referred to as the *insertion* operator (because it inserts characters into a stream); the >> operator is referred to as the *extraction* operator (because it extracts characters from a stream). As you know, << works with output, while >> works with input. The operator functions are referred to as *inserter* and *extractor*, respectively.

Example 10-1: Figure 10-2 illustrates how the << and >> operators may be overloaded for CollegeMember class of earlier discussions.

Figure 10-2: Example of Overloading the << and >> Operators

```
class CollegeMember
{
     protected:
     int cmID_Number;
     string cmFirstName, cmLastName, cmAddressLine1, cmAddressLine2, cmStateProv, cmZip, cmTelephone, cmE_Mail;

     // member functions
     public:
     CollegeMember ( ); // constructor
     void modify (CollegeMember thisMember);
     void inputData(int x);
     void printMe ( );
};
// …
class Student: protected CollegeMember
{
     protected:
     string sAcadDept, sMajor;
     //Member Functions
     public:
     Student ( );       // constructor
     void modify (Student thisStud);
     void inputData(int x);
     void printMe ( );
     friend ostream &operator << (ostream &Stream, Student thisStud);   // Inserter
     friend istream &operator >> (istream &Stream, Student thisStud); // Extractor
};
// … Member functions not shown here; as in figures 9-3 and 9-7
int main(int argc, char *argv[])
{
 Student sBerns, sLeidy, sPierre;
 cin >> sBerns >> sLeidy >> sPierre;
 // …
 cout <<s Berns << sLeidy;
 // …
}
```

Figure 10-2: Example of Overloading the << and >> Operators (continued)

```
ostream &operator << (ostream &Stream, Student thisStud) // Inserter
{    // Inserter works typically with cout
 string  outString = "Student information follows\n" + thisStud.cmID_Number + " -- " + thisStud.cmFirstName + " " +
     thisStud.cmLastName + "\n";
 outString += thisStud.cmAddressLine1 + "\n" + thisStud.cmAddressLine2 + ","  + thisStud.cmStateProv + " " + thisStud.cmZip;
 outString += "Telephone " + thisStud.cmTelephone + "\n" + "E-mail " + thisStud.cmE_Mail + "\n" +
 "Academic Department: " + sAcadDept + "Major: " + sMajor + "\n";
 Stream << outString << endl;
 return Stream;
}

istream &operator >> (istream &Stream, Student thisStud) // Extractor
{    // Extractor works typically with cin
  cout << "Enter the Student Number:  ";
  Stream >> thisStud.cmID_Number; cout  << endl;
  //  … Similarly for the other elements
  cout << " Enter Major: ";
  Stream >> thisStud.sMajor; cout << endl;
  return Stream;
}
```

Here are a few points of clarification:

1. The overloaded operator << function returns a reference to an object of type **ostream**.

2. The stream is not hard coded in the function, since this would limit its flexibility. The reference to an **ostream** (or **istream**) object ensures that whatever stream is used on the call of the operator function will be used and returned.

3. Overloaded inserters and extractors cannot be member functions of a class, but they can be friend functions. The reason for this is that for a member operator function, the left operand (passed implicitly) must be an object of the class. However, in an inserter or extractor, the left operand is the stream, and the right operand is an object of the class being used for output or input.

4. In summary, when overloading << or >>, the general format of the (preferably friend) function must be as follows:

```
ostream &operator<< (ostream &Stream, ObjectClass &Obj)
{
    // ….
    return Stream;
}
```

```
istream &operator >> (istream &Stream, ObjectClass &Obj)
{
    // …
     return Stream;
}
```

10.4 Formatting Input and Output

In lecture 1, we discussed formatting outputs in a C/C++ program. There are four additional ways to manipulate input/output in your C++ program:

- Use of member functions of the **ios** class
- Use of **ostream** member functions
- Use of **istream** member functions
- Use of manipulator functions

10.4.1 Using the ios Member Functions

Each stream has an associated set of format flags to control the way information is formatted by the stream. The **ios** class declares a bit-mask enumeration called **fmtflags**, which defines the values shown in figure 10-3. These values are used to set or clear the format flags.

Figure 10-3: Format Values

skipws:	Skip white space characters (newline, space, tabs, carriage return). When cleared, white space characters are not ignored.
left:	Output is left justified when set.
right:	Right justify output (the default).
internal:	Pad numeric values to fill a field by inserting spaces.
oct:	Display output in octal.
hex:	Display output in hexadecimal.
dec:	Display output in decimal.
showbase:	Show the base of numeric values — e.g. 0x1F for hex IF
uppercase:	Display letters in uppercase.
showpos:	Display leading + in front of positive values
showpoint:	Display decimal point and trailing zeroes for floating point values.
scientific:	Display floating values in scientific notation.
fixed:	Display floating values using normal notation (the default).
unitbuf:	Flush buffer after each insertion.
boolalpha:	Allow boolean values (true or false) to be input.
basefield:	Refers to oct, dec, and hex collectively.
adjustfeld:	Refers to left, right, and internal collectively.
floatfield:	Refers to scientific and fixed collectively.

10.4.1 Using the ios Member Functions (continued)

Use the **setf(…)** function (a member of **ios**) to set a flag. It's most common form of usage is

```
<Stream>.setf (<FormatFlag>)
```

Here, **Stream** is a valid stream (**cout, cin, cerr**, or **clog**) and **Formatflag** is one of the format flags of figure 10-3. The format flag is specified as **ios :: flag** (i.e. via the scope resolution operator).

Example 10-2: The following two statements illustrate how the **setf(…)** function is used.

```
cout.setf (ios :: scientific);     // Floating point values will be in scientific notation
// … // Statements which output floating point values
cout.setf (ios :: fixed);          // Floating point values will be in normal notation
// … // Statements which output floating point values
```

You can combine format flags by using the bit-wise OR operator.

Example 10-3: The following statement illustrates how the bit-wise OR operator is used.

```
cout.setf (ios ::showpos | ios :: dec | ios :: showpoint);

// Produces output that is signed, in decimal and with decimal point
// e.g. cout << 4; will output +4.000000 (default 6 decimal places)
```

To turn off a flag, use the **unset(…)** member function. Similar to the **setf** function, its common form of usage is as follows:

```
Stream.unsetf (<FormatFlag>);
```

Here, **Stream** is a valid stream and **FormatFlag** is specified via the scope resolution operator.

Example 10-4: Below is an example using the **unset(…)** function.

```
cout.unsetf (ios :: scientific); // unsets the scientific notation flag
```

10.4.2 Using ostream Member Functions

In addition to manipulating the format flags via the **setf** and **unsetf** member functions, you can determine the field width, fill character and number of decimal places by using the member functions **width**, **fill**, and **precision**, from the **ostream** class. Figure 10-4 shows the syntax for how each function is typically called.

Figure 10-4: Syntax for Calling the width, fill, and precision Functions

```
cout.width (<IntegerExpression>)
cout.precision (<IntegerExpression>)
cout.fill (<Character>);
```

Following are three points of clarification:

1. In some implementations, the **width(…)** function applies only to the output that immediately follows.

2. The **precision(…)** function applies until it is changed by another precision setting. By default, there are 6 digits of precision.

3. The default fill character is a space. When the **fill(…)** function is called, the fill character is changed to the value specified. This holds until changes by another call of **fill**.

Example 10-5: The following code snippet illustrates usage of the **fill**, **width**, and **precision** functions.

Statements	Results
cout.setf (ios :: showpos);	
cout.setf (ios :: scientific);	
cout << 426 << " " << 426.43	+426 4.264300e +002
cout.unsetf (ios :: scientific);	
cout.unsetf (ios :: showpos);	
…	
cout.precision(4);	
cout.width(10);	
cout << 47 << " "	
cout.width (10);	
cout << 47.34	47 47.3400
…	
cout.fill ('#');	
cout.width (10);	
cout<< 47	###47.0000

10.4.3 Using istream Member Functions

One important member function of **istream** is **get.** Most compilers implement **get** as an overloaded function with several formats as explained by the following prototypes:

Format 1:

```
int get ( );
```

In this form, the function is similar to the **getchar** function (of lecture 1); it may be used when the key pressed is not important to the program.

Example 10-6: Illustrating the most basic use of **get()**

```
// Conduct a dummy read
cout<<  "Press any key to continue"; cin.get ( );
```

Format 2:

```
istream &get (char ch);
```

In this format, the function reads a character into **ch**, then returns a reference to the object (stream that invoked it.

Format 3:

```
istream &get (char* inString, int inLength, char inTermination = '\n')
```

In this format, the function reads a string of specified length into **inString**, until a termination character is read. If no termination character is specified on the call, the default is the new-line character. It returns a reference to the object (stream) that invoked it.

Example 10-7: Illustrating use of **get(…)** to read a string in.

```
// Read input string of a specific length, or until the termination character is encountered
string inName;
cout << "Enter your name:" cin.get (Name, 25); // or simply cin.get (Name);
```

One potential problem with **get** is that it leaves '\n' in the input stream. To avoid this bother, the **getline** function, which except for its name, has the same format, is preferred. Importantly, getline discards the new-line character from the input stream (after reading it), so subsequent reads do not see it.

Example 10-8: Illustrating use of **getline(…)** to read a string in.

```
string inName;
cout << "Enter your name: "
cin.getline (inName, 25);      // preferred to cin.get (inName, 25)
```

10.4.4 Using I/O Manipulation Functions

Another way to alter the format of input and output is via special functions, called *manipulators.* These functions can be included in an I/O expression. To use manipulators (see figure 10-5), you must include the header file **<iomanip.h>**.

Figure 10-5: I/O Manipulator Functions

Manipulator	Purpose	Input/Output
boolalpha	Turns on **boolalpha** flag	Input/Output
dec	Turns on **dec** flag	Input/Output
endl	Output a newline character and flush the stream	Output
ends	Output a null	Output
fixed	Turns on **fixed** flag	Output
flush	Flush a stream	Output
hex	Turns on **hex** flag	Input/Output
internal	Turns on **internal** flag	Output
left	Turns on **left** flag	Output
nobooalpha	Turns off **boolalpha** flag	Input/Output
noshowbase	Turns off **showbase** flag	Output
noshowpoint	Turns off **showpoint** flag	Output
noshowpos	Turns off **showpos** flag	Output
noskipws	Turns off **skipws** flag	Input
nounitbuf	Turns off **unitbuf** flag	Output
nouppercase	Turns off **uppercase** flag	Output
oct	Turns on **oct** flag	Input/Output
resertiosflags (fmtflags f)	Turns off the flags specified in *f*	Input/Output
right	Turns on **right** flag	Output
scientific	Turns on **scientific** flag	Output
setbase (int base)	Set the number base to *base*	Input/Output
setfill (int ch)	Set the fill character to *ch*	Output
setiosflags (fmtflgs f)	Turn on the flags specified in *f*	Input/Output
setprecision (int p)	Set the number of digits of precision	Output
setw (int w)	Set the field width to *w*	Output
showbase	Turns on **showbase** flag	Output
showpoint	Turns on **showpoint** flag	Output
showpos	Turns on **showpos** flag	Output
skipws	Turns on **skipws** flag	Input
unitbuf	Turns on **unitbuf** flag	Output
uppercase	Turns on **uppercase** flag	Output
ws	Skip leading white space	Input

10.4.4 Using I/O Manipulation Functions (continued)

Example 10-9: The following code snippet shows how selected manipulation functions may be used.

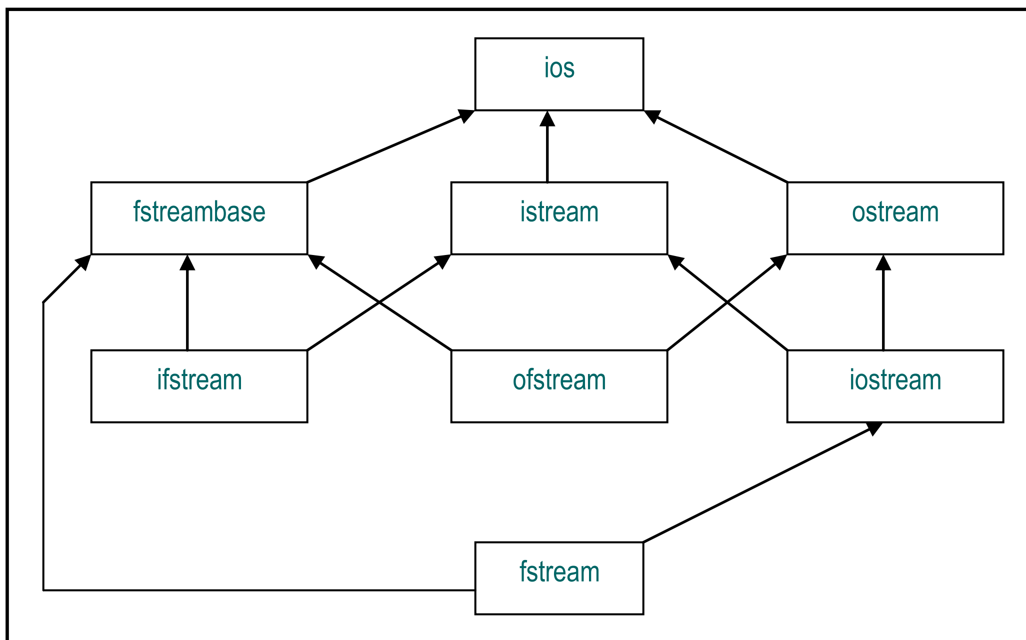
```
double aSalary;
// . . .
cout << setprecision (2) << setw (9) << Salary; // prints Salary of width 9, with 2 decimal places
```

10.5 Introduction to File Management

One of the most important aspects of programming is being able to manipulate files. After all is said and done, your program is in many instances, of little use if you cannot write data to, and read data from files.

Similar to **istream, ostream** and **iostream**, the C++ environment provides other classes which are essential for file processing. These include **ifstream**, **ofstream** and **fstream**. Figure 10-6 illustrates a more detailed version of the C++ I/O system (compared to the representation of figure 10-1).

Figure 10-6: C++ I/O System



If your program is to read from, or write to a disk file, it must include the header file **<fstream.h>**. Notice that **fstream** inherits from iostream; the format features discussed earlier can therefore be applied to file processing.

10.5 Introduction to File Management (continued)

File management involves four basic steps:
- Declaring the file
- Opening the file
- Processing the file
- Closing the file

10.6 Opening and Closing a File

There are two ways to declare and open a file:
- Through a declaration and an explicit open.
- Through a declaration and implicit open.

10.6.1 File Opening via Instantiation and Explicit Open

Because the I/O system classes have overloaded constructors, it is not necessary to supply an argument when a file object is created (instantiated). If none is supplied, then you must use the member function **open**, to open the file and establish a link to an external object. The **open** function requires an argument — the external name of the file (stored on disk). The prototype for each **open** member function is shown in figure 10-7:

Figure 10-7: Function Prototypes for the open(. . .) Function

```
void ifstream :: open (const char* fileName, openmode fileMode = ios :: in);
void ofstream :: open (const char* fileName, openmode fileMode = ios :: trunc | ios :: out);
void fstream :: open (const char* fileName, openmode fileMode);
```

Mode	Meaning
app	Output is appended to the file; file can be used for I/O.
ate	Set the file pointer at the end of the file upon opening.
in	File capable of input only.
out	File capable of output only.
binary	File opened as binary (the default is text); no character translations occur.
trunc	Causes the contents of any preexisting file of the same name to be destroyed, and the file truncated to zero length. This is the default for output files.

Note the following related guidelines:

1. In each of the above prototypes, **fileName** is the external name of the file and can include a path specification. It must be a string (literal or string variable).

2. The **fileMode** determines how the file is to be used. It must be one of the values specified by the enumeration **openmode,** specified in **ios** with values. Two or more of these values can be specified by the bit-wise OR operator.

10.6.1 File Opening via Instantiation and Explicit Open (continued)

3. Unless you are unhappy with the default values (see prototypes above) in some, instances you don't have to specify a mode. However, in many other cases requiring output, it is advisable to change the mode to **app** combined with **out**.

Example 10-10: Illustrating file opening:

```
ifstream Input; // declares an input file or stream
ofstream Output; // declares an output file or stream
fstream Both; // declares an I/O file or stream
Input.open ("InFile.txt");  // opens the file & attaches it to its external object name
// …
Output.open ("OutFile.txt", ios :: app | ios :: out);
// …
```

4. There are a number of ways to check the success of a file open:
   - If **open** fails, the stream evaluates to false. This could be checked.
   - The Boolean member function **is_open** of class **fstreambase** could also be used (review figure 10-6).

Example 10-11: The following code snippet illustrates how checking for file open can be done.

```
if (!Input) cout << "File open failed";  //   File open failed
else…. // file access instructions could follow

//   OR
// …
if (! Input.is_open ( )) cout  << "File not opened";
else… // file access instructions …
```

10.6.2 File Opening via Instantiation and Implicit Open

You may instantiate a file object as you do any other object, by calling its constructor. In calling the constructor, you supply a file name as the argument, and optionally a mode. The constructor will then open the file. Figure 10-8 shows the syntax required for doing so.

Figure 10-8: Syntax for Implicit File Open

```
ImplicitFileOpen ::=
<streamClass> <fileObject> (<ExternalObject>, [<fileMode>] );
```

10.6.2 File Opening via Instantiation and Implicit Open (continued)

Example 10-12: Figure 10-9 illustrates instantiation and implicit file open.

Figure 10-9: Illustrating Instantiation and Implicit File Open

```
ofstream MyFile ("MyFile.txt");
// . . .
/* Instantiates an ofstream object called MyFile. The external filename MyFile.Txt is passed to the constructor, which
opens the file. If the file is absent, it is created; If the file is present, it is overwritten. */

/* The default mode is ios :: out, but this can be changed by specifying a mode after the external filename (separate the
two by a comma). */

ifstream MyFile ("MyFile.txt");

/* Instantiates an ifstream object called MyFile. The external filename MyFile.txt is opened for input.
The default mode is ios :: in but this can be changed by including a new mode as an additional argument */
```

10.6.3 Closing a File

To close a file, use the **close** member function.

Example 10-13: Referring to the files of example 10-10, we could have the following statements:

```
Input.close ( ); Output.close ( ); Both.close ( );
```

10.7 Processing Text Files

The easiest way to read from or write to a text file is to use the >> and << operators. You can also use the **getline** and **get** functions for reading (review section 10.4)

Example 10-14: Figure 10-10 illustrates a simple program that writes a text file.

Figure 10-10: Illustrating Writing to a Text File

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

int main(int argc, char *argv[])
{
    ofstream outFile ("DearMom.txt");
    if (outFile)      //   File open successful
    {
        outFile << "Dear Mother, \n";
        outFile << "I love you immensely \n";
        outFile.close ( );
    }
    else cout << "File not open";
}
```

Example 10-15: Figure 10-11 illustrates a simple program that reads a text file.

Figure 10-11 Illustrating Reading a Text File

```
#include <fstream.h>
#include <iostream.h>
int main(int argc, char *argv[])
{
  char* thisLine;
  ifstream inFile ("DearMom.txt");
  while (inFile) // While not end of file
     if (inFile.getline (thisLine, 80)) cout << thisLine << endl; // if successful read, output
  InFile.close ( );
}
```

Note the following:

1. When >> is used for reading text files, character translations occur, for example white space characters are omitted. If this is to be prevented, the file must be opened for binary access.

2. Remember when >> is used to read a string, input stops when the first white space character is encountered.

10.8 Processing Binary Files

As an alternative to text file, you could work with binary files. With binary files, you have more flexibility as to what you can do. But you must remember that no translation of white space characters occurs.

There are two ways to read and write unformatted binary data from or to a file:

- Via the **get, getline** and **put** member functions.
- Via the block I/O functions **read** and **write** functions.

10.8.1 Using get, getline and put Member Functions

In the interest of clarity, the prototypes of the commonly used versions of the **get, getline** and **put** member functions are repeated here in figure 10-12.

Figure 10-12: Prototypes for Functions get(…), getline(…), and put(…)

```
istream &get (char &ch);
istream &get (char* inString, int inLength, char inTermination = '\n');
istream &getline (char* inString, int inLength, char inTermination = '\n');
ostream &put (char ch);
```

The **get(…)** function reads a character (or string of a specified length, until a termination character is reached) from the associated stream and puts that value in the specified target variable (**ch** or **inString**); the function then returns a reference to the object that invoked it. This reference will be null if end of file is reached. The **getline(…)** function is similar to **get(…),** except that **get** leaves the newline character in the input stream while **getline** discards it after reading it. The **put(…)** function writes a character (**ch**) to the stream and returns a reference to the stream.

Example 10-16: Figure 10-13 illustrates a program reads a binary file, character by character, and outputs to the screen.

Figure 10-13: Illustrating Reading a Binary File

```
#include <fstream.h>
#include <iostream.h>
int main(int argc, char *argv[])
{
  char ch;
  ifstream inFile ("DearMom", ios :: in | ios :: binary);
  while (inFile)  { inFile.get (ch); if (inFile) cout << ch;  }
  inFile.close ( );
// ..
}

/* The above loop can be refined to
while (inFile.get (ch) ) cout << ch; */
```

10.8.1 Using get, getline and put Member Functions (continued)

Example 10-17: Figure 10-14 illustrates a program that reads from a string and writes to a binary file.

Figure 10-14: Illustrating Writing to a Binary File

```
#include <fstream.h>
#include <iostream.h>
int main(int argc, char *argv[])
{
  char* p = “Dear Mother: I love you immensely”;
  ofstream outFile (“DearMom”, ios :: out | ios :: binary);
  if (outFile)
  {
    while (*p) outFile.put (*p++);   outFile.close ( );
  }
  // ..
}
```

10.8.2 Using read and write Member Functions

The **read(…)** and **write(…)** member functions are fortified for reading and writing blocks of binary data. The prototypes are shown in figure 10-15.

Figure 10-15: Prototypes for the read(…) and write(…) Functions

```
istream& read (char* sBuf, streamsize sNumber);
ostream& write (char* sBuf, streamsize sNumber);
```

Following are a few clarifying guidelines:

1. The **read** function reads the **sNumber** bytes (the object’s size) from the associated stream and puts them in the buffer pointed to by **sBuf**. It returns a reference to the object that invoked it.

2. The **write** function writes **sNumber** bytes (the object’s size) to the associated stream from the buffer pointed to by **sBuf**. It also returns a reference to the object that invoked it.

3. The data type **streamsize** is some form of signed integer, defined by the **ios** class. It is capable of holding the largest number of bytes that can be transferred in an I/O operation.

4. When you are writing or reading an object that is not a null-terminated string, you must perform a cast on the address of the object to convert it to a string, as required by the **read** or **write** function. Secondly, to determine the size of the object being read or written, use the **sizeof** operator, which returns the size in bytes of the object received as argument.

10.8.2 Using read and write Member Functions (continued)

Given the above, the general format of a call to the write and read functions is as follows:

```
<outputFile>.write ((char*) (&<Object>), sizeof (<Object>));
<inputFile>.read ((char*) (&<Object>), sizeof (<Object>));
```

Example 10-18: Figure 10-16 illustrates writing objects to a binary file.

Figure 10-16 Illustrating Writing Objects to a Binary File

```
#include <fstream.h>
#include <iostream.h>
class StudentC;
{
    // ... // as defined in the sample program of lecture 6, 8, or 9
};
// . . .
const int sLimit = 100; // Assume 100 student objects
ofstream studFile;
int main(int argc, char *argv[])
{
    StudentC studList [sLimit];
    // ...  Assume that you want to update your file from the array
     UpdateFile (studList);        // function call
     // ...
    studFile.close ( );
}
void updateFile (StudentC studList [sLimit])
{
    studFile.open ("StudFile.dat", ios :: out | ios :: binary | ios :: app);
    for (int x = 0; x < sLimit; x++) studFile.write ((char*) (&studList[x]), sizeof (studList[x]));
}

/* Note: you could actually write an entire array of objects to a file in one stroke thus:
    studFile.write ((char*) (&studList), sizeof (studList)) */
```

10.8.2 Using read and write Member Functions (continued)

Example 10-19: Figure 10-17 illustrates reading objects from a binary file

Figure 10-17: Illustrating Reading Objects from a Binary File

```
#include <fstream.h>
#include <iostream.h>
class StudentC
{
    // ... // as defined in the sample program of lecture 6, 8, or 9
};
// . . .
const int sLimit;
StudentC studList [sLimit];
ifstream studFile;
// . . .
int main(int argc, char *argv[])
{
    // . . . // suppose you want to load your array from the file.
    loadFile ( );
    // . . .
}

void loadFile ( )
{
    studFile.open ("StudFile.dat", ios :: in | ios :: binary);
    if (studFile)
    {
        // Determine required list size and read file in
        StudentC studObject = StudentC();
        sLimit = sizeof(studFile) / sizeof(studObject);
        for (int x = 0; (x < sLimit &&studFile); x++) studFile.read ((char*) (&studList[x]), sizeof (studList[x]));
    }
    studFile.close ( );
}

/* Note: You could actually load the entire array in one statement, thus
    if (studFile) studFile.read ((char*) (&studList), sizeof (studFile)); */

// See figure 10-20 for more insight
```

10.9 Checking for End of File

There are several ways to check for end of file. Three approaches are mentioned here. Suppose that **inFile** is an instance of **istream**. Then you can check for end of file in any of the following ways:

1. Immediately after file open, test **inFile**

```
if (InFile)    // if not end of file
```

as in the examples of the previous sections.

2. Immediately after an attempted file access, check **inFile** as in 1.

3. Use the boolean member function **eof** after an attempted read or open:

```
if (!inFile.eof ( ))      // if not EOF
```

10.10 Additional Functions for Binary I/O

In addition to the functions discussed, the following member functions also can be applied to binary I/O processing:

Getline(. . .): This function can also be used with binary files as used in figure 10-11 of earlier discussion.

peek(): This function determines the next character in the input stream. It returns the next character in the stream or EOF, if end of file is encountered. It has the prototype:

```
int peek ( );
```

flush(): This function forces write to disk, even if the internal buffer is not full. (Under normal circumstances, data written is not immediately transferred to disk storage. Rather, it is done when the internal buffer is full, this transparent to the user). The **flush** function has the prototype.

```
ostream& flush ( );
```

Example 10-20: The following sample statements illustrate use of **peek** and **flush** functions.

```
if (inFile.peek ( ) != EOF) …        // where inFile is an input stream
// …
outFile.flush ( );                   // where outFile is an output stream
```

10.11 Random Access of File

The I/O functions discussed so far allow only sequential access of files. Each record written is given a unique record number, based on its arrival to the file. When the file is read, the records are accessed, based on arrival sequence.

Random access is facilitated when a given record can be read, irrespective of its arrival sequence, and without reading records that were written before it. C++ I/O system has two functions to facilitate this: **seekg(. . .)** and **seekp(. . .)**. Their prototype are shown in figure 10-18.

Figure 10-18: Prototypes for seekg(. . .) and seekp(. . .) Functions

```
istream &seekg (offtype sOffset, seekdir sOrigin);
ostream &seekp (offtype sOffset, seekdir sOrigin);
```

Note the following:

1. The data type **offtype** is an integer type defined by **ios**, and is capable of containing the largest value of **sOffset**.

2. The data type **seekdir** is an enumeration that has three values

ios :: beg	for beginning of file
ios :: cur	for current location
ios :: end	for end of file

3. "**seekg**" is short for seek get and "**seekp**" is short for seek put. The rationale for this follows.

The C++ I/O system manages two pointers for each file: the **get pointer** (for input) and the **put pointer** (for output). The **seekg** function accesses the **get pointer** while the **seekp** function accesses the put pointer.

The **seekg** function positions the file's **get pointer** to **Offset** bytes from the specified **Origin**. The **seekp** function positions the file's **put pointer** to **Offset** bytes from the specified **Origin**.

Two additional functions are used to determine the current position of each file pointer: **tellg** and **tellp**. Their prototype are shown below:

Figure 10-19: Prototypes for tellg() and tellp() Functions

```
pos_type tellg ( );
pos_type tellp ( );
```

Note: pos_type is a type defined by **ios** and is capable of holding the largest possible value that either function can return. Also, random access is applicable only to files opened for binary access; it does not apply to text files.

10.11 Random Access of File (continued)

Example 10-21: Figure 10-20 provides simple illustration.

Figure 10-20: Illustration of Random Access Processing

```
#include <fstream.h>
#include <iostream.h>
#include "StudentC.h"
// …
StudentC* processFile ( ) // Function to process the student tile
{
    fstream studFile;
    studFile.open("StudFile.dat", ios :: in | ios :: out | ios :: binary);
    int fOffset = 0;  int fSize, numRecords;
    // …
    studFile.seekp (fOffset, ios :: beg);   // Positions file pointer for next write
     //…
    ios :: pos_type  fLocn = studFile.tellp ( );    // Stores file pointer
    studFile.close();
    //…

    // Load the file to an array, and determine the number of records loaded
    studFile.open ("StudFile.dat", ios :: in | ios :: binary | ios :: ate); // Open at EOF
    fSize = studFile.tellg(); // fSize = sizeoff(studFile); Obtain file size in bytes
    StudentC studObject = StudentC();
    numRecords = fSize / sizeof(studObject); // Determine number of records
    StudentC* thisList = new StudentC[numRecords]; // StudentC thisList[numRecords];
    studFile.seekg(Origin, ios :: beg); // Reset file pointer to the beginning
    studFile.read((char*) &thisList, sizeof(thisList));  // Read file into an array
    studFile.close();
    return thisList;
}
```

10.12 Renaming and Removing Files

From time to time it may be necessary to rename, relocate, or remove a file. C++ provides the **rename(…)** function and the **remove(…)** function in header file **stdlib**. The prototypes for both functions follow are outlined below:

Figure 10-21: Prototypes for rename(…) and the remove(…) Functions

```
int rename(const char* oldName, const char*  newName)
int remove(char* fileName)
```

The **rename** function renames the file if both **oldName** and **newName** references the same directory; if they reference different directories, the file is moved to the directory specified in **newName**. The **remove(…)** function simply deletes the file specified. Each function returns zero if the operation is successful and a non-zero value otherwise.

Example 10-22: Figure 10-22 illustrates usage of the **rename(…)** function and the **remove(…)** function.

Figure 10-22: Renaming and Removing a File

```
string fileName1, fileName2;
fstream studFile (fileName1, ios :: in | ios :: out | ios :: binary);
ofstream outFile (fileName2);
// …
fileName1 = "StudFile.dat"; fileName2 = "StudentFile.dat"
rename(fileName1, fileName2); // Renames the student file from StudFile.dat to StudentFile.dat
// …
fileName2 = "DearMom.txt";
remove(fileName2.c_str()); // Removes the DearMom.txt file
// …
```

10.13 Summary and Concluding Remarks

Here is a summary of what we have discussed in this lecture:

- C++ provides a comprehensive I/O stream consisting of a hierarchy of several classes. Familiarity with this hierarchy is essential to understanding how I/O is managed and how to program for I/O in C++.
- You can overload the << and >> operators for an I/O stream.
- C++ allows you to format I/O via the **ios** member functions or the I/O manipulation functions.
- C++ allows you to process text files via the **ofstream** and **ifstream** classes.
- C++ allows you to process binary files via the **ostream** and **istream** classes.
- C++ also supports random file access via the **fstream** class.
- Through the **rename(…)** and **remove(…)** functions, you can rename or remove a file.

Like most high level programming languages, C++ is not reputed for its file handling capabilities (contrary to what some of its proposals might be prone to argue). In fact, most high level languages (C++ included) are very limited in the file handling facilities that they provide.

Where sophisticated file management is required, the common practice in industry is summarized here:

- Create a database (of several relational files) by using the services of a database management system (DBMS) or CASE tool.
- The DBMS suit or CASE tool will support several high level languages (typically, C++ is included).
- The DBMS suit or CASE tool will also support a universal database language such as SQL (structured query language) or UML (unified modeling language).
- Use a high level language to help create the user interface of the software product being constructed.
- Whenever database access is required, insert embedded SQL (or UML) statements in the high level language program.

This approach will become clearer to you as you proceed with your studies. For now, you are required to master basic file processing as discussed in the foregoing sections.

10.14 Recommended Readings

[Friedman & Koffman 2011] Friedman, Frank L. & Elliot B. Koffman. 2011. *Problem Solving, Abstraction, and Design using C++*, 6th Edition. Boston: Addison-Wesley. See chapter 8.

[Gaddis, Walters & Muganda 2014] Gaddis, Tony, Judy Walters, & Godfrey Muganda. 2014. *Starting out with C++ Early Objects*, 8th Edition. Boston: Pearson. See chapter 13.

[Savitch 2013] Savitch, Walter. 2013. *Absolute C++*, 5th Edition. Boston: Pearson. See chapter 12.

[Savitch 2015] Savitch, Walter. 2015. *Problem Solving with C++*, 9th Edition. Boston: Pearson. See chapter 6.

[Yang 2001] Yang, Daoqi. 2001. *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*. New York, NY: Springer. See chapter 4.