Unit 9 Files and Streams

Structure:

9.1 Introduction

In the previous unit you studied about polymorphism and its types. You also studied about function overloading and its types. In this unit you are going to learn the ways to handle files in C++ and to process input and output operations, and get an idea of preprocessor directives in C++. You are also going to study the procedure to pass arguments from command line prompt. You are already using files to store your program. The files can also be used to store input for a program or to receive output from a program. These files used for program input-output are the same files that you used to store your programs. The file streams allow you to write programs that handle file and keyboard inputs as well as file and screen outputs in a unified way. In C++, fstream class is used to perform file processing. Unlike the FILE structure, fstream is a complete C++ class with constructors, a destructor and overloaded operators. To perform file processing, you can declare an instance of an fstream object. If you do not yet know the name of the file you want to process, you can use the default constructor.

Objectives:

After studying this unit, you should be able to:

- explain the way file input-output is handled in C++
- describe the read-and-write operation of string content to a file
- describe the methods to transfer output to printer
- discuss preprocessor directives

9.2 Introduction to files and streams in C++

File input and output in C++ is completely handled by predefined file and stream-related classes. The cin and cout statements, which we were using for transfer of input from keyboard, and that of output to display screen, are also predefined objects of these predefined stream classes.

The declarations of these stream classes are contained in the header file 'iostream.h' which you have been using in your programs. Similarly, the declarations for classes used for disk I/O is provided in header file fstream.h. Let us first understand what 'a stream' is. A stream is a flow of characters. If the flow is into your program, then it is called 'input stream', and if the flow is out of your program, then it is called 'output stream'. Your program will take input from the keyboard if input stream flows from the keyboard. And the program will take output from the file if the input stream flows from a file. Similarly, an output stream can go to the screen or to a file. You have already been using streams in your programs. The cin is the inuput stream connected to the keyboard, and the cout is the output stream connected to the screen.

The stream classes are organized in a hierarchical manner. But an overview of how they are organized will provide an excellent example of inheritance, and will help us understand files and streams in a better way.

The following figure 9.1 shows a portion of stream class hierarchy in C++ which we will be using in our programs in this unit.

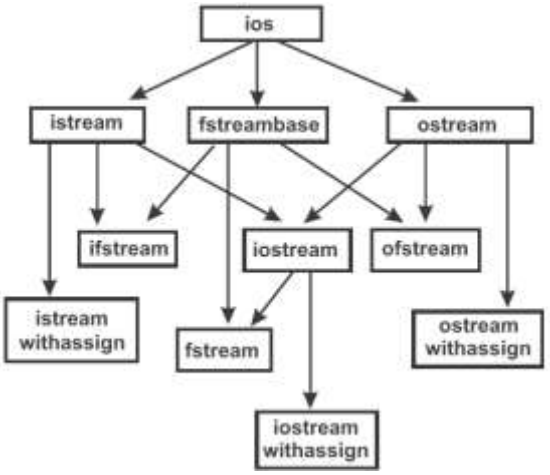

Figure 9.1: Stream classes in C++

As you can see in figure 9.1, all the stream classes are inherited from *the ios class*. It provides support for formatted and unformatted I/O operations. It contains many constants and member functions that can be used by all other input and output classes. All constants and functions that are necessary for handling input and output operations are present in this ios class. *istream (input stream) class* is derived from ios class, and all the necessary functions for input handling are present in it. It provides facilities for formatted and unformatted input. Besides, it contains pointer to a buffer object, i.e. streambuf object. Some functions that are defined in this class are get(), getline(), read() and overload extraction operators (>>).

ostream (output stream) class is also derived from ios class, and performs all types of output related functions like put(), write(). It provides facilities for formatted output. The overload insertion operator (<<) is defined in this class.

iostream (input/output) stream class inherits the properties of istream and ostream class through multiple inheritance, and hence, contains all the input and output functions.

streambuf class provides the interface to physical devices through buffers.

Three classes - istream_withassign, ostream_withassign, iostream with assign are inherited from istream, ostream, iostream respectively, which add assignment operators to these classes. cout is a predefined object of the ostream_withassign class. cin is an object of the istream_withassign class.

The ifstream class is used for input files and it is inherited from classes istream and fstreambase. Similarly, class ostream is used for output files, and is derived from classes- ostream and fstreambase. Files used for both input output operations are inherited from the class - fstream. The class fstream is inherited from both iostream and fstreambase. The classes fstream, ifstream and ofstream are declared in the fstream.h header file. The file fstream.h also includes iostream.h, so the programs using fstream.h need not explicitly include iostream.h.

Self Assessment Questions

1. cin is a predefined object of _____ class.
2. cout is a predefined object of _____ class.
3. All the stream classes are inherited from _____class.

9.3 Character and String input and output to files

In this section you will study the ways to read characters and strings from the text file, and to write it to the text file with the help of a few programs.

To write characters to a file, you need to create an object of ofstream class. The put() function should be used with the object to write the data to the file. The put() function writes one character at a time. The syntax of the put() function is as follows:

objectname.put(character variable)

The following program shows the use of put() function:

```
#include<fstream.h>
# include<string.h>
void main()
{
ofstream outfile("test.txt");
char str[ ]="hello world";
for (int j=0;j<strlen(str);j++)
outfile.put(str[j]);
}
```

In the program shown above, the object named 'outfile of ofstream class' is created. The constructor of the ofstream class is invoked and the object outfile is assigned the text file named 'test.txt'. In the output directory, specified in the compiler, a text file is created. And if there exists a file with the same name, then the file is overwritten. The contents of the string str is written to the file test.txt. It is to be noted that we have included fstream.h header file. Since the insertion operator (<<) is overloaded in the ostream class (of stream is derived from ostream), it can be used along with the ofstream object to write text directly to the file. The following program shows that:

```
#include<fstream.h>
void main()
{
ofstream outfile("example.txt");
```

```
outfile<<“Welcome\n”;
outfile<<“See you\n”;
}
```

You need to create an object of ifstream class to read data from file. To read the data from the file, you have to use the function ‘get()’ or ‘getline()’. The ‘get()’ function also reads one character at a time. The following is the syntax of the ‘get()’ function.

objectname.get(character variable)

Here the objectname is an object of ifstream class and the character to be read from the file is stored in a variable.

The following is the program to show how to read character from file. The program shown below reads the contents of the file character by character, and displays the character on computer screen.

```
#include<fstream.h>
#include <conio.h>
void main()
{
ifstream infile("test.txt");
char ch;
clrscr();
while (infile)          // infile becomes 0 when eof condition is reached
{infile.get(ch);
cout << ch;}
getch();}
```

You can observe in the above program that the end of file (eof) condition is checked using the object name. You can also use eof() function to check this condition. The eof() function returns non-zero value when an end of file is encountered, and returns zero when reading the file.

The getline() function is also used to read contents from a file. But unlike get() function it reads the contents line by line. The program below shows the use of getline function.

```
#include<fstream.h>
#include <conio.h>
void main()
{
ifstream infile("test.txt");
char buffer[80];
clrscr();
while (!infile.eof())      //returns nonzero value when eof condition is reached
{infile.getline(buffer,80);
cout << buffer;}
getch();}
```

The extraction operator (>>) can be used with ifstream object to read the text as this operator is overloaded in the istream class. (ifstream is derived from istream). However, it reads one word at a time. The following program shows the same:

```
#include<fstream.h>
#include <conio.h>
void main()
{
ifstream infile("test.txt");
char buffer[25];
clrscr();
while (infile)              //infile becomes 0 when eof condition is reached
{infile>>buffer;
cout << buffer;
}
getch();
}
```

In these programs, while reading contents from the file, we have presumed that the files are existing on the disk. But it would be a good practice if you apply a check to find whether the file to be read exists on the disk or not.

Many such status checks are contained in ios class. In the program shown below, the check applied reports an error if the file doesn't exist.

```
#include<fstream.h>
#include <conio.h>
void main()
{
ifstream infile("test.txt");
if (!infile)                    // check for error while opening the file
cout<<"Cannot open file";
else
{
char buffer[25];
clrscr();
while (infile) //infile becomes 0 when eof condition is reached
{infile>>buffer;
cout << buffer;}
}
getch();}
```

Self Assessment Questions:

4. To write characters to a file, you need to create an object of _____ class.
5. You need to create an object of _____ class to read data from file.
6. The put() function writes _____ character at a time to the files.
7. The _____ function reads contents from a file, line by line.
8. The _____ function returns non-zero value when an end of file is encountered.

9.4 Command Line Arguments and Printer Output

You have already studied that when a program is compiled and linked, then an executable file is produced by the compiler. And when the program runs, the execution of programs begins from the main() function. Till now we have declared the main function either as void main() or as int main(). This

version of main doesn't take any arguments. But many a time, programs need some kind of input to work with. For example, you are writing a code to count the number of words in a text file.

The user has to have some way of telling the program which file to open. To do this, you may take this approach:

```
int main()
 {
 using namespace std;
cout << "Enter the name of the file: ";
char strFilename[255];
 cin >> strFilename;       // open file and process it
}
```

The major problem associated with this approach is that every time the file runs, the program waits for the user to enter the input. It indicates that execution of the program cannot be automated easily. For example, if you want to execute this program on 600 files per week, then the program will not proceed. Instead, it will wait until you enter the name of the file every time it is executed.

Command line arguments

For programs that have minimal and/or optional inputs, command line arguments offer a great way to make programs more modular. The optional string arguments that are given by the user to a program during execution are known as 'command line arguments'. The operating system passes these arguments to the program, and the program uses them as input. Programs are normally run by invoking them by name. To run a program you have to type the name of the program on the command line. For example, to run the executable file "WordCount" that is located in the root directory of the C:\ drive on a Windows machine, you could type:

```
C:\\>WordCount
```

To pass the command line arguments to the program 'WordCount', you have to list the command line arguments after the executable name. The example is shown below:

```
C:\\>WordCount Myfile.txt
```

You can see in the above statement that when the program named 'WordCount' is executed, Myfile.txt will be passed to it as a command line argument. It is possible for a program to have a number of command line arguments. For example:

C:\\>WordCount Myfile.txt Myotherfile.txt

This also works for other command line operating systems, such as Linux (though your prompt and directory structure will undoubtedly vary).

If a program is run from an IDE (Integrated Development Environment) then some way should be provided by IDE to enter command line arguments. For example, in Microsoft Visual Studio 2005, right click on your project in the solution explorer, and then choose properties. Open the "Configuration Properties" tree element, and choose "Debugging". In the right pane, there is a line called "Command Arguments". You can enter your command line arguments there for testing, and they will be automatically passed to your program when you run it.

Now you have already studied the way to provide command line arguments to a program. The next thing you have to study is to access them from within our C++ program. To achieve this task, we use some other form of the main program that we have used in our programs till now. We have to pass two arguments in the main function i.e. argc and argv. The names of the arguments are by convention. The example is as follows:

int main(int argc, char *argv[])

argc is an argument containing the number of arguments passed to the program. argc will always be at least 1, because the first argument is always the name of the program itself! The value of argc will be incremented by 1 each time the user provides the command line argument. Each command line argument the user provides will cause argc to increase by 1.

The actual arguments are stored in argv. Although the declaration of argv looks intimidating, argv is really just an array of C-style strings. The length of this array is argc. Let's write a short program to print the value of all the command line parameters:

```
#include <iostream>
int main(int argc, char *argv[])
{
using namespace std;
cout << "There are " << argc << " arguments:" << endl;
// Loop through each argument and print its number and value
for (int nArg=0; nArg < argc; nArg++)
cout << nArg << " " << argv[nArg] << endl;
return 0;
}
```

If we provide the following command line argument - “Myfile.txt” to the program, the output will be:

```
There are 3 arguments:
C:\\WordCount
Myfile.txt
100
```

The name of the running program is always the argument 0. The two command-line parameters are argument 1 and 3. Going back to our previous example, let us go ahead and partially write WordCount so that it uses command line arguments instead of asking the user for input. Command line arguments are useful whenever we want to pass arguments while the program is executing. The following is one more example which shows the implementation of command line arguments, and simply displays them.

```
//comline.cpp
# include <iostream.h>
void main(int argc, char* argv[])
{
cout<<endl<<”argc= ”<<argc;
for (int j=0;j<argc;j++)
```

```
cout<<endl<<”Argument ”<<j<<”= ”<<argv[j];
}
```

If you invoke the above program with the statement

C:/tc> ABC EFG PQR XYZ

Then the output will be as follows

argc= 4

Argument0= c:/tc/comline.exe

Argument1= ABC

Argument2= EFG

Argument3= PQR

You can see in the program above that the number of the arguments including the name of the file is contained in argc variable. The strings passed to the program along with the path of the executable program is contained in the argv variable. You can refer them by their index number. Now we will study how data can be sent to the printer. It is similar to sending data to a disk file seen in the last section. The program uses filenames predefined in DOS for various hardware devices. PRN or LPT1 is the filename for the printer which is connected to the first parallel port. LPT2, LPT3 and so on can be used if the printer is connected to the second, the third parallel ports respectively.

The following program prints a test message to the printer

```
//printtest.cpp
# include<fstream.h>
void main()
{ ofstream prnfile(“PRN”);
prnfile<<”This is a test print page”;
}
```

As you can see in the program above, an ofstram class object has been created and the printer file name has been associated with it and the contents have been saved to the object. As you can see, the code and the way we direct the contents to a stream do not differ. Whether it is sending

contents to display screen or printer or disk file, the syntax remains the same. That is the beauty of stream class organization hierarchy.

Self Assessment Questions

9. The optional string arguments that are given by the user to a program during execution are known as _____.
10. _____ variable contains the number of arguments passed to the program.
11. _____ variable contains the actual arguments passed to the program.

9.5 Preprocessor Directives

There are some instructions given to the compiler to preprocess the information before the actual compilation begins. These instructions are known as preprocessor directives. All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. The preprocessor directives do not end with semicolon. You are familiar with the #include directive. The purpose of this macro is to include a header file into the source file.

So many directives are supported by C++: #include, #define, #if, #else, #line, etc are only a few to name. Let us study a few important directives in detail.

9.5.1 The #define Preprocessor

The symbolic constants are created by #define directive. These symbolic constants are known as macros. The syntax to declare macros are:

```
#define macro-name replacement-text
```

When this statement appears in a file, then the macro name in the program is replaced with the replacement text before the program is compiled. For example:

```
#include <iostream.h>
#define PI 3.14159

int main ()
{
   cout << "Value of PI :" << PI << endl;
      return 0;
    }
```

Now let us preprocess this code to see the result, assuming that we have the source code file. We will compile this file with –E option and save the result in result.p file. Now, if you check the result.p file, you will see that it has lots of information, and at the bottom of the file, you will find the value replaced as follows:

```
$gcc -E test.cpp > test.p
int main ()
{
    cout << "Value of PI :" << 3.14159 << endl;
   return 0;
}
```

Function-Like Macros

You can use #define to define a macro which will take argument as follows:

```
#include <iostream.h>
#define MIN(a,b) ( ((a)<(b)) ? a : b)
int main ()
{
  int i, j;
   i = 100;
   j = 30;
  cout <<"The minimum is " << MIN(i, j) << endl;
   return 0;
}
```

The output of the above program will be:
The minimum is 30

9.5.2 Conditional Compilation

C++ provides the facility to compile the selective portions of our program's source code. This can be performed by using some directives. This procedure is known as conditional compilation.

The conditional preprocessor construct is much like the *if* selection structure. Consider the following preprocessor code:

```
#ifndef NULL
#define NULL 0
```

```
#endif
```

You can compile a program for debugging purpose, and can turn 'debugging' on or off using a single macro as follows:

```
#ifdef DEBUG
 cerr <<"Variable x = " << x << endl;
#endif
```

This causes the cerr statement to be compiled in the program if the symbolic constant DEBUG has been defined before directive #ifdef DEBUG. You can use #if 0 statement to comment out a portion of the program as follows:

```
#if 0
   code prevented from compiling
#endif
```

The following example illustrates the above concept:

```
#include <iostream.h>
#define DEBUG
#define MIN(a,b) (((a)<(b)) ? a : b)
int main ()
{
   int i, j;
   i = 100;
   j = 30;
#ifdef DEBUG
   cerr <<"Trace: Inside main function" << endl;
#endif
#if 0
   /* This is commented part */
   cout << MKSTR(HELLO C++) << endl;
#endif
   cout <<"The minimum is " << MIN(i, j) << endl;
#ifdef DEBUG
   cerr <<"Trace: Coming out of main function" << endl;
```

```
#endif
   return 0;
}
```

The output of the above program will be:

Trace: Inside main function

The minimum is 30

Trace: Coming out of main function

9.5.3 Predefined C++ Macros

C++ provides a number of predefined macros. Table 9.1 shows the predefined macros provided by C++.

Table 9.1: Predefined macros

Macro	Description
_LINE__	An integer that gives the current line number of the program when it is being compiled.
__FILE__	A string that gives the current file name of the program when it is being compiled.
__DATE__	This contains a string of the form month/day/year that is the date of the translation of the source file into object code.
__TIME__	This contains a string of the form ‘hour:minute:second’ that is the time at which the program was compiled.

The example to show the usage of the above macros is as follows:

```
#include <iostream>
using namespace std;
int main ()
{
   cout << "Value of __LINE__ : " << __LINE__ << endl;
   cout << "Value of __FILE__ : " << __FILE__ << endl;
   cout << "Value of __DATE__ : " << __DATE__ << endl;
   cout << "Value of __TIME__ : " << __TIME__ << endl;
   return 0;
}
```

The output of the above program is as follows:

Value of __LINE__ : 6

Value of __FILE__ : test.cpp

Value of __DATE__ : Feb 01 2015

Value of __TIME__ : 18:52:48

Self Assessment Questions

12. _____are the instructions given to the compiler to preprocess the information before actual compilation begins.
13. The symbolic constants are created by #define directive, and are known as macros. (True/False).
14. _____ macro is a string that gives the current file name of the program when it is being compiled.

9.6 Summary

- A stream is a flow of characters. If the flow is into your program then it is called input stream, and if the flow is out of your program then it is called output stream.
- The stream classes are organized in a hierarchical manner.
- ofstram class is used to write characters to a file.
- If stream class is used to read data from a file.
- The optional string arguments that are given during execution by the user to a program are known as command line arguments.
- argc is an interger type argument containing the number of arguments passed to the program. argv contains the actual arguments passed to the program.
- There are some instructions given to the compiler to preprocess the information before actual compilation begins. These instructions are known as preprocessor directives. All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line.
- The symbolic constants are created by #define directive. These symbolic constants are known as macros.

- C++ provides the facility to compile the selective portions of our program's source code. This can be performed by using some directives. This procedure is known as conditional compilation.

9.7 Terminal Questions

1. Write a program that opens a text file named sample.txt and then prints the contents to the printer.
2. Write a program that will be called mtype.cpp, which imitates the type command of the DOS. The user should be able to specify the name of the text file along with mtype which is read, and the contents are displayed on the screen.
3. Define streams. List and explain all the stream classes in C++.
4. Explain how to use command line arguments with the help of an example.
5. Write a note on 'preprocessor directives'.

9.8 Answers

Self Assessment Questions

1. istream_withassign
2. ostream_withassign
3. ios
4. ofstream
5. ifstream
6. one
7. getline()
8. eof()
9. command line arguments
10. argc
11. argv
12. Preprocessor directives
13. True
14. __FILE__

Terminal Questions

1. //file2prn.cpp

```
# include<fstream.h>
void main()
{
char ch;
ifstream infile;
infile.open("sample.txt");
ofstream outfile;
outfile.open("PRN");
while(infile.get(ch))
{ outfile.put(ch);
}
}
```

2. //mytype.cpp

```
# include<fstream.h>
# include<process.h>
void main(int argc, char* argv[])
{ if (argc!=2)
{ cerr<<“\nFormat: mytype filename”;
  exit(-1);
}
char ch;
ifstream infile;
infile.open(arg[1]);
if (!infile)
{ cerr<<“cannot open”<<argv[1];
exit(-1);
}
while(infile)
{infile.get(ch);
cout<<ch;}
}
```

3. A stream is a flow of characters. If the flow is into your program then it is called input stream, and if the flow is out of your program, then it is called output stream. The stream classes are organized in a hierarchical manner. For more details refer section 9.2.
4. For programs that have minimal and/or optional inputs, command line arguments offer a great way to make programs more modular. The optional string arguments that are given by the user to a program during execution are known as command line arguments. For more details refer section 9.4.
5. There are some instructions given to the compiler to preprocess the information before the actual compilation begins. These instructions are known as preprocessor directives. All preprocessor directives begin with #, and only white-space characters may appear before a preprocessor directive on a line. For more details refer section 9.5.

References:

- Object Oriented Programming with C++ - Sixth Edition, by E Balagurusamy. Tata McGraw-Hill Education.
- Problem Solving with C++, 6/e, By Savitch, Pearson Education India.
- C++ for Engineers and Scientists, By Gary Bronson, Cengage Learning.
- http://enggedu.com/
- http://www.tutorialspoint.com