CHAPTER THREE

FILE MANAGEMENT

File Management

- File handling is an important part of all programs.
- Most of the applications will have their own features to save some data to the local disk and read data from the disk again.
- Files which are on the secondary storage device are called physical files.
- In order to process file through program, logical file must be created on the RAM.
- This logical file is nothing but an object having file data type.
- As an object there should be a variable identifier that points to it.
- This variable is called file variable and some times also called file handler.
- C++ File I/O classes simplify such file read/write operations for the programmer by providing easier to use classes.

Streams and Files

- The I/O system supplies a consistent interface to the C++ programmer independent of the actual device being accessed.
- This provides a level of abstraction between the programmer and the device.
- This abstraction is called **stream**. The actual device is called a **file**.

Streams

- The C++ file system is designed to work with a wide variety of devices, including terminals, disk drives, and tape drives.
- Even though each device is very different, the C++ file system transforms each into a logical device called stream.
- There are two types of streams: text and binary.

Text Streams

- A text stream is a sequence of characters.
- In a text stream, certain character translations may occur as required by the host environment.
- For example a new line may be converted to a carriage return/linefeed pair.
- There may not be a one-to-one relationship between the characters that are written (or read) and those on the external device.
- Because of possible transformations, the number of characters written (or read) may not be the same as those on the external device.

Binary streams

- A binary stream is a sequence of bytes with a one-to-one correspondence to those in the external device i.e., no character translations occur.
- The number of bytes written (or read) is the same as the number on the external device.
- However, an implementation-defined number of null bytes may be appended to a binary stream.
- These null bytes might be used to pad the information so that it fills a sector on a disk, for example.

Files

- In C++, a file can be anything from a disk file to a terminal or printer.
- You associate a stream with a specific file by performing an open operation.
- Once a file is open, information can be exchanged between it and a program.
- All streams are the same but all files are not.
- If the file can support position requests, opening that file also initializes the file position indicator to the start of the file.

- As each character is read from or written to the file, the position indicator is incremented.
- You disassociate a file from a specific stream with a close operation.
- If you close a file opened for output, then contents, if any, of its associated stream are written to the external device. -- this process is referred to as flushing the stream.
- All files are closed automatically when the program terminates normally.
- Files are not closed when a program terminates abnormally.
- Each *stream* that is associated with a file has a *file control structure* of *type FILE*.
- This structure FILE is defined in the header stdio.h.

The File Pointer

- A file pointer is a pointer to information that defines various things about the file, including its name, status, and the current position of the file.
- In essence, the file pointer identifies a specific disk file and is used by the associated stream to direct the operation of the I/O functions.
- A file pointer is a pointer variable of type FILE.
- FILE * fp;

The standard streams

- When ever a program starts execution, three streams are opened automatically.
  - stdin --- standard input.
  - stdout -- standard output
  - stderr -- standard error
- Normally, these streams refer to the console.
- Because the standard streams are file pointers, they can be used by the ANSI C file system to perform I/O operations on the console.

C++ File I/O Classes and Functions

- To perform file I/O, the header file fstream.h is requied. fstream.h defines several classes, including ifstream, ofstream, and fstream.
- These classes are derived form istream and ostream, repectively. istream and ostream are derived form ios.

- Three file I/O classes are used for File Read/Write operations:
  - ifstream - Can be used for File read/input operations
  - ofstream - Can be used for File write/output operations
  - fstream - Can be used for both read/write c++ file I/O operations
- These classes are derived directly or indirectly from the classe istream, and ostream.
- We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream.
- Therefore, we have already been using classes that are related to our file streams.
- And in fact, we can use our file streams the same way we are already used to use cin and cout, with the only difference that we have to associate these streams with physical files.
- Let's see an example:

Text and Binary Files

- In file processing, files are generally classified into two as
  - Text file and
  - Binary file
- Text file is a file in which its content is treated as a sequence of characters and can be accessed sequentially.
- Where as binary file is a file in which its content is treated as record sequence in a binary format.
- Binary format refers to the actual format the data is going to be placed and processed in the memory which is directly related to its data type.

- For example, the value int count 321 will be stored in three byte if it is written in text file considering the digit sequence ‘3’, ‘2’, ‘1’.
- It will be stored in two byte if it is written in binary file since int requires two byte to store any of its value.
- When you open the binary file you will see the character equivalence of the two bytes.
- 321 in binary equals 0000 0001 0100 0001
- The first byte holds the character with ASCII value equals to one and the second byte a character with ASCII value equals 65 which is ‘A’.
- Then if you open the binary file you will see these characters in place of 321.

Text File processing

- File processing involves the following major steps
  - Declaring file variable identifier
  - Opening the file
  - Processing the file
  - Closing the file when process is completed.

Opening and Closing a file

- An open file is represented within a program by a stream object and any input or output operation performed on this stream object will be applied to the physical file associated to it.
- In C++, you open a file by linking it to a stream. Before you can open a file, you must first obtain a stream.
- There are three types of streams: input, output, and input/output.
- To create an input stream, you must declare the stream to be of class ifstream.
- To create an output stream, you must declare it as class ofstream.
- Streams that will be performing both input and output operations must be declared as class fstream.

- ifstream in ; //input stream
- ofstream out ; // output stream
- fstream io ; // input and output

- Once you have declared a stream, you associate it with a file by using the method open().
- The method open ( ) is a member of each of the three stream classes. Its prototype is:
- void open (const char *filename, int mode, int access = filebuf::penprot);
- Where:
  - Filename is the name of the file.
  - The value of the mode determines how the file is opened. It must be one (or more) of these values:

Mode	Description
ios::app	Write all output to the end of the file
ios::ate	Open a file for output and move to the end of the file (normally used to append data to a file). Data can be written anywhere in the file.
ios::binary	Cause the file to be opened in binary mode.
ios::in	Open a file for input
ios::nocreate	If the file does not exist, the open operation fails.
ios::noreplace	If the file exists, the open operation fails.
ios::out	Open a file for output
ios:trunc	Discard the file's content if it exists (this is also the default action ios::out)

- You can combine two or more of these values by using them together.

```
ofstream out ;
out.open ( "test", ios::out); // correct statement
ofstream out1 ;
out.open ( " test"); // the default value of mode is ios::out –   // correct statment
```

- To open a stream for input and output, you must specify both the ios::in and the ios::out mode values. (No default value for mode is supplied in this case.)

```
fstream myStream;
myStream.open ( "test", ios::in | ios::out );
If open ( ) fails, myStrream will be zero
if (myStream){
cout << "Cannot open a file.\n";
// handle error
}
```

- Each one of the open() member functions of the classes ofstream, ifstream and fstream has a default mode that is used if the file is opened without a second argument:

class	default mode parameter
ofstream	ios::out
ifstream	ios::in
fstream	ios::in | ios::out

- To check if a file stream was successful opening a file, you can do it by calling to member is_open() with no arguments.
- This member function returns a bool value of true in the case that indeed the stream object is associated with an open file, or false otherwise:
- if (myfile.is_open()) { /* ok, proceed with output */ }
- ifstream myStream ("myfile"); // open file for input

- When we are finished with our input and output operations on a file we shall close it so that its resources become available again.
- In order to do that we have to call the stream's member function close().
- This member function takes no parameters, and what it does is to flush the associated buffers and close the file:
- myfile.close();

- Once this member function is called, the stream object can be used to open another file, and the file is available again to be opened by other processes.
- In case that an object is destructed while still associated with an open file, the destructor automatically calls the member function close(). The close method takes no parameters and returns no value.

Opening file for writing

```
// writing to file
ofstream MyFile("c:\\a\\filename.txt");
MyFile << "hello world!";
myFile.close();
```

```
string myText;
while(getline(MyReadFile,myText)){
   cout<<myText;
}
// Close the file
MyReadFile.close();
```

Checking state flags

- In addition to eof(), which checks if the end of file has been reached, other member functions exist to check the state of a stream (all of them return a bool value):

Function	Description
bad()	Returns true if a reading or writing operation fails. For example in the case that we try to write to a file that is not open for writing or if the device where we try to write has no space left.
fail()	Returns true in the same cases as bad(), but also in the case that a format error happens, like when an alphabetical character is extracted when we are trying to read an integer number.
eof()	Returns true if a file open for reading has reached the end.
good()	It is the most generic state flag: it returns false in the same cases in which calling any of the previous functions would return true.

- In order to reset the state flags checked by any of these member functions we have just seen we can use the member function clear(), which takes no parameters.
- **get and put stream pointers**
  - All I/O streams objects have, at least, one internal stream pointer:
  - ifstream, like istream, has a pointer known as the get pointer that points to the element to be read in the next input operation.
  - ofstream, like ostream, has a pointer known as the put pointer that points to the location where the next element has to be written.
  - Finally, fstream, inherits both, the get and the put pointers, from iostream (which is itself derived from both istream and ostream).

- These internal stream pointers that point to the reading or writing locations within a stream can be manipulated using the following member functions:
- **tellg() and tellp()**
  - These two member functions have no parameters and return a value of the member type pos_type, which is an integer data type representing the current position of the get stream pointer (in the case of tellg) or the put stream pointer (in the case of tellp).

seekg() and seekp()

- These functions allow us to change the position of the get and put stream pointers. Both functions are overloaded with two different prototypes. The first prototype is:
  - seekg (position);
  - seekp (position);
- Using this prototype the stream pointer is changed to the absolute position (counting from the beginning of the file).
- The type for this parameter is the same as the one returned by functions tellg and tellp: the member type pos_type, which is an integer value.

- The other prototype for these functions is:
  - seekg (offset, direction);
  - seekp (offset, direction);
- Using this prototype, the position of the get or put pointer is set to an offset value relative to some specific point determined by the parameter direction.
- offset is of the member type off_type, which is also an integer type.
- And direction is of type seekdir, which is an enumerated type (enum) that determines the point from where offset is counted from, and that can take any of the following values:

ios::beg	offset counted from the beginning of the stream
ios::cur	offset counted from the current position of the stream pointer
ios::end	offset counted from the end of the stream

```
#include <iostream>
#include <fstream>
using namespace std;
int main () {
  long begin,end;
  ifstream myfile ("example.txt");
  begin = myfile.tellg();
  cout<<"begin="<<begin<<endl;
  myfile.seekg (0, ios::end);
  end = myfile.tellg();
  cout<<"end="<<end<<endl;
  myfile.close();
  cout << "size is: " << (end-begin) << " bytes.\n";
  return 0;
}
```

Binary File processing

- In binary files, to input and output data with the extraction and insertion operators (<< and >>) and functions like getline is not efficient, since we do not need to format any data, and data may not use the separation codes used by text files to separate elements (like space, newline, etc...).
- File streams include two member functions specifically designed to input and output binary data sequentially: write and read.
- The first one (write) is a member function of ostream inherited by ofstream.
- And read is a member function of istream that is inherited by ifstream.
- Objects of class fstream have both members. Their prototypes are:

- write (memory_block, size);
- read (memory_block, size);
- Where memory_block is of type "pointer to char" (char*), and represents the address of an array of bytes where the read data elements are stored or from where the data elements to be written are taken.
- The size parameter is an integer value that specifies the number of characters to be read or written from/to the memory block.
- There are two ways to write and read binary data to and from a file.
  - get ( ) and put ( )
  - read ( ) and write ( )
- If you will be performing binary operations on a file, be sure to open it using the ios::binary mode specifier.

get ( ) and put ( )

- These functions are byte-oriented.
  - get ( ) will read a byte of data.
  - put ( ) will write a byte of data.
- The get ( ) method has many forms.
  - istream & get(char ch);
  - ostream & put (char ch);
- The get ( ) method read a single character from the associated stream and puts the value in ch, and returns a reference to the stream. The put ( ) method writes ch to the stream and returns a reference to the stream.

Example

```
char in;
ifstream in ( "test", ios::in | ios::binary);
if (!in){
cout <<"Cannot open file";
return 1;
}
while (in) //in will be 0 when eof is reached
{  in.get ( ch );
    cout << ch;
}
```

- When the end-of-file is reached, the stream associated with the file becomes zero.

Example: put()

```
ofstream out ( "chars", io::out |
ios::binary);
for (int i= 0; i < 256; i++)
out.put ( (char ) i ) ; //write all
characters to disk
out.close ( );
```

More read ( ) and write ( )

- The read ( ) method reads num bytes from the associated stream, and puts them in a memory buffer (pointed to by buf).
  - istream & read (unsigned char * buf, int num);
- The write ( ) method writes num bytes to the associated stream from the memory buffer (pointed to by buf).
  - ostream & write (const unsigned char * buf, int num);
- If the end-of-file is reached before num characters have been read, then read ( ) simply stops, and the buffer contains as many characters as were available.
- You can find out how many characters have been read by using another member function, called gcount ( ), which has the prototype:
  - int gcount ( );

More get ( ) functions

- The method get ( ) is overloaded in several ways.
  - istream &get (char *buf, int num, char delim = '\n');
- This get ( ) method reads characters into the array pointed to by the buf until either num characters have been read, or the character specified by delim has been encountered.
- The array pointed to by buf will be null terminated by get ( ).
- If the delimiter character is encountered in the input stream, it is not extracted.
- Instead, it remains in the stream until the next input operation.

a. int get ( )

- It returns the next character from the stream. It returns EOF if the end of file is encountered.

b. getline ( )

- istream & getline (char *buf, int num, char delim ='\n');

- This method is virtually identical to the get (buf, num, delim) version of get ( ).
- The difference is getline ( ) reads and removes the delimiter from the input stream.

reading a complete binary file

```
ifstream file ("chars1.bin", ios::in|ios::binary|ios::ate);
if (file.is_open())
 {
   long size = file.tellg();
   char memblock[size];
   file.seekg (0, ios::beg);
   file.read (memblock, size);
   file.close();
   cout << "the complete file content is in memory";
   for(int i=0;i<size;i++)
   cout<<memblock[i];
 }
 else cout << "Unable to open file";
```