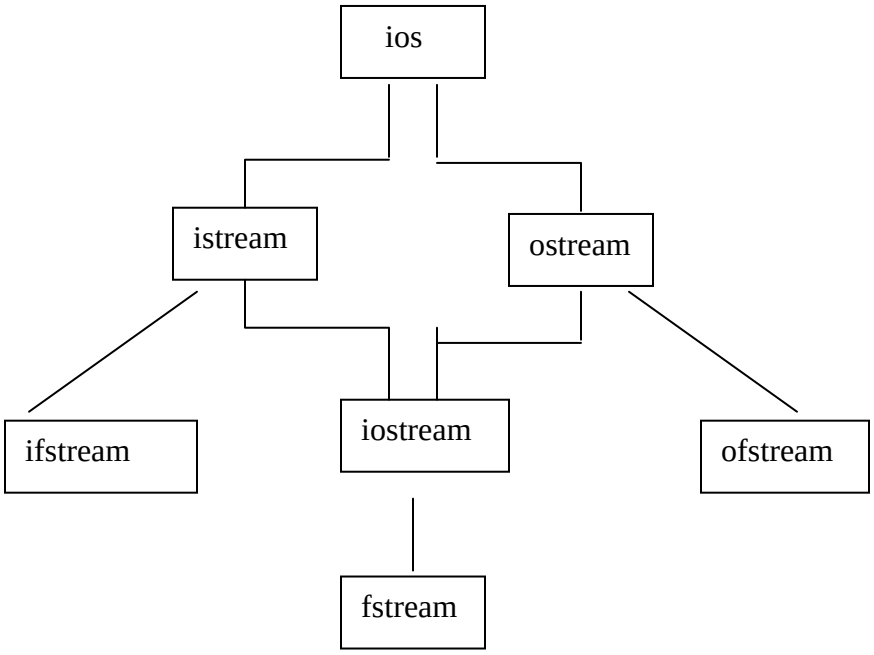
Chapter Seven
File Operations (File I/O)

7.1 Introduction

- The data created by the user and assigned to variables with an assignment statement is sufficient for some applications. With large volume of data most real-world applications use a better way of storing that data. For this, disk files offer the solution.
- When working with disk files, C++ does not have to access much RAM because C++ reads data from your disk drive and processes the data only parts at a time.

7.2 Stream

- Stream is a general name given to *flow of data*. In C++, there are different types of streams. Each stream is associated with a particular class, which contains member function and definition for dealing with file. Lets have a look at the figure:



- According to the above hierarchy, the class iostream is derived from the two classes' istream and ostream and both istream and ostream are derived from ios. Similarly the class fstream is derived from iostream.

Generally two main header files are used iostream.h and fstream.h. The classes used for input and output to the video display and key board are declared in the header file iostream.h and the classes used for disk file input output are declared in fstream.h.

- Note that when we include the header file fstream.h in our program then there is no need to include iostream.h header file. Because all the classes which are in fstream.h they are derived from classes which are in iostream.h therefore, we can use all the functions of iostream class.

7.3: Operation With File

- First we will see how files are opened and closed. A file can be defined by following class ifstream, ofstream, fstream, all these are defined in fstream.h header file.
    - if a file object is declared by *ifstream* class, then that object can be used for *reading* from a file.
    - if a file object is declared by *ofstream* class, then that object can be used for *writing* onto a file.
    - If a file object is declared by *fstream* class then, that object can be used for both *reading from* and *writing to* a file

7.4: Types of Disk File Access

- Your program can access files either in *sequential* manner or *random* manner. The access mode of a file determines how one can read, write, change, add and delete data from a file.
- A sequential file has to be accessed in the same order as the file was written. This is analogues to cassette tapes: you play music in the same order as it was recorded.
- Unlike the sequential files, you can have random-access to files in any order you want. Think of data in a random-access file as being similar to songs on compact disc (CD): you can go directly to any song you want to play without having to play or fast-forward through the other songs.

7.4.1: Sequential File Concepts

- You can perform three operations on sequential disk files. You can *create* disk files, *add* to disk files, and *read* from disk files.

7.4.1.1: Opening and Closing Sequential Files

- When you open a disk file, you only have to inform C++, the file name and what you want to do with it. C++ and your operating system work together to make sure that the disk is ready, and they create an entry in your file directory for the filename (if you are creating a file). When you

close a file, C++ writes any remaining data to the file, releases the file from the program, and updates the file directory to reflect the file's new size.

- You can use either of the two methods to open a file in C++:
  - using a ***Constructor*** or
  - using the ***open function***
- The following C++ statement will create an object fout of ofstream class and this object will be associated with file name "hello.txt".

  Ofstream fout ("hello.txt");

- This statement uses the constructor method.
- The following C++ statement will create an object fout of ofstream class and this object will be associated with file name "hello.txt".

  ofstream fout;
  fout.open("hello.txt");

- If you open a file for writing (out access mode), C++ creates the file. If a file by that name already exists, C++ *overwrite* the old file with no warning. You must be careful when opening files not to overwrite existing data that you want.
- If an error occurs during opening of a file, C++ does not create a valid file pointer (file object). Instead, C++ creates a file pointer (object) equal to zero. For example if you open a file for output, but use an invalid disk name, C++ can't open the file and therefore makes the file object equal to zero.
- You can also determine the file access mode when creating a file in C++. If you want to use the open function to open a file then the syntax is:

  fileobject.open(filename,accessmode);
  - File name is a string containing a valid file name for your computer.
  - Accessmode is the sought operation to be taken on the file and must be one of the values in the following table.

Mode	**Description**
app	Opens file for appending
ate	Seeks to the end of file while opening the file
in	Opens the file for reading
out	Opens the file for writing
binary	Opens the file in binary mode

- You should always check for the successful opening of a file before starting file manipulation on it. You use the fail() function to do the task:

- Lets have an example here:

```
ifstream indata;
indata.open("c:\\myfile.txt",ios::in);
if(indata.fail())
{
    //error discriton here
}
```

- In this case, the open operation will fail (i.e the fail function will return true), if there is no file named myfile.txt in the directory C:\
- After you are done with your file manipulations, you should use the close function to release any resources that were consumed by the file operation. Here is an example

```
indata.close();
```

- The above close() statement will terminate the relation ship b/n the ifstream object indata and the file name "c:\myfile.txt", hence releasing any resource needed by the system.

7.4.1.2: Writing to a sequential File

- The most common file I/O functions are
  - get() and put()
  - gets() and puts()
- You can also use the output redirection operator (<<) to write to a file.
- The following program creates a file called names.txt in C:\ and saves the name of five persons in it:

```
#include<fstream.h>
#include<stdlib.h>
ofstream fp;
void main()
{
        fp.open("c:\\names.txt",ios::out);
        if(fp.fail())
        {
                cerr<< "\nError opening file";
                getch();
                exit(1);
        }
        fp<< "Abebe Alemu"<<endl;
        fp<< "Lemelem Berhanu"<<endl;
        fp<< "Tesfaye Mulugeta"<<endl;
        fp<< "Mahlet Kebede"<<endl;
        fp<< "Assefa Bogale"<<endl;
        fp.close();
```

```
}//end main
```

Writing characters to sequential files:

- A character can be written onto a file using the *put()* function. See the following code:

```
#include<fstream.h>
#include<stdlib.h>// for exit() function
…
void main()
{
        char c;
        ofstream outfile;
        outfile.open("c:\\test.txt",ios::out);

        if(outfile.fail())
        {
                cerr<< "\nError opening test.txt";
                getch();
                exit(1);
        }


        for(int i=1;i<=15;i++)
        {
                cout<< "\nEnter a character : ";
                cin>>c;
                outfile.put(c);
        }
        output.close();
}//end main
```

- The above program reads 15 characters and stores in file test.txt.
- You can easily add data to an existing file, or create new files, by opening the file in *append access mode*.
- Files you open for append access mode (using ios::app) do *not have to exist*. If the file exists, C++ appends data to the end of the file (as is done when you open a file for write access).
- The following program adds three more names to the *names.txt* file created in the earlier program.

```
#include<fstream.h>
#include<stdlib.h>
…
void main()
```

```
{
        ofstream outdata;
        outdata.open("c:\\names.txt",ios::app);
        if(outdata.fail())
        {
                cerr<< "\nError opening names.txt";
                getch();
                exit(1);
        }
        outdata<< "Berhanu Teka"<<endl;
        outdata<< "Zelalem Assefa"<<endl;
        outdata<< "Dagim Sheferaw"<<endl;
        outdata.close();
}//end main
```

- If the file names.txt *does not exist*, C++ creates it and stores the three names to the file.
- Basically, you have to change only the open() function's access mode to turn a file-creation program into a file-appending program.

7.4.1.3: Reading from a File

- Files you open for read access (using ios::in) *must exist already*, or C++ gives you an error message. You can't read a file that does not exist. Open() returns zero if the file does not exist when you open it for read access.
- Another event happens when you read files. Eventually, you read all the data. Subsequently reading produces error because there is *no more* data to read. C++ provides a solution to the end-of-file occurrence.
- If you attempt to read a file that you have completely read the data from, C++ returns the value zero. To find the end-of-file condition, be sure to check for zero when reading information from files.
- The following code asks the user for a file name and displays the content of the file to the screen.

```
#include<fstream.h>

#include<stdlib.h>
void main()
{
  clrscr();
  char name[20],filename[15];
  ifstream indata;
  cout<<"\nEnter the file name : ";
```

```
  cin.getline(filename,15);
  indata.open(filename,ios::in);
  if(indata.fail())
  {
    cerr<<"\nError opening file : "<<filename;
    getch();
    exit(1);
  }
  while(!indata.eof())// checks for the end-of-file
  {
    indata>>name;
    cout<<name<<endl;
  }
  indata.close();
  getch();
}
```

Reading characters from a sequential file

- You can read a characters from a file using *get()* function. The following program asks for a file name and displays *each character* of the file to the screen. NB. A space among characters is considered as a character and hence, the exact replica of the file will be shown in the screen.

```
#include<fstream.h>

#include<stdlib.h>
void main()
{
  char c,filename[15];
  ifstream indata;
  cout<<"\nEnter the file name : ";
  cin.getline(filename,15);
  indata.open(filename,ios::in);
  if(indata.fail())// check id open succeeded
  {
     cerr<<"\nError opening file : "<<filename;
     getch();
     exit(1);
  }
  while(!indata.eof())// check for eof
  {
     indata.get(c);
     cout<<c;
  }
```

```
  indata.close();
  getch();
}
```

7.4.1.4: File Pointer and their Manipulators

- Each file has two pointers one is called *input pointer* and second is *output pointer*. The input pointer is called *get pointer* and the output pointer is called *put pointer*.
- When input and output operation take places, the appropriate pointer is automatically set according to mode.
- For example when we open a file in reading mode, file pointer is automatically set to start of file.
- When we open a file in append mode, the file pointer is automatically set to the end of file.
- In C++ there are some manipulators by which we can control the movement of the pointer. The available manipulators are:
    1. seekg()
    2. seekp()
    3. tellg()
    4. tellp()

1. seekg(): this moves get pointer i.e input pointer to a specified location. For eg. infile.seekg(5); move the file pointer to the byte number 5 from starting point.
2. seekp(): this move put pointer (output pointer) to a specified location for example: outfile.seekp(5);
3. tellg(): this gives the current position of get pointer (input pointer)
4. tellp(): this gives the current position of put pointer (output pointer)

```
eg.   ofstream fileout;
      fileout.open("c:\\test.txt",ios::app);
      int length = fileout.tellp();
```

- By the above statement in length, the total number byte of files are assigned. Because the file is opened in append mode that means, the file pointer is the last part of the file.
- Now lets see the seekg() function in action

```
#include<fstream.h>

#include<stdlib.h>
void main()
{
  clrscr();
```

```
    fstream fileobj;
    char ch; //holds A through Z
    //open the file in both output and input mode
    fileobj.open("c:\\alph.txt",ios::out|ios::in);
    if(fileobj.fail())
    {
       cerr<<"\nError opening alph.txt";
       getch();
       exit(1);
    }
    //now write the characters to the file
    for(ch = 'A'; ch <= 'Z'; ch++)
    {
       fileobj<<ch;
    }
    fileobj.seekg(8L,ios::beg);//skips eight letters, points to I
    fileobj>>ch;
    cout<<"\nThe 8th character is : "<<ch;
    fileobj.seekg(16L,ios::beg);//skips 16 letters, points to Q
    fileobj>>ch;
    cout<<"\nThe 16th letter is : "<<ch;
    fileobj.close();
    getch();
}
```

- To point to the end of a data file, you can use the seekg() function to position the file pointer at the last byte. This statement positions the file pointer to the last byte in the file. ***Fileobj.seekg(0L,ios::end);***
- This seekg() function literally reads "move the file pointer 0 bytes from the end of the file." The file pointer now points to the end-of-file marker, but you can seekg() backwards to find other data in the file.
- The following program is supposed to read "c:\alph.txt" file backwards, printing each character as it skips back in the file.
- Be sure that the seekg() in the program seeks two bytes backwards from the *current* position, not from the beginning or the end as the previous programs. The for loop towards the end of the program needs to perform a "skip-two-bytes-back", read-one-byte-forward" method to skip through the file backwards.

```
#include<fstream.h>

#include<stdlib.h>

void main()
```

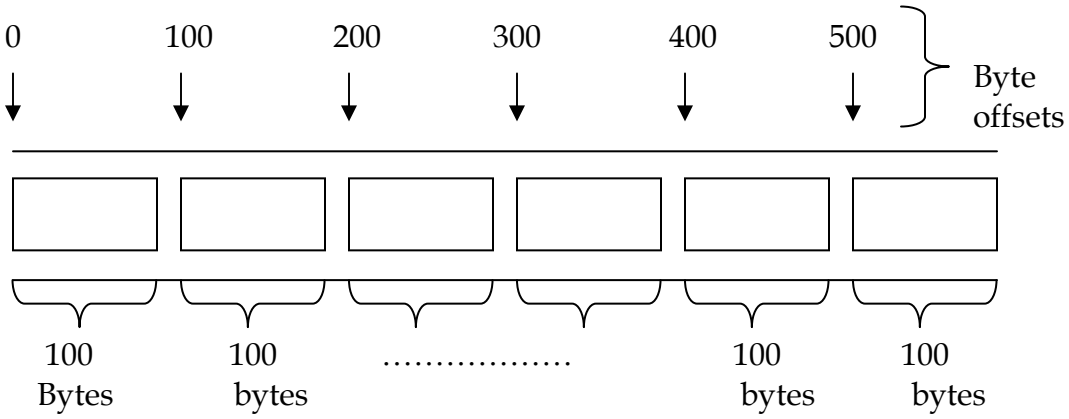
```
{
  clrscr();
  ifstream indata;
  int ctr=0;
  char inchar;

  indata.open("c:\\alph.txt",ios::in);
  if(indata.fail())
  {
     cerr<<"\nError opening alph.txt";
     getch();
     exit(1);
  }
  indata.seekg(-1L,ios::end);//points to the last byte in the file
  for(ctr=0;ctr<26;ctr++)
  {
     indata>>inchar;
     indata.seekg(-2L,ios::cur);
     cout<<inchar;
  }
  indata.close();
  getch();
}
```

Text Files And Binary Files (Comparison)

- The default access mode for file access is *text mode*. A text file is an ASCII file, compatible with most other programming languages and applications. Programs that read ASCII files can read data you create as C++ text files.
- If you specify binary access, C++ creates or reads the file in binary format. Binary data files are *"squeezed"*- that is, they take less space than text files. The disadvantage of using binary files is that other programs can't always read the data files. Only C++ programs written to access binary files can read and write them. The advantage of binary files is that you save disk space because your data files are more compact.
- The binary format is a *system-specific* file format. In other words, not all computers can read a binary file created on another computer.

7.4.2: Random Access File Concepts

- Random access enables you to read or write any data in your disk file with out having to read and write every piece of data that precedes it.
- Generally you read and write file records. A record to a file is analogus to a C++ structure. A record is a collection of one or more data values (called fields) that you read and write to disk. Generally you store data in the structures and write structures to disk.
- When you read a record from disk, you generally read that record into a structure variable and process it with your program.
- Most random access files are fixed-length records. Each record (a row in the file) takes the same amount of disk space.
- With fixed length records, your computer can better calculate where on the disk the desired record is located.



7.2.2.1 Opening Random-Access Files

- There is really no difference between sequential files and random files in C++. The difference between the files is not physical, but lies in the method that you use to access them and update them.
- The ostream member function write outputs a fixed number of bytes, beginning at a specific location in memory, to the specified stream.
- When the stream is associated with a file, function write writes the data at the location in specified by the "put" file position pointer.
- The istream member function read inputs a fixed number of bytes from the specified stream into an area in memory beginning at the specified address.
- When the stream is associated with a file, function read inputs bytes at the location in the filespecified by the "get" file poison pointer.
- Syntax of write: *fileobject.write((char*) & name ofobject, sizeof(name of object))*
- Function write expects data type const char* as its first argument. The second argument of write is an integer of type size_t specifying the number of bytes to be written.

Writing randomly to a random access file

- Here is an example that shows how to write a record to a random access file.

```
#include<fstream.h>

#include<stdlib.h>
struct stud_info{
    int id;
    char name[20];
    char fname[20];
    float CGPA;
}student;

void main()
{
  clrscr();
  char filename[15];
  ofstream outdata;
  cout<<"\nenter file name : ";
  cin>>filename;
  outdata.open(filename,ios::out);
  if(outdata.fail())
  {
     cerr<<"\nError opening "<<filename;
     getch();
     exit(1);
  }
  //stud_info student;
  //accept data here
  cout<<"\nEnter student id : ";
  cin>>student.id;
  cout<<"\nEnter student name : ";
  cin>>student.name;
  cout<<"\nEnter student father name : ";
  cin>>student.fname;
  cout<<"\nEnter student CGPA : ";
  cin>>student.CGPA;

  //now write to the file
  outdata.seekp(((student.id)-1) * sizeof(student));
  outdata.write((char*) &student, sizeof(student));
  outdata.close();
  cout<<"\nData has been saved";
```

```
    getch();
}
```

- The above code uses the combination of ostream function seekp and write to store data at exact locations in the file.
- Function seekp sets the put file-position pointer to a specific position in the file, then the write outputs the data.
- 1 is subtracted from the student id when calculating the byte location of the record. Thus, for record 1, the file position pointer is set to the byte 0 of the file.
- The istream function read inputs a specified number of bytes from the current position in the specified stream into an object.
- The syntax of read : read((char*)&name of object, sizeof(name of object));
- Function read requires a first argument of type char *. The second argument of write is an integer of type size_t specifying the number of bytes to be read.
- Here is a code that shows how to read a random access record from a file.

```
#include<fstream.h>

#include<stdlib.h>
struct stud_info{
   int studid;
   char name[20];
   char fname[20];
   float CGPA;
};
void main()
{
    clrscr();
    ifstream indata;
    char filename[15];
    cout<<"\nEnter the file name : ";
    cin>>filename;
    indata.open(filename,ios::in);
    if(indata.fail())
    {
        cerr<<"\nError opening "<<filename;
        getch();
        exit(1);
    }
    stud_info student;
    cout<<"\nEnter the id no of the student : ";
```

```
    int sid;
    cin>>sid;
    indata.seekg((sid-1) * sizeof(student));
    indata.read((char*) &student, sizeof(student));
    cout<<"\nhere is the information";
    cout<<"\nstudent id : "<<student.studid;
    cout<<"\nstudent name : "<<student.name;
    cout<<"\nstudent fname : "<<student.fname;
    cout<<"\nstudent CGPA : "<<student.CGPA;
    indata.close();
    getch();
}
```

7.5: Command Line Argument

- Command line argument means facility by which you can supply arguments to the main() function. These arguments are supplied to the program when the main function is called from the command line. Eg. c:\> file-name arg1 arg2
- Where file-name is the name of file(the program) and arg1, arg2 are arguments passed to the program.
- If we want to pass arguments to the main function, then main function should be written as follow.

  Return type main(int argc, char * argv[])
- The first argument argc represents the number of arguments in a command line. The second argument argv is an array of character type pointers that points to the command line arguments. Argc is known as argument counter and argv is called argument vector.
- C:\> student A B. the value of argc is 3 (student, A & B) and the argv would be an array of three pointers to string as follows:

  Argv[0] – points to student
  Argv[1] – points to A
  Argv[2] – points to B
- note that argv[0] always reperesents the command name that invokes the program.
- Here is a sample command line argument code

```
#include<fstream.h>

#include<stdlib.h>
void main(int argc, char *argv[])
{
   clrscr();
   char ch;
```

```
  if(argc < 3)
  {
    cerr<<"\ntoo few parameters";
    getch();
    exit(1);
  }
  else if(argc > 3)
  {
    cerr<<"\ntoo many parametes";
    getch();
    exit(1);
  }
  else//number of parameters okay
  {
        ofstream outdata;
        ifstream indata;
        outdata.open(argv[2],ios::out);
        if(outdata.fail())
        {
           cerr<<"\nlow disk space to create file : "<<argv[2];
           getch();
           exit(1);
        }
        indata.open(argv[1],ios::in);
        if(indata.fail())
        {
           cerr<<"\nfile : "<<argv[1]<<" does not exist";
           getch();
           exit(1);
        }
        //now start copying the file
        while(!indata.eof())
        {
          indata.get(ch);
          outdata.put(ch);
        }
       indata.close();
       outdata.close();
       cout<<"\nfinished copying";
  }
}
```