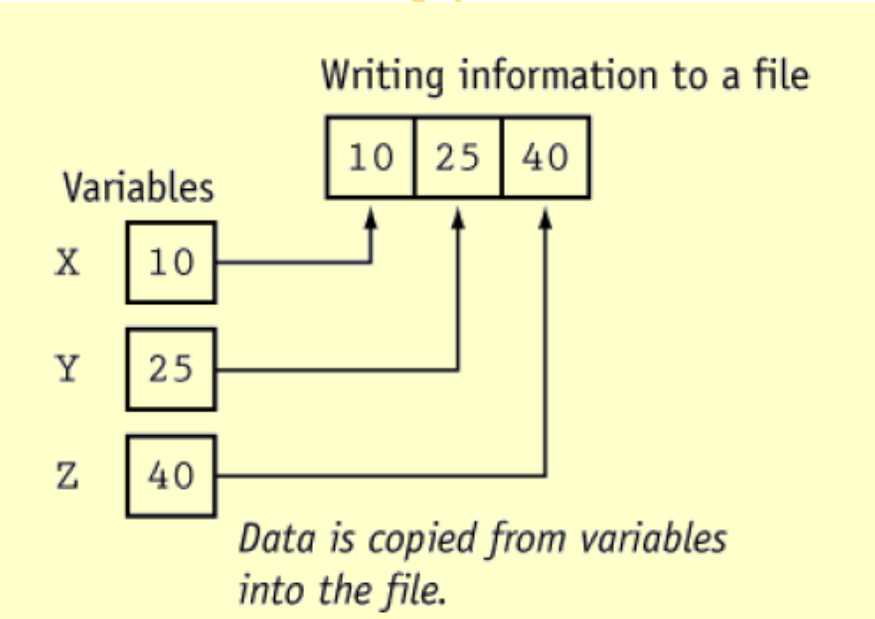
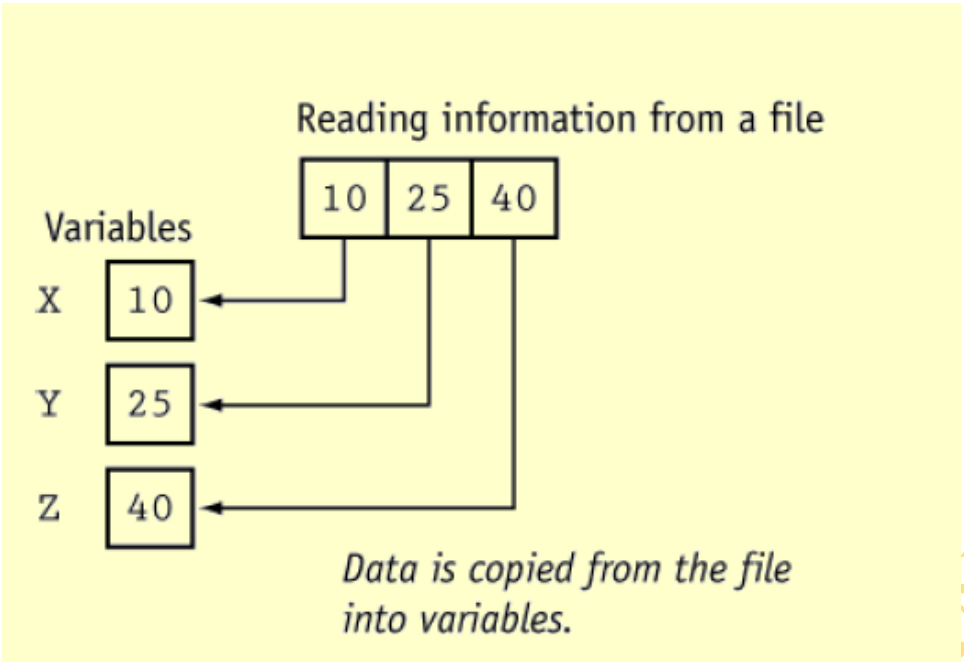
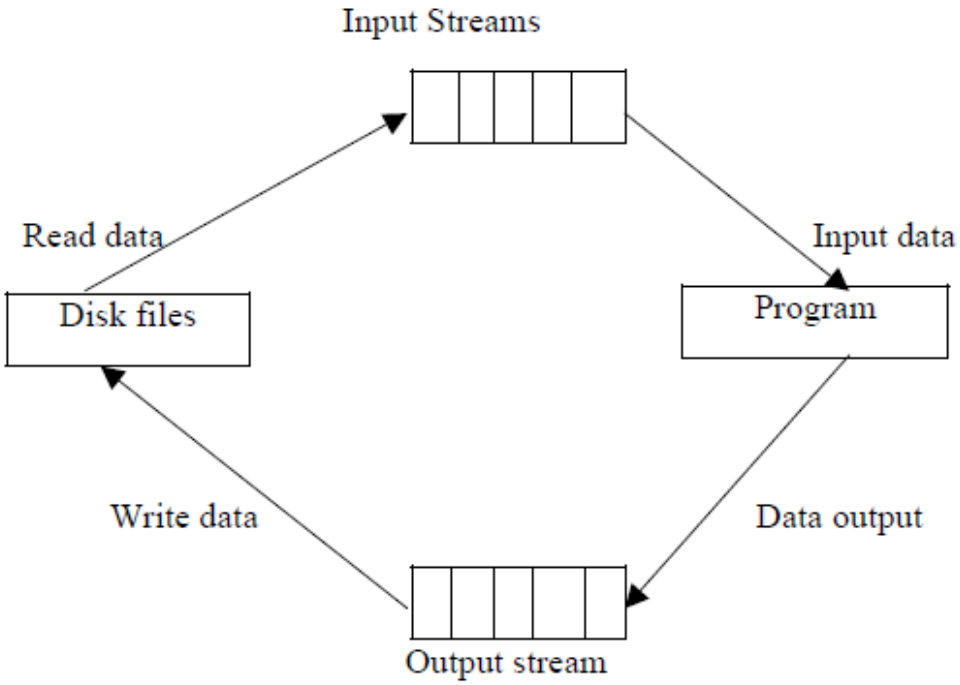
Files and Streams

Introduction :

- When a large amount of data is to be handled in such situations floppy disk or hard disk are needed to store the data .
- The data is stored in these devices using the concept of files.
- A file is a collection of related data stored in a particular area on a disk.
- Programs can be designed to perform the read and write operations on these files .
- All files are assigned a name that is used for identification purposes by the operating system and the user.
- Focus on Software Engineering: The Process of Using a File
    - Using a file in a program is a simple three-step process
    - The file must be opened. If the file does not yet exits, opening it means creating it.
    - Information is then saved to the file, read from the file, or both.
    - When the program is finished using the file, the file must be closed.
- The I/O system of C++ handles file operations which are very much similar to the console input and output operations.
- It uses file streams as an interface between the programs and files.
- The stream that supplies data to the program is called input stream and the one that receives data from the program is called output stream.
- In other words input stream extracts data from the file and output stream inserts data to the file.
- The input operation involves the creation of an input stream and linking it with the program and input file.
- Similarly, the output operation involves establishing an output stream with the necessary links with the program and output file.

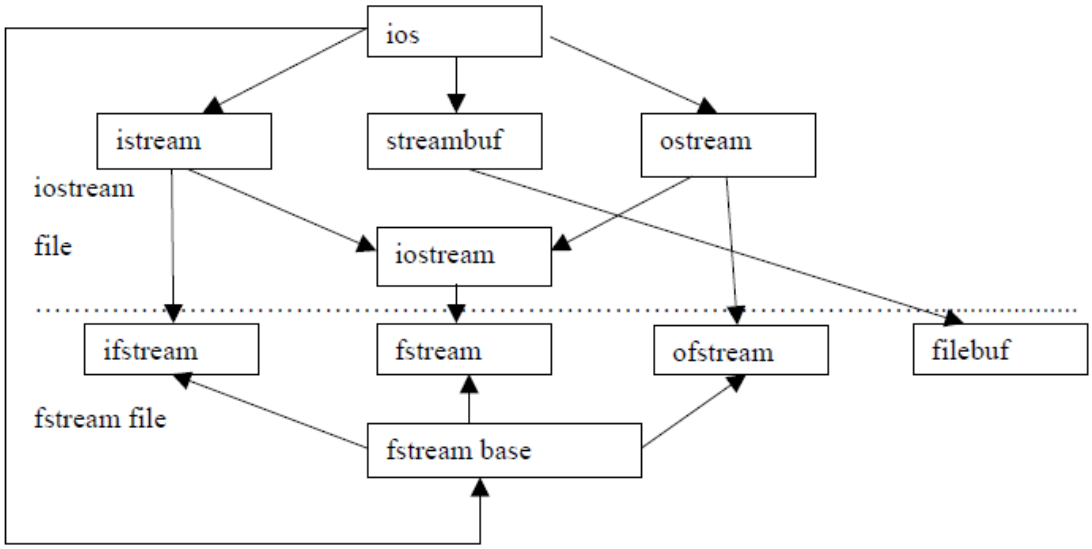






File Stream Classes:-

The I/O system of C++ contains a set of classes that defines the file handling methods. These include ifstream, ofstream and fstream.These classes are derived from fstreambase and form the corresponding iostream class. These classes ,designed to manage the disk files are declared in fstream and therefore this file is included in any program that uses files.



Steps of File Operations:-

For using a disk file the following things are necessary

1. Suitable name of file
2. Data type and structure
3. Purpose
4. Opening Method

Class	Contents
filebuf	Its purpose is to set the file buffers to read and write. Contains **Openprot** constant used in the **open()** of file stream classes.Also contain **close()** and **open()** as members.
fstreambase	Provides operations common to file streams.Serves as a base for **fstream,ifstream** and **ofstream** class.Contains **open()** and **close()** functions.
ifstream	Provides input operations.Contains **open()** with default input mode.Inherits the functions **get(),getline(),read(),seekg(),tellg()** functions from istream.
ofstream	Provides output operations.Contains **open()** with default output mode.Inherits **put(),seekp(),tellp()** and **write()** functions from **ostream**.
fstream	Provides support for simultaneous input and output operations.Contains open with default input mode.Inherits all the functions from **istream** and **ostream** classes through **iostream**.

- The filename is a string of characters that makeup a valid filename for the operating system.
- It may contain two parts ,primary name and optional period with extension.
- Examples are Input.data, Test.doc etc.
- For opening a file firstly a file stream is created and then it is linked to the filename.
- A file stream can be defined using the classes ifstream**, ofstream** and fstream that contained in the header file **fstream**.
- The class to be used depends upon the purpose whether the write data or read data operation is to be performed on the file.
- A file can be opened in two ways:

(a) Using the constructor function of class.

(b) Using the member function **open()** of the class.

- The first method is useful only when one file is used in the stream.
- The second method is used when multiple files are to be managed using one stream.
- Before data can be written to or read from a file, the file must be opened.

```
ifstream inputFile;
inputFile.open("customer.dat");
```

```
// This program demonstrates the declaration of an fstream
// object and the opening of a file.
#include <iostream.h>
#include <fstream.h>
void main(void)
{
fstream dataFile; // Declare file stream object
char fileName[81];
cout << "Enter the name of a file you wish to open\n";
cout << "or create: ";
cin.getline(fileName, 81);
dataFile.open(fileName, ios::out);
cout << "The file " << fileName << " was opened.\n";
        }
```

- Opening a File at Declaration : fstream dataFile("names.dat", ios::in | ios::out);

```
// This program demonstrates the opening of a file at the
// time the file stream object is declared.
#include <iostream.h>
#include <fstream.h>
void main(void)
{
fstream dataFile("names.dat", ios::in | ios::out);
cout << "The file names.dat was opened.\n";
}
```

Opening Files using Constructor:

- While using constructor for opening files,filename is used to initialize the file stream object.
- This involves the following steps

(i) Create a file stream object to manage the stream using the appropriate class i.e the class ofstream is used to create the output stream and the class **ifstream** to create the input stream.

(ii) Initialize the file object using desired file name.

- For example, the following statement opens a file named "results" for output:

ofstream outfile("results"); //output only

- This create outfile as an **ofstream** object that manages the output stream.
- Similarly ,the following statement declares infile as an **ifstream** object and attaches it to the file data for reading (input).

ifstream infile ("data"); //input only

- The same file name can be used for both reading and writing data.
- For example

ofstream outfile (“salary”); //creates outfile and connects salary to it
……
ifstream infile (“salary”); //creates infile and connects salary to it
…….

- The connection with a file is closed automatically when the stream object expires i.e when a program terminates.
- In the above statement ,when the program is terminated, the salary file is disconnected from the outfile stream.
- The same thing happens when second program terminates.
- Instead of using two programs,one for writing data and another for reading data ,a single program can be used to do both operations on a file.

  outfile.close(); //disconnect salary from outfile and connect to infile
  ifstream infile (“salary”);
  infile.close();
- The following program uses a single file for both reading and writing the data .
- First it take data from the keyboard and writes it to file.
- After the writing is completed the file is closed.
- The program again opens the same file read the information already written to it and displays the same on the screen.

Opening Files using open()

- The function **open()** can be used to open multiple files that uses the same stream object.
- For example to process a set of files sequentially, in such case a single stream object can be created and can be used to open each file in turn.
- This can be done as follows;

  File-stream-class stream-object;
  stream-object.**open** (“filename”);
- The following example shows how to work simultaneously with multiple files

Testing for Open Errors:

```
dataFile.open(“cust.dat”, ios::in);
if (!dataFile)
{
cout << “Error opening file.\n”;
}
```

Another way to Test for Open Errors:

```
dataFile.open(“cust.dat”, ios::in);
if (dataFile.fail())
{
cout << “Error opening file.\n”;
}
```

Closing a File:

•A file should be closed when a program is finished using it.

```
// This program demonstrates the close function.
#include <iostream.h>
#include <fstream.h>
void main(void)
{
fstream dataFile;
```

```
dataFile.open("testfile.txt", ios::out);
if (!dataFile)
{
cout << "File open error!" << endl;
return;
}
cout << "File was created successfully.\n";
cout << "Now closing the file.\n";
dataFile.close();
}
```

Using << to Write Information to a File

•The stream insertion operator (<<) may be used to write information to a file.

```
outputFile << "I love C++ programming !"
// This program uses the << operator to write information to a file.
#include <iostream.h>
#include <fstream.h>
void main(void)
{
fstream dataFile;
char line[81];
dataFile.open("demofile.txt", ios::out);
if (!dataFile)
{
cout << "File open error!" << endl;
return;
}
cout << "File opened successfully.\n";
cout << "Now writing information to the file.\n";
dataFile << "Jones\n";
dataFile << "Smith\n";
dataFile << "Willis\n";
dataFile << "Davis\n";
dataFile.close();
cout << "Done.\n";
}
```

ANOTHER PROGRM

```
// This program writes information to a file, closes the file,
// then reopens it and appends more information.
#include <iostream.h>
#include <fstream.h>
void main(void)
{
fstream dataFile;
dataFile.open("demofile.txt", ios::out);
dataFile << "Jones\n";
dataFile << "Smith\n";
dataFile.close();
dataFile.open("demofile.txt", ios::app);
dataFile << "Willis\n";
dataFile << "Davis\n";
dataFile.close();
```

```
}
```

File Output Formatting

•File output may be formatted the same way as screen output.

```
// This program uses the precision member function of a
// file stream object to format file output.
#include <iostream.h>
#include <fstream.h>
void main(void)
{
fstream dataFile;
float num = 123.456;
dataFile.open("numfile.txt", ios::out);
if (!dataFile)
{
cout << "File open error!" << endl;
return;
}
dataFile << num << endl;
dataFile.precision(5);
dataFile << num << endl;
dataFile.precision(4);
dataFile << num << endl;
dataFile.precision(3);
dataFile << num << endl;
}
```

MORE PROGRAM

```
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
void main(void)
{ fstream outFile("table.txt", ios::out);
int nums[3][3] = { 2897, 5, 837,
34, 7, 1623,
390, 3456, 12 };
// Write the three rows of numbers
for (int row = 0; row < 3; row++)
{
for (int col = 0; col < 3; col++)
{
outFile << setw(4) << nums[row][col] << " ";
}
outFile << endl;
}
outFile.close();
}
```

Using >> to Read Information from a File

•The stream extraction operator (>>) may be used to read information from a file.

```
// This program uses the >> operator to read information from a file.
#include <iostream.h>
#include <fstream.h>
void main(void)
```

```
{
fstream dataFile;
char name[81];
dataFile.open("demofile.txt", ios::in);
if (!dataFile)
{
cout << "File open error!" << endl;
return;
}
cout << "File opened successfully.\n";
cout << "Now reading information from the file.\n\n";
for (int count = 0; count < 4; count++)
{
dataFile >> name;
cout << name << endl;
}
dataFile.close();
cout << "\nDone.\n";
}
```

Detecting the End of a File

The eof() member function reports when the end of a file has been encountered.

```
if (inFile.eof())
inFile.close();
// This program uses the file stream object's eof() member
// function to detect the end of the file.
#include <iostream.h>
#include <fstream.h>
void main(void)
{
fstream dataFile;
char name[81];
dataFile.open("demofile.txt", ios::in);
if (!dataFile)
{
cout << "File open error!" << endl;
return;
}
cout << "File opened successfully.\n";
cout << "Now reading information from the file.\n\n";
dataFile >> name; // Read first name from the file
while (!dataFile.eof())
{
cout << name << endl;
dataFile >> name;
}
dataFile.close();
cout << "\nDone.\n";
}
```

- In C++, “end of file” doesn‟t mean the program is at the last piece of information in the file, but beyond it. The eof() function returns true when there is no more information to be read.

WORKING WITH SINGLE FILE

```
//Creating files with constructor function
#include <iostream.h>
#include <fstream.h>
int main()
{
ofstream outf("ITEM");
cout <<"enter item name: ";
char name[30];
cin >>name;
outf <<name <<"\n";
cout <<"enter item cost :";
float cost;
cin >>cost;
outf <<cost <<"\n";
outf.close();
ifstream inf("item");
inf >>name;
inf >>cost;
cout <<"\n";
cout <<"item name : " << name <<"\n";
cout <<"item cost: " << cost <<"\n";
inf.close();
return 0;
}
```

WORKING WITH MULTIPLE FILES

```
//Creating files with open() function
#include <iostream.h>
#include<fstream.h>
int main() {
ofstream fout;
fout.open("country");
fout<<"United states of America \n";
fout<<"United Kingdom";
fout<<"South korea";
fout.close();
fout.open("capital");
fout<<"Washington\n";
fout<<"London\n";
fout<<"Seoul \n";
fout.close();
const int N =80;
char line[N];
ifstream fin;
fin.open("country");
cout<<"contents of country file \n";
while (fin)
{
fin.getline(line,N);
cout<<line;
}
```

```
fin.close();
fin.open("capital");
cout<<"contents of capital file";
while(fin)
{
fin.getline(line,N);
cout<<line;
}
fin.close();
return 0;
}
```

Finding End of File:

- While reading a data from a file,it is necessary to find where the file ends i.e end of file.
- The programmer cannot predict the end of file,if the program does not detect end of file,the program drops in an infinite loop.
- To avoid this,it is necessary to provide correct instruction to the program that detects the end of file.Thus when end of file of file is detected,the process of reading data can be easily terminated.
- An **ifstream** object such as **fin** returns a value of 0 if any error occurs in the file operation including the end-of –file condition.
- Thus the while loop terminates when **fin** returns a value of zero on reaching the end-of –file condition.There is an another approach to detect the end of file condition.
- The statement

```
if(fin1.eof() !=0 )

{
        exit(1);
}
```

returns a non zero value if end of file condition is encountered and zero otherwise.

- Therefore the above statement terminates the program on reaching the end of file.

File Opening Modes:

- The **ifstream** and **ofstream** constructors and the function **open()** are used to open the files.
- Upto now a single arguments a single argument is used that is filename.
- However,these functions can take two arguments, the second one for specifying the file mode.
- The general form of function open() with two arguments is:

stream-object.open("filename",mode);

- The second argument mode specifies the purpose for which the file is opened.
- The prototype of these class member functions contain default values for second argument and therefore they use the default values in the absence of actual values.
- The default values are as follows :

ios::in for **ifstream** functions meaning open for reading only.
ios::out for **ofstream** functions meaning open for writing only.

- The file mode parameter can take one of such constants defined in class **ios**.
- The following table lists the file mode parameter and their meanings.

File Mode Operation **Parameter**	**Meaning**
ios::app	Append to end-of-file
ios::ate	Go to end-of-file on opening
ios::binary	Binary file
ios::in	Open file for reading only
ios::nocreate	Open fails if file the file does not exist
ios::noreplace	Open fails if the file already exists
ios::out	Open file for writing only
ios::trunk	Delete the contents of the file if it exists

File Pointers and Manipulators:

- Each file has two pointers known as file pointers,one is called the input pointer and the other is called output pointer.
- The input pointer is used for reading the contents of of a given file location and the output pointer is used for writing to a given file location.
- Each time an input or output operation takes place,the appropriate pointer is automatically advanced.

Default actions:

- When a file is opened in read-only mode,the input pointer is automatically set at the beginning so that file can be read from the start.
- Similarly when a file is opened in write-only mode the existing contents are deleted and the output pointer is set at the beginning.
- This enables us to write the file from start.In case an existing file is to be opened in order to add more data,the file is opened in 'append' mode.
- This moves the pointer to the end of file.

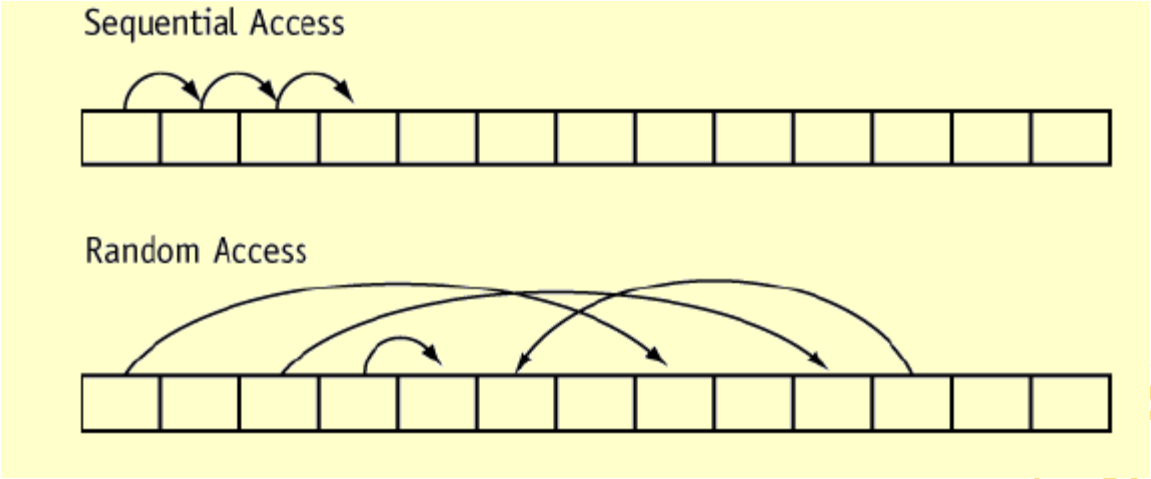
Functions for Manipulations of File pointer:

- All the actions on the file pointers takes place by default.
- For controlling the movement of file pointers file stream classes support the following functions
  (i) **seekg()** Moves get pointer (input)to a specified location.
  (ii) **seekp()** Moves put pointer (output) to a specified location.
  (iii) **tellg()** Give the current position of the get pointer.
  (iv) **tellp()** Give the current position of the put pointer.

- For example, the statement **infile.seekg(10);** moves the pointer to the byte number 10.The bytes in a file are numbered beginning from zero.
- Therefore ,the pointer to the 11th byte in the file.Consider the following statements:

```
ofstream fileout;
fileout.open("hello",ios::app);
int p=fileout.tellp();
```

- On execution of these statements, the output pointer is moved to the end of file "hello"
- And the value of p will represent the number of bytes in the file.



Specifying the Offset:

- 'Seek' functions **seekg()** and **seekp()** can also be used with two arguments as follows:

  seekg (offset,refposition);
  seekp (offset,refposition);

- The parameter offset represents the number of bytes the file pointer is to be moved from the location specified by the parameter refposition.
- The refposition takes one of the following three constants defined in the ios class:

  - **ios::beg** Start of file
  - **ios::cur** Current position of the pointer
  - **ios::end** End of file

- The **seekg()** function moves the associated file's 'get' pointer while the **seekp()** function moves the associated file's 'put 'pointer.
- The following table shows some sample pointer offset calls and their actions.
- **fout** is an **ofstream** object.

Pointer offset calls **Seek call**	**Action**
fout.seekg(o,ios::beg)	Go to start
fout.seekg(o,ios::cur)	Stay at the current position
fout.seekg(o,ios::end)	Go to the end of file
fout.seekg(m,ios::beg)	Move to (m+1)th byte in the file
fout.seekg(m,ios::cur)	Go forward by m byte from current position
fout.seekg(-m,ios::cur)	Go backward by m bytes from current position.
fout.seekg(-m,ios::end)	Go backward by m bytes from the end

```
// This program demonstrates the seekg function.
#include <iostream.h>
#include <fstream.h>
void main(void)
{
fstream file("letters.txt", ios::in);
char ch;
file.seekg(5L, ios::beg);
file.get(ch);
cout << "Byte 5 from beginning: " << ch << endl;
```

```
file.seekg(-10L, ios::end);
file.get(ch);
cout << "Byte 10 from end: " << ch << endl;
file.seekg(3L, ios::cur);
file.get(ch);
cout << "Byte 3 from current: " << ch << endl;
file.close();
}
```

Member Functions for Reading and Writing Files

- File stream objects have member functions for more specialized file reading and writing.
- The file stream classes support a number of member functions for performing the input and output operations on files.
- One pairs of functions,**put()** and **get()** are designed for handling a single character at a time.
- Another pair of functions, **write(),read()** are designed to write and read blocks of binary data.

```
// This program uses the file stream object's eof() member
// function to detect the end of the file.
#include <iostream.h>
#include <fstream.h>
void main(void)
{
fstream nameFile;
char input[81];
nameFile.open("murphy.txt", ios::in);
if (!nameFile)
{
cout << "File open error!" << endl;
return;
}
nameFile >> input;
while (!nameFile.eof())
{ cout << input;
nameFile >> input;
}
nameFile.close();
}
```

getline() function :dataFile.getline(str, 81, „\n“);

- str – This is the name of a character array, or a pointer to a section of memory.
- The information read from the file will be stored here.
- 81 – This number is one greater than the maximum number of characters to be read.
- In this example, a maximum of 80 characters will be read.
- “\n” – This is a delimiter character of your choice.
- If this delimiter is encountered, it will cause the function to stop reading before it has read the maximum number of characters.
- This argument is optional. If it‘s left our, „\n“ is the default.

```
// This program uses the file stream object's getline member
// function to read a line of information from the file.
#include <iostream.h>
```

```
#include <fstream.h>
void main(void)
{
fstream nameFile;
char input[81];
nameFile.open("murphy.txt", ios::in);
if (!nameFile)
{
cout << "File open error!" << endl;
return;
}
nameFile.getline(input, 81); // use \n as a delimiter
while (!nameFile.eof())
{
cout << input << endl;
nameFile.getline(input, 81); // use \n as a delimiter
}
nameFile.close();
}
```

PROGRAM WITH USER SPECIFIED DELIMITER

```
// This file shows the getline function with a user- specified delimiter.
#include <iostream.h>
#include <fstream.h>
void main(void)
{
fstream dataFile("names2.txt", ios::in);
char input[81]; dataFile.getline(input, 81, '$');
while (!dataFile.eof())
{
cout << input << endl;
dataFile.getline(input, 81, '$');
}
dataFile.close();
}
```

put() and get() Functions:

- The function **put()** writes a single character to the associated stream.
- Similarly,the function **get()** reads a single character from the associated stream.
- The following program illustrates how the functions work on a file.
- The program requests for a string.On receiving the string,the program writes it,character by character,to the file using the **put()** function in a for loop.
- The length of string is used to terminate the for loop.
- The program then displays the contents of file on the screen.
- It uses the function **get()** to fetch a character from the file and continues to do so until the end –of –file condition is reached.
- The character read from the files is displayed on screen using the operator <<.

PROGRAM

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

```
int main() {
char string[80];
cout<<"enter a string \n";
cin>>string;
int len =strlen(string);
fstream file;
file.open("TEXT". Ios::in | ios::out);
for (int i=o;i<len;i++)
file.put(string[i]);
file .seekg(0);
char ch;
while(file)
{
file.get(ch);
cout<<ch;
}
return 0;
}
```

Working with Multiple Files: It‟s possible to have more than one file open at once in a program

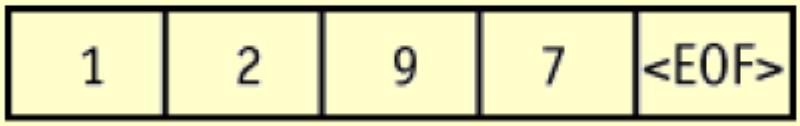
```
// This program demonstrates reading from one file and writing
// to a second file.
#include <iostream.h>
#include <fstream.h>
#include <ctype.h> // Needed for the toupper function
void main(void)
{ ifstream inFile;
ofstream outFile("out.txt");
char fileName[81], ch, ch2;
cout << "Enter a file name: ";
cin >> fileName;
inFile.open(fileName);
if (!inFile)
{ cout << "Cannot open " << fileName << endl;
return;
}
inFile.get(ch); // Get a characer from file 1
while (!inFile.eof()) // Test for end of file
{
ch2 = toupper(ch); // Convert to uppercase
outFile.put(ch2); // Write to file2
inFile.get(ch); // Get another character from file 1
}
inFile.close();
outFile.close();
cout << "File conversion done.\n";
}
```

Binary Files

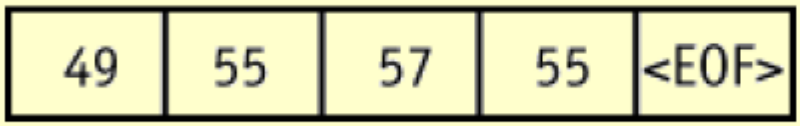
•Binary files contain data that is unformatted, and not necessarily stored as ASCII text.

```
file.open("stuff.dat", ios::out | ios::binary);
```

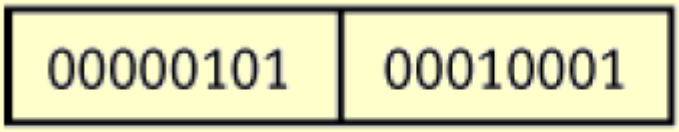
1297 expressed in Character format



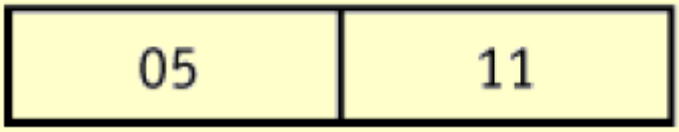
1297 expressed in ASCII



1297 as an integer, in binary



1297 as an integer, in hexadecimal




```
// This program uses the write and read functions.
#include <iostream.h>
#include <fstream.h>
void main(void)
{ fstream file("NUMS.DAT", ios::out | ios::binary);
int buffer[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
cout << "Now writing the data to the file.\n";
file.write((char*)buffer, sizeof(buffer));
file.close();
file.open("NUMS.DAT", ios::in); // Reopen the file.
cout << "Now reading the data back into memory.\n";
file.read((char*)buffer, sizeof(buffer));
for (int count = 0; count < 10; count++)
cout << buffer[count] << " ";
file.close();
}
```

Creating Records with Structures : Structures may be used to store fixed-length records to a file.

```
struct Info
{
```

```
char name[51];
int age;
char address1[51];
char address2[51];
char phone[14];
};
```

Since structures can contain a mixture of data types, you should always use the ios::binary mode when opening a file to store them.

```
// This program demonstrates the use of a structure variable to
// store a record of information to a file.
#include <iostream.h>
#include <fstream.h>
#include <ctype.h> // for toupper
// Declare a structure for the record.
struct Info
{
char name[51];
int age;
char address1[51];
char address2[51];
char phone[14];
};
void main(void)
{
fstream people("people.dat", ios::out | ios::binary);
Info person;
char again;
if (!people)
{
cout << "Error opening file. Program aborting.\n";
return;
}
do
{
cout << "Enter the following information about a "
<< "person:\n";
cout << "Name: ";
cin.getline(person.name, 51);
cout << "Age: ";
cin >> person.age;
cout << "Address line 1: ";
cin.getline(person.address1, 51);
cout << "Address line 2: ";
cin.getline(person.address2, 51);
cout << "Phone: ";
cin.getline(person.phone, 14);
people.write((char *)&person, sizeof(person));
cout << "Do you want to enter another record? ";
cin >> again;
cin.ignore();
```

```
} while (toupper(again) == 'Y');
people.close();
}
```

write() and read () functions:

The functions **write()** and **read(),**unlike the functions **put()** and **get()** ,handle the data in binary form.

This means that the values are stored in the disk file in same format in which they are stored in the internal memory.

An int character takes two bytes to store its value in the binary form,irrespective of its size.

But a 4 digit int will take four bytes to store it in the character form.

The binary input and output functions takes the following form:

infile.read ((char *) & V,sizeof (V));
outfile.write ((char *) & V ,sizeof (V));

- This function takes two arguments, first is the variable V
- And second is size of variable V in bytes.
- The address of the variable must be type cast to char*.

PROGRAM

```
// I/O OPERATIONS ON BINARY FILES
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
const char * filename =”Binary”;
int main()
{
float height[4] ={ 175.5,153.0,167.25,160.70};
ofstream outfile;
outfile.open(filename);
outfile.write((char *) & height,sizeof(height));
outfile.close();
for (int i=0;i<4;i++)
height[i]=0;
ifstream infile;
infile.open(filename);
infile.read ((char *) & height,sizeof (height));
for (i=0;i<4;i++)
{
cout.setf(ios::showpoint);
cout<<setw(10)<<setprecision(2)<<height[i];
}
infile.close();
return 0;
}
```

Error Handling during File Operations:

- There are many problems encounterd while dealing with files like

  • a file which we are attempting to open for reading does not exist.
  • The file name used for a new file may already exist.
  • We are attempting an invalid operation such as reading past the end of file.
  • There may not be any space in the disk for storing more data.

- We may use invalid file name.
- We may attempt to perform an operation when the file is not opened for that purpose.The C++ file stream inherits a 'stream-state 'member from the class ios.

- This member records information on the status of a file that is being currently used.
- The stream state member uses bit fields to store the status of error conditions stated above.
- The class ios support several member functions that can be used to read the status recorded in a file stream.

Bit	Description
ios::eof bit	Set when the end of an input stream isencountered.
ios::failbit	Set when an attempted operation has failed.
ios::hardfail	Set when an unrecoverable error has occurred.
ios::badbit	Set when an invalid operation has beenattempted.
ios::goodbit	Set when all the flags above are not set.Indicates the stream is in good condition.

Error Handling Functions

Function	Return value and meaning
eof()	Returns true(non zero value) if end of file is encountered while reading otherwise returns false(zero).
fail()	Returns true when an input or output operation has failed .
bad()	Returns true if an invalid operation is attempted or any unrecoverable error has occurred.However,if it is false,it may be possible to recover from any other error reported and continues operation.
good()	Returns true if no error has occurred.This means all the above functions are false.For instance,if file.good() is true.all is well with the stream file and we can proceed to perform I/O operations.When it returns false,no further operations is carried out.
clear()	When called with no arguments, clears all theflags listed above. Can also be called with aspecific flag as an argument

- These functions can be used at the appropriate places in a program to locate the status of a file stream and thereby to take the necessary corrective measures.
- Example:

```
//This program demonstrates the return value of the stream object error testing member functions.
#include <iostream.h>
#include <fstream.h>
// Function prototype
void showState(fstream &);
void main(void)
{
fstream testFile("stuff.dat", ios::out);
if (testFile.fail())
{
cout << "cannot open the file.\n";
return; }
int num = 10;
cout << "Writing to the file.\n";
```

```
testFile << num; // Write the integer to testFile
showState(testFile);
testFile.close(); // Close the file
testFile.open("stuff.dat", ios::in); // Open for input
if (testFile.fail())
{
cout << "cannot open the file.\n";
return;
}
cout << "Reading from the file.\n";
testFile >> num; // Read the only number in the file
showState(testFile);
cout << "Forcing a bad read operation.\n";
testFile >> num; // Force an invalid read operation
showState(testFile);
testFile.close(); // Close the file
}
// Definition of function ShowState. This function uses
// an fstream reference as its parameter. The return values of
// the eof(), fail(), bad(), and good() member functions are
// displayed. The clear() function is called before the function
// returns.
void showState(fstream &file)
{
cout << "File Status:\n";
cout << " eof bit: " << file.eof() << endl;
cout << " fail bit: " << file.fail() << endl;
cout << " bad bit: " << file.bad() << endl;
cout << " good bit: " << file.good() << endl;
file.clear(); // Clear any bad bits
}
```

Program Output

```
Writing to the file.
File Status:
eof bit: 0
fail bit: 0
bad bit: 0
good bit: 1
Reading from the file.
File Status:
eof bit: 0
fail bit: 0
bad bit: 0
good bit: 1
Forcing a bad read operation.
File Status:
eof bit: 1
fail bit: 2
bad bit: 0
good bit: 0
```

Opening file for both reading and writing: Following statement allows to open file for both reading and writing simultaneously.
fstream file("data.dat",ios::in|ios::out);

```
// This program sets up a file of blank inventory records.
#include <iostream.h>
#include <fstream.h>
// Declaration of Invtry structure
struct Invtry
{
char desc[31];
int qty;
float price;
};
void main(void)
{
fstream inventory("invtry.dat", ios::out | ios::binary);
Invtry record = { "", 0, 0.0 };
// Now write the blank records
for (int count = 0; count < 5; count++)
{
cout << "Now writing record " << count << endl;
inventory.write((char *)&record, sizeof(record));
}
inventory.close();
}
```

ANOTHER PROGRAM

```
// This program displays the contents of the inventory file.
#include <iostream.h>
#include <fstream.h>
// Declaration of Inventory structure
struct Invtry
{
char desc[31];
int qty;
float price;
};
void main(void)
{
fstream inventory("invtry.dat", ios::in | ios::binary);
Invtry record = { "", 0, 0.0 };
// Now read and display the records
inventory.read((char *)&record, sizeof(record));
while (!inventory.eof())
{
cout << "Description: ";
cout << record.desc << endl;
cout << "Quantity: ";
cout << record.qty << endl;
cout << "Price: ";
```

```
cout << record.price << endl << endl;
inventory.read((char *)&record, sizeof(record));
}
inventory.close();
}
```

PROGRAM

```
// This program allows the user to edit a specific record in
// the inventory file.
#include <iostream.h>
#include <fstream.h>
// Declaration of Invtry structure
struct Invtry
{
char desc[31];
int qty;
float price;
};
void main(void)
{
fstream inventory("invtry.dat", ios::in | ios::out | ios::binary);
Invtry record;
long recNum;
cout << "Which record do you want to edit?";
cin >> recNum;
inventory.seekg(recNum * sizeof(record), ios::beg);
inventory.read((char *)&record, sizeof(record));
cout << "Description: ";
cout << record.desc << endl;
cout << "Quantity: ";
cout << record.qty << endl;
cout << "Price: ";
cout << record.price << endl;
cout << "Enter the new data:\n";
cout << "Description: ";
cin.getline(record.desc, 31);
cout << "Quantity: ";
cin >> record.qty;
cout << "Price: ";
cin >> record.price;
inventory.seekp(recNum * sizeof(record), ios::beg);
inventory.write((char *)&record, sizeof(record));
inventory.close();
}
```


Accepting command line arguments in C++ using argc and argv:

- In C++ it is possible to accept command line arguments.
- Command-line arguments are given after the name of a program in command-line operating systems like DOS or Linux, and are passed in to the program from the operating system.

- To use command line arguments in your program, you must first understand the full declaration of the main function, which previously has accepted no arguments.
- In fact, main can actually accept two arguments: one argument is number of command line arguments, and the other argument is a full list of all of the command line arguments.
- The full declaration of main looks like this:
  int main (int argc, char *argv[])
- The integer, argc is the ARGument Count (hence argc).
- It is the number of arguments passed into the program from the command line, including the name of the program.
- The array of character pointers is the listing of all the arguments. argv[0] is the name of the program, or an empty string if the name is not available.
- After that, every element number less than argc is a command line argument.
- You can use each argv element just like a string, or use argv as a two dimensional array. argv[argc] is a null pointer.

```
#include <iostream>
using namespace std;
int main(int argc, char* argv[])
{
cout << "argc = " << argc << endl;
for(int i = 0; i < argc; i++)
cout << "argv[" << i << "] = " << argv[i] << endl;
return 0;
}
```

- You'll notice that argv[0] is the path and name of the program itself.
- This allows the program to discover information about itself.
- It also adds one more to the array of program arguments, so a common error when fetching command-line arguments is to grab argv[0] when you want argv[1].
- You are not forced to use argc and argv as identifiers in main( ), those identifiers are only conventions (but it will confuse people if you don't use them).
- Also, there is an alternate way to declare argv:
  int main(int argc, char** argv) { ... }
- Both forms are equivalent.