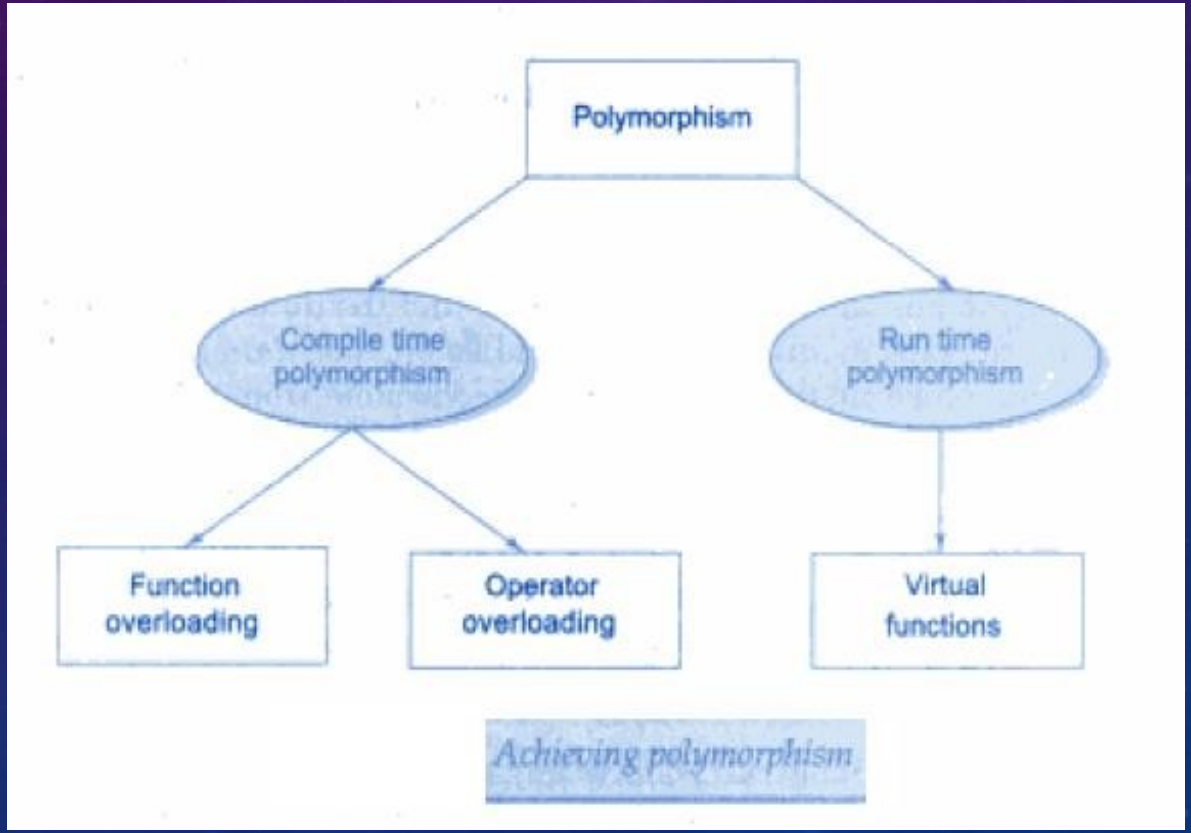
Pointers, Virtual Functions & Polymorphism

Polymorphism



Achieving polymorphism

The concept of *polymorphism* is implemented using the overloaded functions and operators. The overloaded member functions are 'selected' for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called *early binding* or *static binding* or *static linking*. Also known as *compile time polymorphism*, early binding simply means that an object is bound to its function call at compile time.

Now let us consider a situation where the function name and prototype is the same in both the base and derived classes. For example, consider the following class definitions:

```
class A
{
   int x;
  public:

  void show() {....}                    // show() in base class
};
class B: public A
{
      int y;
  public:
      void show() {....}                // show() in derived class
};
```

How do we use the member function **show()** to print the values of objects of both the classes **A** and **B**?. Since the prototype of **show()** is the same in both the places, the function is not overloaded and therefore static binding does not apply.

It would be nice if the appropriate member function could be selected while the program is running. This is known as *run time polymorphism*. How could it happen? C++ supports a mechanism known as *virtual function* to achieve run time polymorphism.

At run time, when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as *late binding*. It is also known as *dynamic binding* because the selection of the appropriate function is done dynamically at run time.

Dynamic binding is one of the powerful features of C++. This requires the use of pointers to objects.

this Pointer

C++ uses a unique keyword called "this" to represent an object that invokes a member function. 'this' is a pointer that points to the object for which this function was called. This unique pointer is called and it passes to the member function automatically. The pointer this acts as an implicit argument to all the member function,
for e.g.

```
class ABC
{
int a ;
-----
-----
};
```

The private variable ‘a’ can be used directly inside a member function, like a=123;

We can also use the following statement to do the same job.

this → a = 123

e.g.

```
class student
{
int a;
public:
void set (int a1)
{
this->a = a1;
}

void show ( )
{
cout << a<<"\n";
}
};

main ( )
{
student S1, S2;
S1.set (5) ; S2.set(100);
S1.show( );S2.show ( );
}
```

//here this pointer is used to assign a class level ‘a’ with the argument ‘a1’

o/p

5

100

Pointers to Derived Classes

Polymorphism is also accomplished using pointers in C++. It allows a pointer in a base class to point to either a base class object or to any derived class object. We can have the following Program segment show how we can assign a pointer to point to the object of the derived class.

```
class base
{
//Data Members
//Member Functions
};
class derived : public base
{
//Data Members
//Member functions
};
void main ( ) {
base *ptr; //pointer to class base
derived obj ;
ptr = &obj ; //indirect reference obj to the pointer
//Other Program statements
}
```

The pointer ptr points to an object of the derived class obj. But, a pointer to a derived class object may not point to a base class object without explicit casting.

For example, the following assignment statements are not valid

```
void main ( )
{
base obja;
derived *ptr;
ptr = &obja;          //invalid.... .explicit casting required
//Other Program statements
}
```

A derived class pointer cannot point to base class objects. But, it is possible by using explicit casting.

```
void main ( )
{
base obj ;
derived *ptr;          // pointer of the derived class
ptr = (derived *) & obj; //correct reference
//Other Program statements
}
```

Pointers to objects of a base class are type-compatible with pointers to objects of a derived class. Therefore, a single pointer variable can be made to point to objects belonging to different classes. For example, if **B** is a base class and **D** is a derived class from **B**, then a pointer declared as a pointer to **B** can also be a pointer to **D**. Consider the following declarations:

```
B *cptr;          // pointer to class B type variable
B  b;             // base object
D  d;             // derived object
cptr = &b;        // cptr points to object b
```

We can make **cptr** to point to the object **d** as follows:

```
cptr = &d;        // cptr points to object d
```

This is perfectly valid with C++ because **d** is an object derived from the class **B**.

However, there is a problem in using **cptr** to access the public members of the derived class **D**. Using **cptr**, we can access only those members which are inherited from **B** and not the members that originally belong to **D**. In case a member of **D** has the same name as one of the members of **B**, then any reference to that member by **cptr** will always access the base class member.

Although C++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. We may have to use another pointer *declared* as pointer to the derived type.

```
#include <iostream>

using namespace std;

class BC
{
  public:
     int b;
     void show()
     { cout << "b = " << b << "\n";}
};
class DC : public BC

{
  public:
     int d;
     void show()
     { cout << "b = " << b << "\n"
            << "d = " << d << "\n";
     }
};

int main()
{
      BC *bptr;                    // base pointer
      BC base;
      bptr = &base;                // base address

      bptr->b = 100;               // access BC via base pointer
      cout << "bptr points to base object \n";
      bptr -> show();
      // derived class
      DC derived;
      bptr = &derived;             // address of derived object
      bptr -> b = 200;             // access DC via base pointer

      /* bptr -> d = 300;*/        // won't work
      cout << "bptr now points to derived object \n";
      bptr -> show();              // bptr now points to derived obj

      /* accessing d using a pointer of type derived class DC */

      DC *dptr;                   // derived type pointer
      dptr = &derived;
      dptr->d = 300;

      cout << "dptr is derived type pointer\n";
      dptr -> show();

      cout << "using ((DC *)bptr)\n";
      ((DC *)bptr) -> d = 400;
      ((DC *)bptr) -> show();

      return 0;
```

```
bptr points base object
b = 100
bptr now points to derived object
b = 200
dptr is derived type pointer
b = 200
d = 300
using ((DC *)bptr)
b = 200
d = 400
```

When we use the same function name in both the base and derived classes, the function in base class is declared as *virtual* using the keyword **virtual** preceding its normal declaration. When a function is made **virtual**, **C++** determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the **virtual** function.

VIRTUAL FUNCTIONS

```
#include <iostream>

using namespace std;

class Base
{
  public:
        void display() {cout << "\n Display base ";}
        virtual void show() {cout << "\n show base";}
};
class Derived : public Base
{
  public:
        void display() {cout << "\n Display derived";}
        void show() {cout << "\n show derived";}
};

int main()
{
        Base B;
        Derived D;
        Base *bptr;

        cout << "\n bptr points to Base \n";
        bptr = &B;
        bptr -> display();   // calls Base version
        bptr -> show();      // calls Base version

        cout << "\n\n bptr points to Derived\n";
        bptr = &D;
        bptr -> display();   // calls Base version
        bptr -> show();      // calls Derived version

        return 0;
}
```

```
bptr points to Base

Display base
Show base

bptr points to Derived

Display base
Show derived
```

Rules for Virtual Functions

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.
10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

Pure Virtual Functions

A "do-nothing" function may be defined as follows:

```
virtual void display() = 0;
```

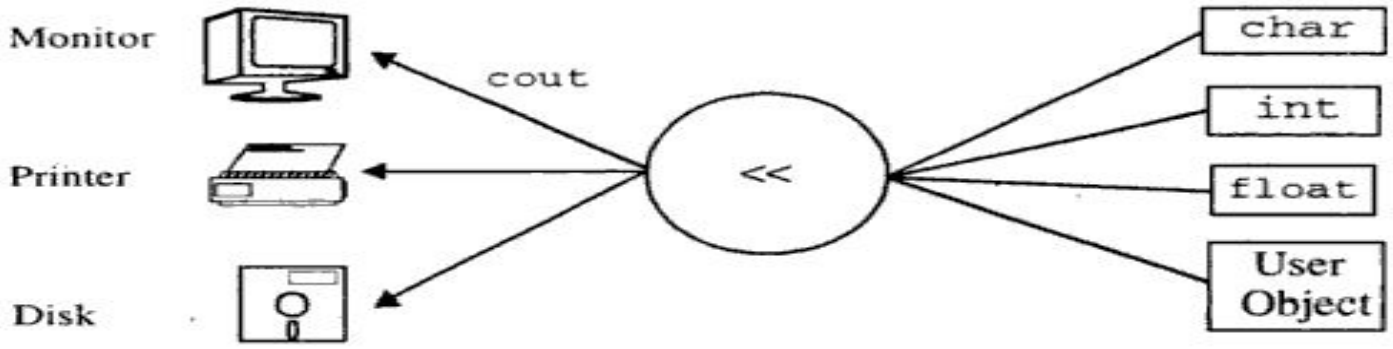
Such functions are called *pure virtual* functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function. Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. As stated earlier, such classes are called *abstract base classes*. The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

Streams & Console I/O

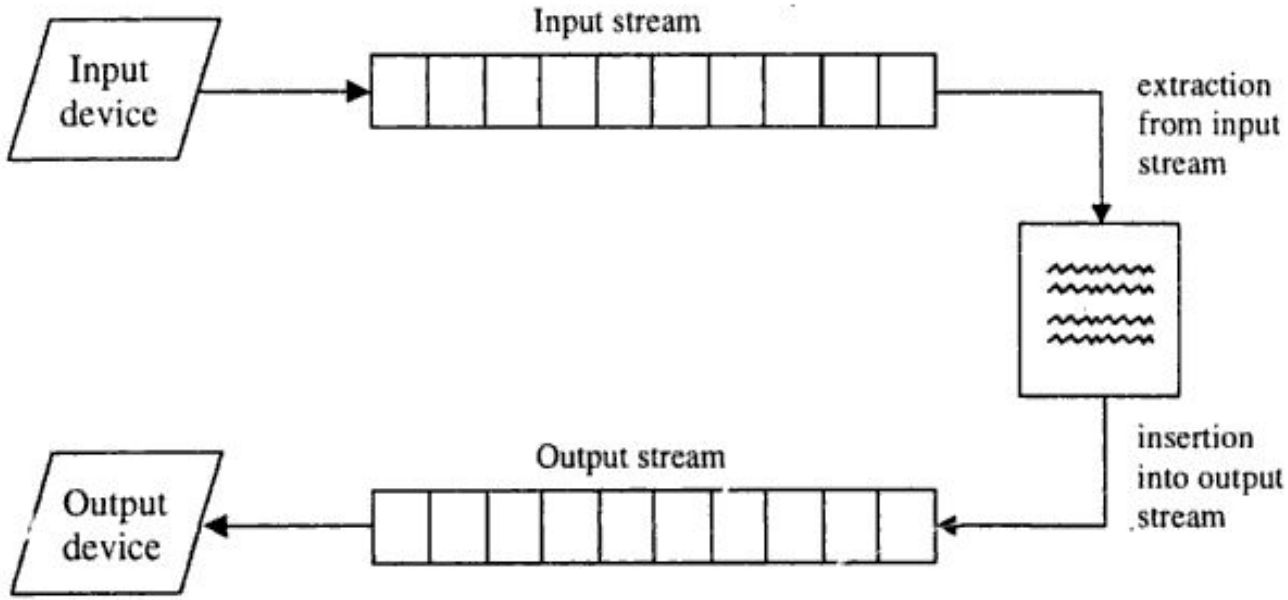
The input and output operations were performed using `cin` and `cout` with the stream operators >> and << respectively.

C++ uses the concept of streams and stream classes to perform I/O operations with console and disk files. C++ streams deal with a sequence of characters

Streams are classified into input streams and output streams. Streams resemble the *producer and consumer model*. The *producer* produces items to be consumed by the *consumer*. The producers and consumers are connected by the C++ operators >> or <<. In C++, the I/O system is designed to operate on a wide variety of devices including console, disks, printer etc. It is designed to provide a consistent and device independent interface. It allows uniform use of any I/O device—be it a disk, a terminal, or a printer as shown in Figure . The computer resources involved in the stream computation include display, keyboard, files, printer, etc. The stream is an object flowing from one place to another. For instance, in nature, a stream normally refers to the flow of water from the hills to the oceans. Similarly, in C++, a stream is used to refer to the flow of data from a particular device to the program's variables. The device here refers to files, memory arrays, keyboard, console, and so on. In C++, these streams are treated as objects to support consistent access interface.



(a) Consistent stream interface with devices



(b) Data streams

Some of the above devices exhibit the characteristics of either a producer or a consumer and others exhibit the characteristics of both the producer and consumer depending on the operations performed on them. For instance, the keyboard exhibits the nature of only a producer; printer or monitor screen exhibit the nature of only a consumer. Whereas, a file stored on the disk, can behave as a producer or consumer depending on the operation initiated on it.

A stream is a series of bytes, which act either as a source from which input data can be extracted or as a destination to which the output can be sent. The source stream provides data to the program called the input stream and the destination stream that receives data from the program is called the output stream.

C++ accomplishes input/output operations using concept of stream.
A stream is a series of bytes whose value depends on the variable in which it is stored. This way,

C++ is able to treat all the input and output operations in a uniform manner.

Thus, whether it is reading from a file or from the keyboard, for a C++ program it is simply a stream.
Objects cin and cout (pre-defined in the iostream.h file) are used for the input and output of data of various types. This has

been made possible by overloading the operators >> and << to recognize all the basic C++ types.

The >> operator is overloaded in the istream class and « is overloaded in the ostream class.
The following is the general format for reading data from the keyboard: cin >>

variable1 >> variable2 >>… …>> variableN;

Where variable1, variable2,.... are valid C++ variable names that have been declared already.

This statement will cause the computer to halt the execution and look for input data from the keyboard.
The input data for this statement would be:

data1 data2.....dataN

The input data are separated by white spaces and should match the type of variable in the cin list.

Spaces, newlines and tabs will be skipped.

The operator >> reads the data character by character and assigns it to the indicated location.

The reading for a variable will be terminated at the encounter of a white space or a character that does not match the destination type.

For example, consider the following code:
int code;
cin >> code;

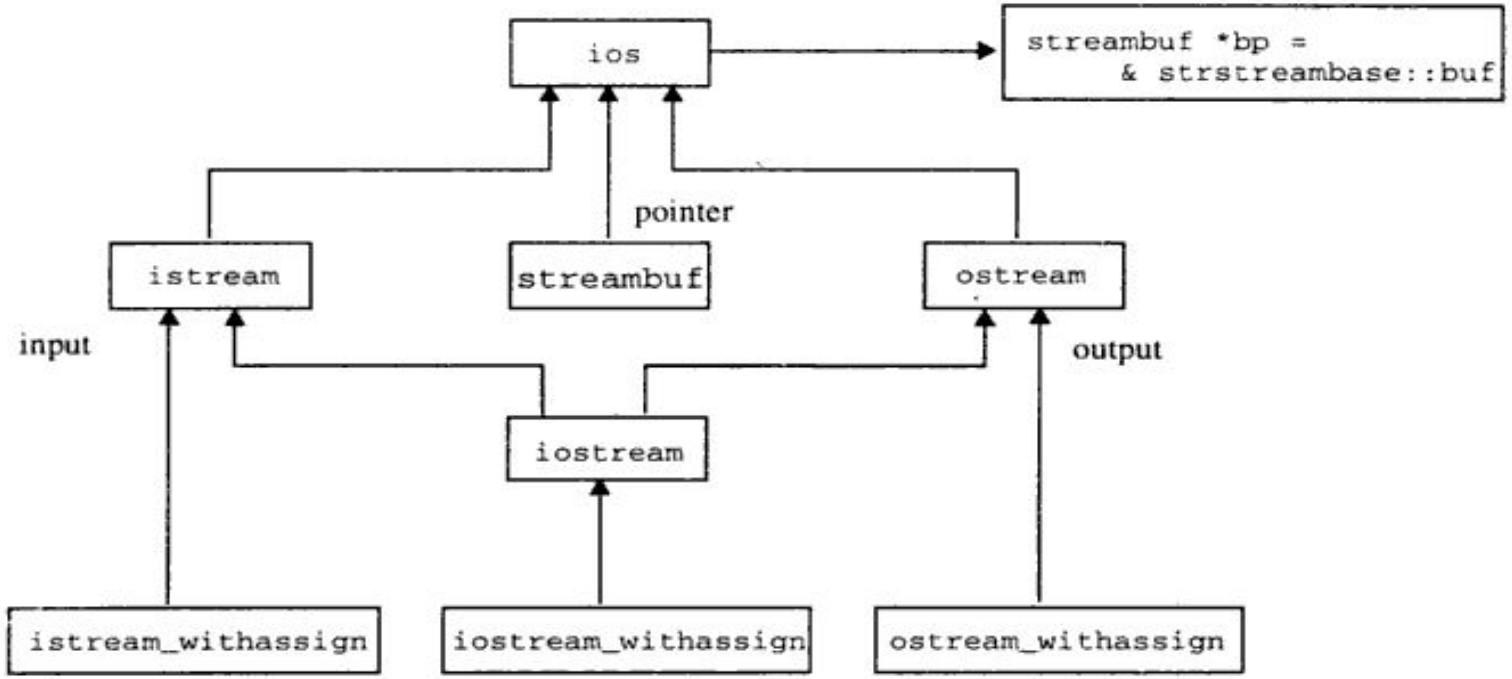
Suppose the following data is given as input:
1267E

The operator will read the characters up to 7 and the value 1267 is assigned to code. The character E remains in the input stream and will be input to the next cin statement.
The general format of outputting data:

cout << iteml <<item2 << .. ..<< itemN;
The items, item1 through itemN may be variables or constants of any basic types.

Hierarchy of Console Stream Classes

The C++ input-output system supports a hierarchy of classes that are used to manipulate both the console and disk files, called stream classes. The stream classes are implemented in a rather elaborate hierarchy. The knowledge of C++'s input and output stream class hierarchy will result in the potential utilization of stream classes. Figure depicts hierarchy of classes, which are used with the console device.



Hierarchy of console stream classes

The iostream facility of C++ provides an easy means to perform I/O. The class istream uses the predefined stream cin that can be used to read data from the standard input device. The extraction operator >>, is used to get data from a stream. The insertion operator <<, is used to output data into a stream. A stream object must appear on the left side of the << or >> operator; however, multiple stream operators can be concatenated on a single line, even when they refer to objects of different types. For instance, consider the following statements:

```
cout << item1 << "***" << c1 << my_object << 22;
cin >> int_var >> float_var >> my_object;
```

The first statement outputs objects of different types (both the standard and user defined) and the second statement reads data of different types.

The classes istream, ostream, and iostream, which are designed exclusively to manage the console device, are declared in the header file iostream.h. The actions performed by these classes related to console device management are described below:

ios class: It provides operations common to both input and output. It contains a pointer to a buffer object (streambuf). It has constants and member functions that are essential for handling formatted input and output operations.

The classes derived from the `ios` class (`istream`, `ostream`, `iostream`) perform specialized input-output operations with high-level formatting:

- `istream` (input stream) does formatted input.
- `ostream` (output stream) does formatted output.
- `iostream` (input/output stream) does formatted input and output.

The pointer `streambuf` in the `ios` class provides an abstraction for communicating to a physical device and classes derived from it deal with files, memory, etc. The class, ios communicates to a streambuf, which maintains information on the state of the streambuf (good, bad, eof, etc.), and maintains flags used by the `istream` and `ostream`.

istream class: It is a derived class of ios and hence inherits the properties of `ios`. It defines input functions such as `get()`, `getline()`, and `read()`. In addition, it has an overloaded member function, stream extraction operator >>, to read data from a standard input device to the memory items.

ostream class: It is a derived class of `ios`, and hence, inherits the properties of `ios`. It defines output functions such as `put()` and `write()`. In addition, it has an overloaded member function, stream insertion operator <<, to write data from memory items to a standard output device.

iostream class: It is derived from multiple base classes, `istream` and `ostream`, which are in turn inherited from the class ios. It provides facility for handling both input and output streams, and supports all the operations provided by `istream` and `ostream` classes.

The classes `istream_withassign`, `ostream_withassign`, and `iostream_withassign` add the assignment operators to their parent classes.

Formatted Console I/O Operations

C++ supports a number of features that could be used for formatting the output.

These features include:

- ios class functions and flags.
- Manipulators.
- User-defined output functions.

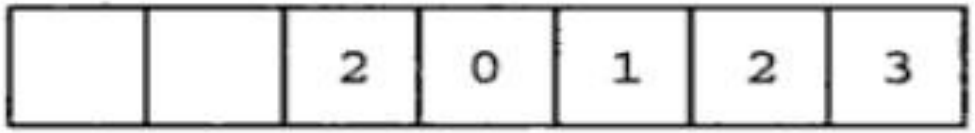
The ios class contains a large number of member functions that could be used to format the output in a number of ways. The most important ones among them are listed below.

Function	Task
width()	To specify the required field size for displaying an output value
Precision()	To specify the number of digits to be displayed after the decimal point of a float value
fill()	To specify a character that is used to fill the unused portion of a field.
self()	To specify format flags that can control the form of output display (such as Left-justification and right-justification).
Unself()	To clear the flags specified.

For instance, the statements

```
cout.width( 4 );
cout << 20 << 123;
```

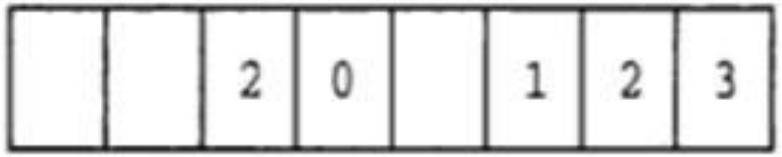
produce the following output:



The first value is printed in right-justified form in four columns. The next item is printed immediately after first item without any separation; `width(4)` is then reverted to the default value, which prints in left-justified form with default size. It can be overcome by explicitly setting width of every item with each `cout` statement as follows:

```
cout.width( 4 );
cout << 20;
cout.width( 4 );
cout << 123;
```

These statements produce the following output.



```
cout.precision( 2 );
cout << 2.23 << endl;
cout << 5.169 << endl;
cout << 3.5055 << endl;
cout << 4.003 << endl;
```

will produce the following output:

```
2.23            (perfect fit)
5.17            (rounded)
3.51            (rounded)
4               (no trailing zeros, truncated)
```

Flags value	Bit field	Effect produced
ios::left	ios::adjustfield	Left-justified output
ios::right	ios::adjustfield	Right-adjust output
ios::internal	ios::adjustfield	Padding occurs between the sign or base indicator and the number, when the number output fails to fill the full width of the field.
ios::dec	ios::basefield	Decimal conversion
ios::oct	ios::basefield	Octal conversion
ios::hex	ios::basefield	Hexadecimal conversion
ios::scientific	ios::floatfield	Use exponential floating notation
ios::fixed	ios::floatfield	Use ordinary floating notation

Flags and bit fields for setf function

Consider the following statements:

```
cout.setf( ios::left, ios::adjustfield );
cout.fill( '*' );
cout.precision( 2 );
cout.width( 6 );
cout << 12.53;
cout.width( 6 );
cout << 20.5;
cout.width( 6 );
cout << 2;
```

The output produced by the above statements is:

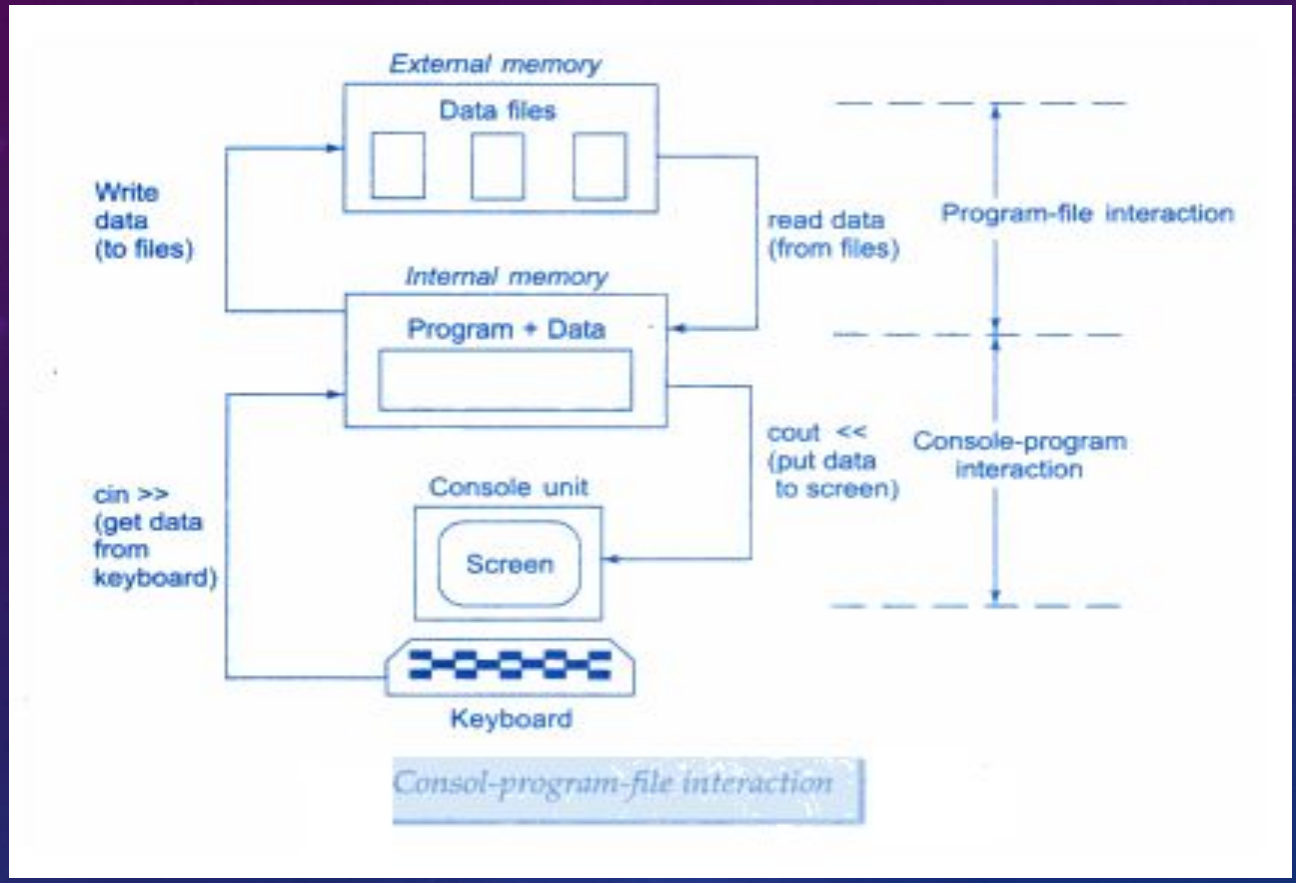
1	2	.	5	3	*	2	0	.	5	*	*	2	*	*	*	*	*

Working with Files

Many real-life problems handle large volumes of data and, in such situations, we need to use some devices such as floppy disk or hard disk 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 the disk. Programs can be designed to perform the read and write operations on these files.

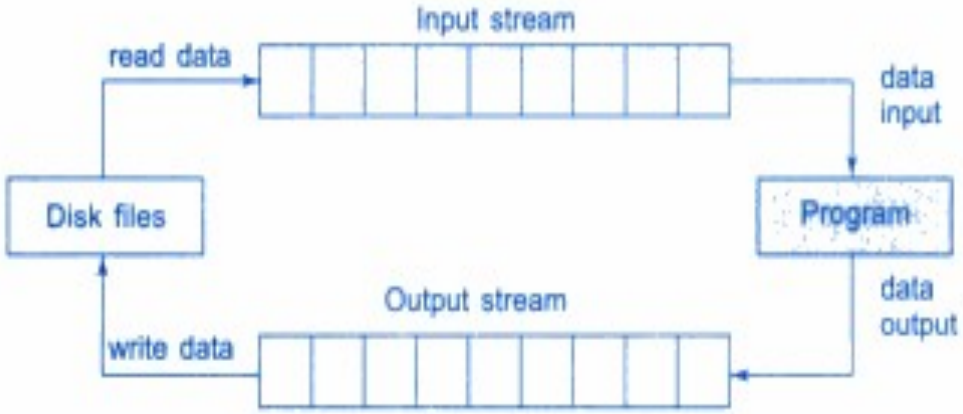
A program typically involves either or both of the following kinds of data communication:

1. Data transfer between the console unit and the program.
2. Data transfer between the program and a disk file.



Consol-program-file interaction

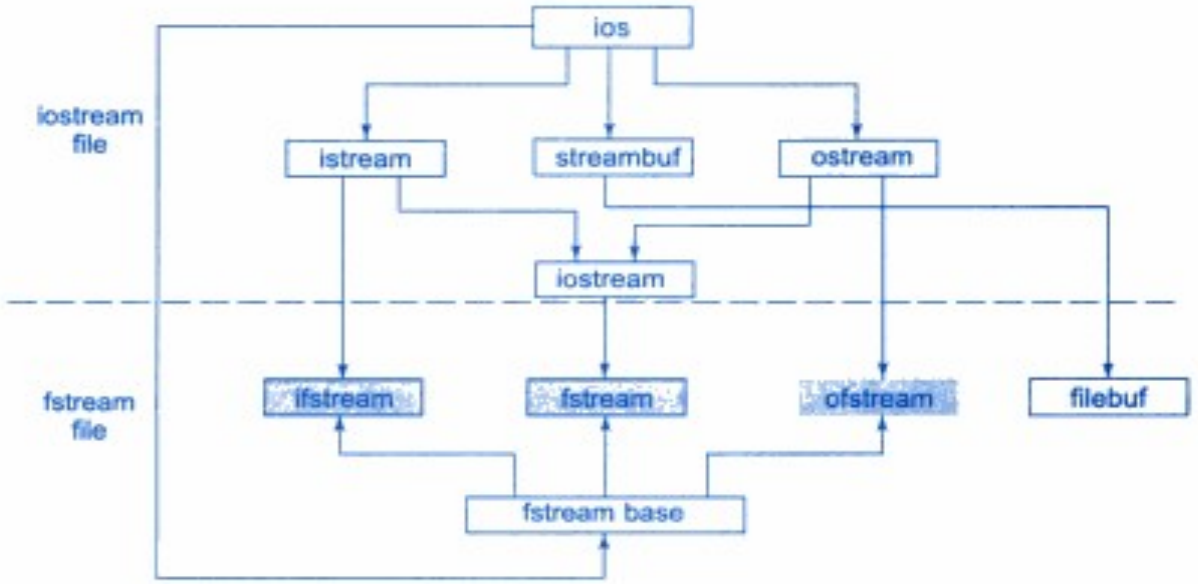
The I/O system of C++ handles file operations which are very much similar to the console input and output operations. It uses file streams as an interface between the programs and the files. The stream that supplies data to the program is known as *input stream* and the one that receives data from the program is known as *output stream*. In other words, the input stream extracts (or reads) data from the file and the output stream inserts (or writes) data to the file.'



File input and output streams

Classes for File Stream Operations

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



Stream classes for file operations (contained in fstream file)

Opening and Closing a File

1. Suitable name for the file.
2. Data type and structure.
3. Purpose.
4. Opening method.

Details of file stream classes

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 the 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()** and **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 make up a valid filename for the operating system. It may contain two parts, a primary name and an optional period with extension. Examples:

```
Input.data
Test.doc
```

As stated earlier, for opening a file, we must first create a file stream and then link it to the filename. A file stream can be defined using the classes **ifstream, ofstream,** and **fstream** that are contained in the header file *fstream*. The class to be used depends upon the purpose, that is, whether we want to read data from the file or write data to it. A file can be opened in two ways:

1. Using the constructor function of the class.
2. Using the member function **open()** of the class.

The first method is useful when we use only one file in the stream. The second method is used when we want to manage multiple files using one stream.

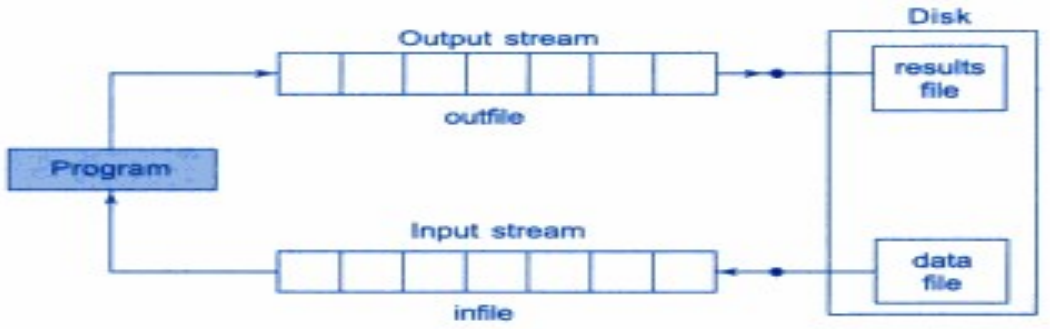
Opening Files Using Constructor

1. Create a file stream object to manage the stream using the appropriate class. That is to say, the class **ofstream** is used to create the output stream and the class **ifstream** to create the input stream.
2. Initialize the file object with the desired filename.

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

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

This creates **outfile** as an **ofstream** object that manages the output stream. This object can be any valid C++ name such as **o_file, myfile** or **fout**. This statement also opens the file results and attaches it to the output stream **outfile**.



Two file streams working on separate files

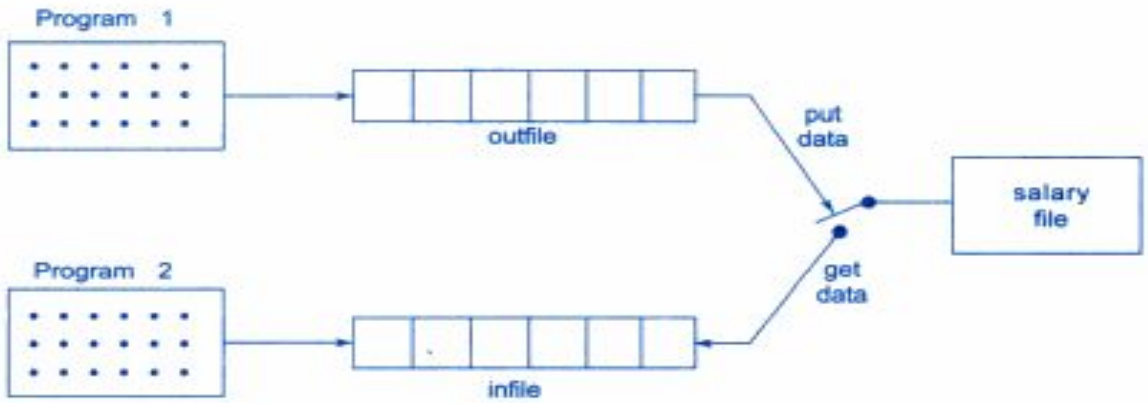
Similarly, the following statement declares **infile** as an **ifstream** object and attaches it to the file **data** for reading (input).

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

The program may contain statements like:

```
outfile << "TOTAL";
outfile << sum;
infile  >> number;
infile  >> string;
```

```
ofstream outfile("salary");          // creates outfile and connects
                                     // "salary" to it
.....
.....
Program2
.....
.....
ifstream infile("salary");           // creates infile and connects
                           // "salary" to it
.....
.....
```



Two file streams working on one file

The connection with a file is closed automatically when the stream object expires (when the program terminates). In the above statement, when the *program1* is terminated, the **salary** file is disconnected from the **outfile** stream. Similar action takes place when the *program 2* terminates.

Instead of using two programs, one for writing data (output) and another for reading data (input), we can use a single program to do both the operations on a file. Example.

```
.....
.....
     outfile.close();                  // Disconnect salary from outfile
     ifstream infile("salary");        // and connect to infile
.....
.....
     infile.close();   // Disconnect salary from infile
```

WORKING WITH SINGLE FILE

```
// Creating files with constructor function

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

int main()
{
        ofstream outf("ITEM");          // connect ITEM file to outf

        cout << "Enter item name:";
        char name[30];
        cin >> name;                    // get name from key board and

        outf << name << "\n";           // write to file ITEM

        cout << "Enter item cost:";
        float cost;
        cin >> cost;                    // get cost from key board and

        outf << cost << "\n";           // write to file ITEM

        outf.close();                   // Disconnect ITEM file from outf

        ifstream inf("ITEM");           // connect ITEM file to inf

        inf >>  name;                   // read name from file ITEM
        inf >> cost;                    // read cost from file ITEM

        cout << "\n";
        cout << "Item name:" << name << "\n";
        cout << "Item cost:" << cost << "\n";

        inf.close();                    // Disconnect ITEM from inf

        return 0;
}
```

The output of Program

```
Enter item name:CD-ROM
Enter item cost:250

Item name:CD-ROM
Item cost:250
```

Opening Files Using open()

As stated earlier, the function **open()** can be used to open multiple files that use the same stream object. For example, we may want to process a set of files sequentially. In such cases, we may create a single stream object and use it to open each file in turn. This is done as follows:

```
file-stream-class stream-object;
stream-object.open ("filename");
```

Example:

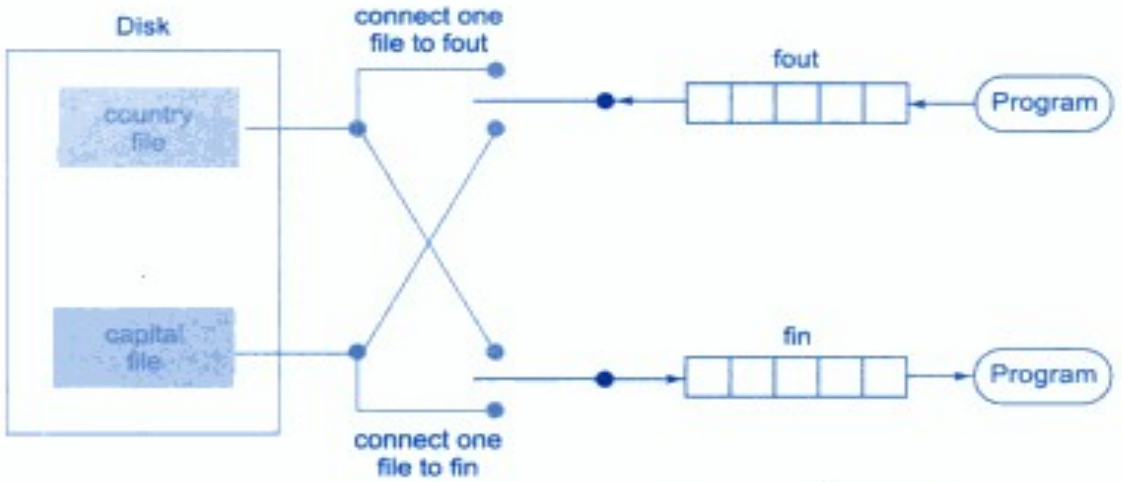
```
ofstream outfile;           // Create stream (for output)
outfile.open("DATA1");      // Connect stream to DATA1
.....
.....
outfile.close();            // Disconnect stream from DATA1
outfile.open("DATA2);       // Connect stream to DATA2
.....
.....
outfile.close();            // Disconnect stream from DATA2
.....
.....
```

WORKING WITH MULTIPLE FILES

```
// Creating files with open() function

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

int main()
{
   ofstream fout;                          // create output stream
   fout.open("country");                   // connect "country" to it
   fout << "United States of America\n";
   fout << "United Kingdom\n";
   fout << "South Korea\n";

   fout.close();                           // disconnect "country" and

   fout.open("capital");                   // connect "capital"

   fout  << "Washington\n";
   fout  << "London\n";
   fout  << "Seoul\n";

   fout.close();                           // disconnect "capital"

   // Reading the files
   const int N = 80;                       // size of line
   char line[N];

   ifstream fin;                           // create input stream
   fin.open("country");                    // connect "country" to it

   cout <<"contents of country file\n";

   while(fin)                              // check end-of-file
   {
          fin.getline(line, N);            // read a line
          cout << line ;                   // display it
   }

   fin.close();                            // disconnect "country" and
   fin.open("capital");                    // connect "capital"

   cout << "\nContents of capital file \n";

   while(fin)
   {
          fin.getline(line, N);
          cout << line ;
   }
   fin.close();

   return 0;
}
```

The output of Program

```
Contents of country file
United States of America
United Kingdom
South Korea

Contents of capital file
Washington
London
Seoul
```



Streams working on multiple files

READING FROM TWO FILES SIMULTANEOUSLY

```
// Reads the files created

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>                    // for exit() function

int main()
{
        const  int SIZE = 80;
        char line[SIZE];

        ifstream fin1, fin2;           // create two input streams
        fin1.open("country");
        fin2.open("capital");

        for(int i=1; i<=10; i++)
        {
                if(fin1.eof() != 0)
                {
                        cout << "Exit from country \n";
                        exit(1);
                }
                fin1.getline(line, SIZE);
                cout << "Capital of "<< line ;

                if(fin2.eof() != 0)
                {
                        cout << "Exit from capital\n";
                        exit(1);
                }

                fin2.getline(line,SIZE);
                cout << line << "\n";
        }
        return 0;
}
```

The output of Program would be:

```
Capital of United States of America
Washington
Capital of United Kingdom
London
Capital of South Korea
Seoul
```

Detecting end-of-file

Detection of the end-of-file condition is necessary for preventing any further attempt to read data from the file.

```
while(fin)
```

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. Remember, this loop may terminate due to other failures as well. (We will discuss other error conditions later.)

There is another approach to detect the end-of-file condition.

```
if(fin1.eof() != 0) {exit(1);}
```

eof() is a member function of **ios** class. It returns a non-zero value if the end-of-file(EOF) condition is encountered, and a zero, otherwise. Therefore, the above statement terminates the program on reaching the end of the file.

More about Open(): File Modes

We have used **ifstream** and **ofstream** constructors and the function **open()** to create new files as well as to open the existing files. Remember, in both these methods, we used only one argument that was the filename. However, these functions can take two arguments, the second one for *specifying the file mode*. The general form of the function **open()** with two arguments is:

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

The second *argument mode* (called file mode parameter) specifies the purpose for which the file is opened.

The prototype of these class member functions contain default values for the second argument and therefore they use the default values in the absence of the 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.
```

File mode parameters

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 the file does not exist
ios :: noreplace	Open fails if the file already exists
ios :: out	Open file for writing only
ios :: trunc	Delete the contents of the file if it exists

Sequential Input and Output Operations

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

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 put() and get() Functions

The classes istream and ostream define two member functions get() and put() respectively to handle the single character input/output operations.

There are two types of get() functions.

Both get(char*) and get(void) prototypes can be used to fetch a character including the blank space, tab and the newline character.

The get(char*) version assigns the input character to its argument and the get(void) version returns the input character. Since these functions are members of the input/output stream classes, these must invoked by using an appropriate object.

For instance, look at the code snippet given below:

```
char c;
while (c!= '\n')
{
cin.get (c);                //get a character from keyboard and assign it to c
cout << c;                  //display the character on screen cin.get (c);
 }                          //get another character
```

This code reads and displays a line of text (terminated by a newline character).

Remember, the operator> >can also be used to read a character but it will skip the white spaces and newline character.

The above while loop will not work properly if the statement

cin >> c;

is used in place of

cin.get (c);

The get(void) version is used as follows:

```
char c;
c = cin.get();      //cin.get (c) replaced
```

The value returned by the function get() is assigned to the variable c.

The function put(), a member of ostream class, can be used to output a line of text, character by character. For example,

```
cout. put ('x');
```

displays the character x and cout.put

```
(ch) ;
```

displays the value of variable ch.

The variable ch must contain a character value. We can also use a number as an argument to the function put (). For example,

```
cout . put (68) ;
```

displays the character D. This statement will convert the int value 68 to a char value and display the character whose ASCII value is 68,

The following segment of a program reads a line of text from the keyboard and displays it on the screen.

```
char c; .
cin.get (c)
while (c!= '\n'){
   cout. put(c); //display the character on screen
                                          //read a character
   cin.get (c ) ;
   }
```

The getline () and write () Functions

We can read and display a line of text more efficiently using the line-oriented input/output functions getline() and write(). The getline() function reads a whole line of text that ends with a newline character. This function can be invoked by using the object cin as follows:

cin.getline(line, size);

This function call invokes the function which reads character input into the variable line. The reading is terminated as soon as either the newline character '\n' is encountered or size number of characters are read (whichever occurs first). The newline. character is read but not saved. Instead, it is replaced by the null character.

For example; consider the following code:

char name [20] ;

cin.getline(name, 20);

Assume that we have given the following input through the keyboard:

Neeraj good

This input will be read correctly and assigned to the character array name. Let us suppose the input is as follows:

Object Oriented Programming

In this case, the input will be terminated after reading the following 19 characters:

Object Oriented Pro

After reading the string/ cin automatically adds the terminating null character to the character array.

Remember, the two blank spaces contained in the string are also taken into account, i.e.

between Objects and Oriented and Pro.

We can also read strings using the operator >> as follows:

cin >> name;

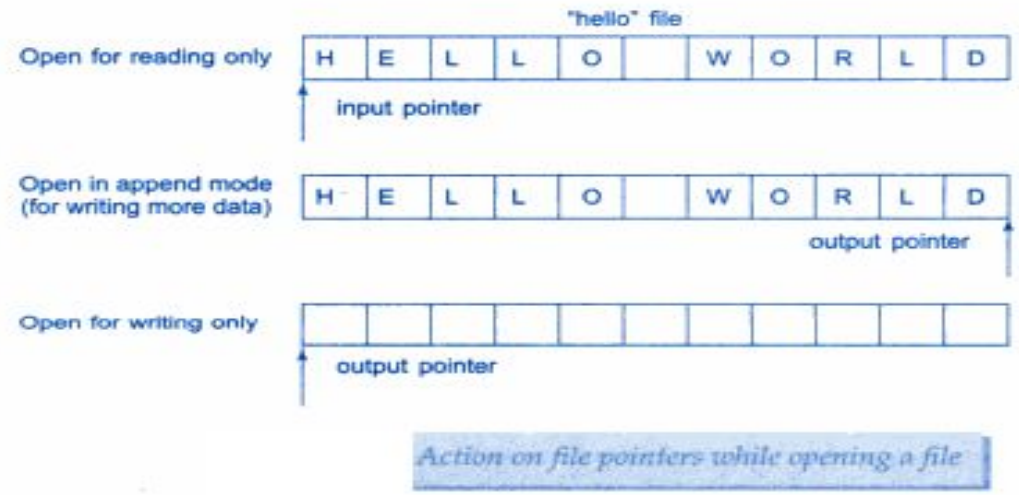
But remember cin can read strings that do not contain white space. This means that cin can read just one word and not a series of words such as “Neeraj good”

Random Access Files

File Pointers and Their Manipulations

Each file has two associated pointers known as the *file pointers*. One of them is called the input pointer (*or get pointer*) and the other is called the output pointer (*or put pointer*).

The input pointer is used for reading the contents 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.



Action on file pointers while opening a file

Functions for Manipulation of File Pointers

How do we then move a file pointer to any other desired position inside the file? This is possible only if we can take control of the movement of the file pointers ourselves. The file stream classes support the following functions to manage such situations:

- **seekg()** Moves get pointer (input) to a specified location.
- **seekp()** Moves put pointer(output) to a specified location.
- **tellg()** Gives the current position of the get pointer.
- **tellp()** Gives the current position of the put pointer.

For example, the statement

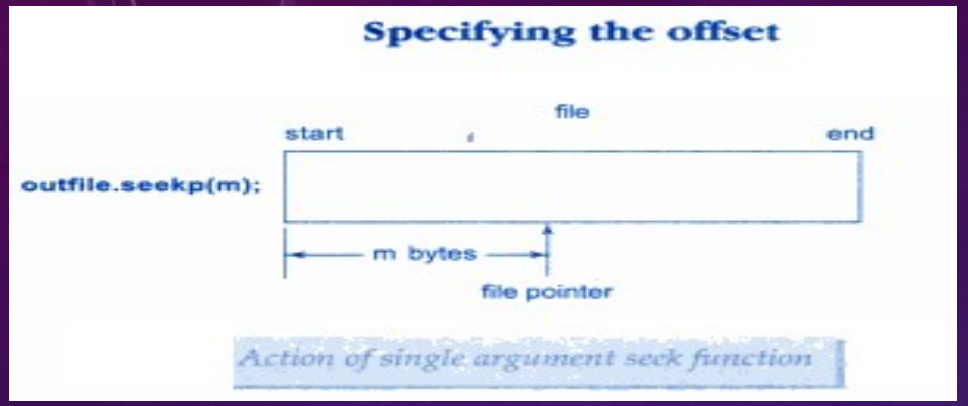
```
infile.seekg(10);
```

moves the file pointer to the byte number 10. Remember, the bytes in a file are numbered beginning from zero. Therefore, the pointer will be pointing 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 the file "hello" and the value of **p** will represent the number of bytes in the file.



'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 the file
- **ios::cur** current position of the pointer
- **ios::end** End of the file

The **seekg()** function moves the associated file's 'get' pointer while the **seekp()** function moves the associated file's 'put' pointer. Table ` lists 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 form the current position
fout.seekg(-m,ios::cur);	Go backward by m bytes from the current position
fout.seekg(-m,ios::end);	Go backward by m bytes form the end

Reading and Writing a Class Object

Since the class objects are the central elements of C++ programming, it is quite natural that the language supports features for writing to and reading from the disk files objects directly. The binary input and output functions **read()** and **write()** are designed to do exactly this job. These functions handle the entire structure of an object as a single unit, using the computer's internal representation of data. For instance, the function **write()** copies a class object from memory byte by byte with no conversion. One important point to remember is that only data members are written to the disk file and the member functions are not.

Program : illustrates how class objects can be written to and read from the disk files. The length of the object is obtained using the **sizeof** operator. This length represents the sum total of lengths of all data members of the object.

Reading and Writing a Class Object

READING AND WRITING CLASS OBJECTS

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

class INVENTORY
{
        char  name[10];                     // item name
        int   code;                         // item code
        float cost;                         // cost of each item
  public:
        void readdata(void);
        void writedata(void);
};

void INVENTORY :: readdata(void)            // read from keyboard
{
        cout << "Enter name: "; cin >> name;
        cout << "Enter code: "; cin >> code;
        cout << "Enter cost: "; cin >> cost;
}

void INVENTORY :: writedata(void)            // formatted display on
{                                            // screen
        cout << setiosflags(ios::left)
             << setw(10) << name
             << setiosflags(ios::right)
             << setw(10) << code
             << setprecision(2)
             << setw(10) << cost
             << endl;
}

int main()
{
        INVENTORY item[3];                  // Declare array of 3 objects

        fstream file;                       // Input and output file

        file.open("STOCK.DAT", ios::in | ios::out);

        cout << "ENTER DETAILS FOR THREE ITEMS \n";
        for(int i=0;i<3;i++)
        {
              item[i].readdata();

              file.write((char *) & item[i],sizeof(item[i]));
        }
        file.seekg(0);                      // reset to start

        cout << "\nOUTPUT\n\n";
        for(i = 0;  i < 3; i++)
        {
              file.read((char *) & item[i], sizeof(item[i]));
              item[i].writedata();
        }
        file.close();
        return 0;
}
```

The output of Program :

```
ENTER DETAILS FOR THREE ITEMS
Enter name: C++
Enter code: 101
Enter cost: 175
Enter name: FORTRAN
Enter code: 102
Enter cost: 150
Enter name: JAVA
Enter code: 115
Enter cost: 225

OUTPUT

C++             101   175
FORTRAN         102   150
JAVA            115   225
```

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 the error conditions

The class **ios** supports several member functions that can be used to read the status recorded in a file stream. These functions along with their meanings are listed in Table

Table *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 continue 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 can be carried out.

These functions may be used in the appropriate places in a program to locate the status of a file stream and thereby to take the necessary corrective measures.

REFERENCES:

1. E. Balagurusamy, “Object Oriented Programming with C++”, Fourth edition, TMH, 2008.

2.LECTURE NOTES ON Object Oriented Programming Using C++ by Dr. Subasish Mohapatra, Department of Computer Science and Application College of Engineering and Technology, Bhubaneswar Biju Patnaik University of Technology, Odisha

3. K.R. Venugopal, Rajkumar, T. Ravishankar, “Mastering C++”, Tata McGraw-Hill Publishing Company Limited

4. Object Oriented Programming With C++ - PowerPoint Presentation by Alok Kumar

5. OOPs Programming Paradigm – PowerPoint Presentation by an Anonymous Author