Pointers

Pointers to objects

Just as you can have pointers to other types of variables, you can have pointers to objects. When accessing members of a class given a pointer to an object, use the arrow (->) operator instead of the dot operator.

Program:

```
#include<iostream.h>
#include<conio.h>
class A
{
 public:
  int sum(int a,int b)
  {
   return(a+b);
  }
};
void main()
{
 clrscr();
 A *obj;
 cout<<obj->sum(6,4)<<endl;
 getch();
}
output:
10
```

Memory management using new and delete operators

C++ provides two dynamic allocation operators: new and delete. These operators are used to allocate and free memory at run time.

The new operator can be used to create objects of any type. It takes the following general form:

```
Pointer_variable=new data_type
```

Here pointer_variable is a pointer of type data_type. The new operator allocates sufficient memory to hold a data object of type data_type and returns the address of the object. The data type may be any valid data type. The pointer variable holds the address of the memory space allocated.

When a data object is no longer needed, it is destroyed to release the memory space for reuse. The general form of its use is:

```
delete pointer_variable
```

The pointer_variable is the pointer that points to a data object created with new.

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 pointer is automatically passed to a member function when it is called. The pointer this acts as an implicit argument to all the member functions.

Program:

```
#include<iostream.h>
#include<conio.h>
class A
{
 public:
  int a,b,c;
 public:
  void sum(void)
  {
   cin>>this->a>>this->b;
   this->c=this->a+this->b;
   cout<<this->c<<endl;
  }
};
void main()
{
 A obj;
 clrscr();
 obj.sum();
 getch();
}
output:
20 10
30
```

Virtual functions

A function qualified by the virtual keyword. When a virtual function is called via a pointer, the class of the object pointed to determine which function definition will be used. Virtual functions implement polymorphism, whereby objects belonging to different classes can respond to the same message in different ways.

Rules for virtual functions

1. The virtual functions must be a 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.

7. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class.

Program:

```
#include<iostream.h>
#include<conio.h>
class A
{
 public:
  virtual int arith(int a,int b)
  {
   return(a+b);
  }
};
class B : public A
{
 public:
  int arith(int a,int b)
  {
   return(a-b);
  }
};
class C : public B
{
 public:
  int arith(int a,int b)
  {
   return(a*b);
  }
};
class D : public C
{
 public:
  int arith(int a,int b)
  {
   return(a/b);
  }
};
void main()
{
 A *obj;
 A obj1;
 B obj2;
 C obj3;
 D obj4;
 clrscr();
 obj=&obj1;
 cout<<obj->arith(5,6)<<endl;
 obj=&obj2;
 cout<<obj->arith(5,6)<<endl;
 obj=&obj3;
 cout<<obj->arith(5,6)<<endl;
 obj=&obj4;
 cout<<obj->arith(5,6)<<endl;
 getch();
}
output:
11
-1
```

```
30
0
```

Pure virtual functions

A virtual function, equated to zero is called a pure virtual function. It is a function declared in a base class that has no definition relative to the base class. A class containing such pure function is called an abstract class.

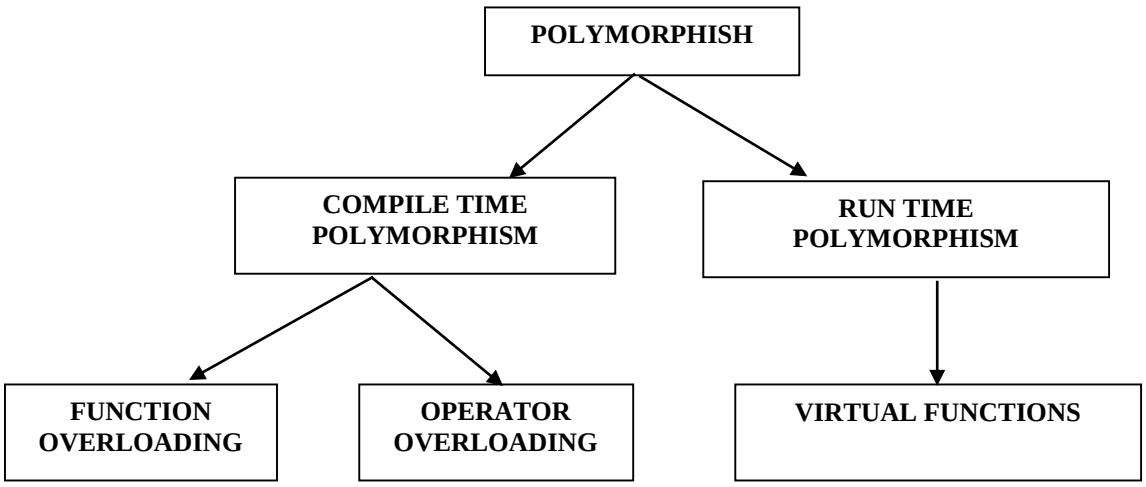
Eg:

```
Virtual int sum()=0;
```

Polymorphism

Polymorphism simply means 'one name', 'multiple forms'. The overloaded member functions are 'selected' for invoking by matching arguments, both type and number. The compiler knows this information 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 a static linking. Also known as compile time polymorphism.

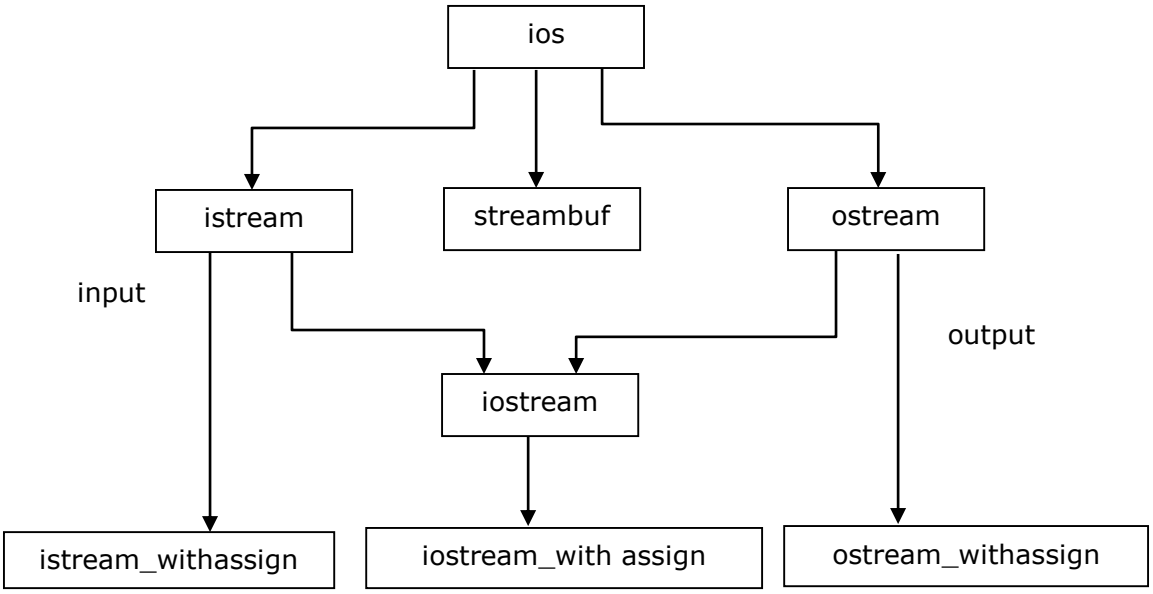
At run time, when it is known what class objects are under consideration the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as late binding. It is also known as dynamic binding because the selection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



13. STREAMS

Stream

A source from which input data can be obtained or a destination to which output data can be sent. The source stream that provides data to the program is called the input stream and the destination. Stream that receives output from the program is called the output stream.



ios

1. Contains basic facilities that are used all other input and output classes.
2. Declares constants and functions that are necessary for handling form input and output operation.

istream

1. Inherits the properties of ios.
2. Declares input functions such as get(), getline(), and read().
3. Contains overloaded extraction operator >>.

ostream

1. Inherits the properties of ios.
2. Declares output functions put() and write()
3. Contains overloaded insertion operator <<.

iostream

1. Inherits the properties of ios, istream and ostream through multiple inheritance and thus contains all the input and output functions.

streambuf

1. Provides an interface to physical devices through buffers.
2. Acts as a base for filebuf class used ios files.

Uniformatted I/O Operations

Overloaded operators >> and <<

cin and cout are defined in the iostream.h file 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.

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.

Syntax for get function

```
cin.get(variable_name);
```

(or)

```
variable_name=cin.get();
```

Syntax for put function

```
cout.put(variable_name);
```

Program:

```
#include<iostream.h>
#include<conio.h>
void main()
{
 char ch;
 clrscr();
 cin.get(ch);
 cout.put(ch);
 getch();
}
output:
k
k
```

getline() and write() functions

The getline() function reads a whole line of text that ends with a new line character.

Syntax for getline():

```
cin.getline(line,size);
```

This function call invokes the function getline() which reads character input into the variable line. The reading is terminated as soon as either the new line character '\n' in encountered or size-1 characters are read.

The write() function displays an entire line and has the following form:

```
cout.write(line,size);
```

The first argument line represents the name of the string to be displayed and the second argument size indicates the number of characters to display.

Program:

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
void main()
{
 char ch[20];
 int k;
 clrscr();
 cin.getline(ch,20);
 k=strlen(ch);
 cout.write(ch,k);
 getch();
}
output:
Govind
Govind
```

Formatted console I/O operations

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

These features include:

1. ios class functions and flags.
2. Manipulators
3. User-defined output functions.

ios class functions and flags

1. width(): To specify the required field size for displaying an output value.

Syntax:

```
cout.width(w)
```

Where w is the field width(number of columns). The output will be printed in a field of w character wide at the right end of the field.

2. precision(): To specify the number of digits to be displayed after the decimal point of float value.

Syntax:

```
Cout.precision(d);
```

Where d is the number of digits to the right of the decimal point.

3. fill(): To specify a character that is used to fill the unused portion of a field.

Syntax:

```
cout.fill(ch);
```

Where ch represents the character which is used for filling the unused for filling the unused positions.

4. setf(): To specify format flags that can control the form of output display

Syntax:

```
cout.setf(arg1,arg2);
```

The arg1 is one of the formatting flags defined in the class ios. The formatting flag specifies the format action required for the output. Another ios constant, arg2 known as bit field specifies the group to which the formatting flags belongs.

Format required	Flag(arg1)	Bit-field(arg2)
Left-justified	ios::left	ios::adjustfield
Right-justified	ios::right	ios::adjustfield
Padding after sign or base Indicator	ios::internal	ios::adjustfield
Scientific notation	ios::scientific	ios::float field
Fixed notation	ios::fixed	ios::float field
Decimal base	ios::dec	ios::basefield
Octal base	ios::oct	ios::basefield
Hexadecimal base	ios::hex	ios::basefield

Flags that do not have bit fields

Flag	Meaning
ios::showbase	Use base indicator on output
ios::showpos	Print + before
ios::showpoint	Show trailing decimal point and zeros
ios::uppercase	Use uppercase letters for hex output
ios::skipws	Skip white space on input
ios::unitbuf	Flush all streams after insertion

Managing output with manipulators

Manipulator functions are special stream functions that change certain characteristics of the input and output. The main advantage of using manipulator functions is that they facilitate the formatting of input and output stream.

1. endl: The endl is an output manipulator to generate a carriage return or line feed character. The endl may be used severl times in a single statement.
2. setw(): The setw() manipulator set the width of the output field.

   Syntax:

   ```
   setw(int width)
   ```

3. setprecision(): This manipulator control the display of number of digits after the decimal point for the floating numbers.

   Syntax:

   ```
   setprecision(int);
   ```

4. setfill(): This manipulator fills the given character in the unused field.

   Syntax:

   ```
   setfill(char);
   ```

5. setbase(): This manipulator is used to convert the base of one value into another base value. There are three common base converters in c++. They are dec, hex and oct.

   ```
   setbase(10)
   setbase(8)
   setbase(16)
   ```

6. setiosflags(): This manipulator is used to set the format flag.

   Syntax:

   ```
   setiosflags(format flag);
   ```

7. resetiosflags(): This manipulator is used to clear the flag specified by format flag.

   Syntax:

   ```
   resetiosflags(format flag);
   ```

Designing our own manipulators

We can design our own manipulators for certain special purposes. The general form for creating a manipulator without any argument is

```
ostream & manipulator(ostream & output)
{
 ---------------------
 ---------------------
 ---------------------
```

```
 return output;
}
```

Here the manipulator is the name of the manipulator under creation.

Program:

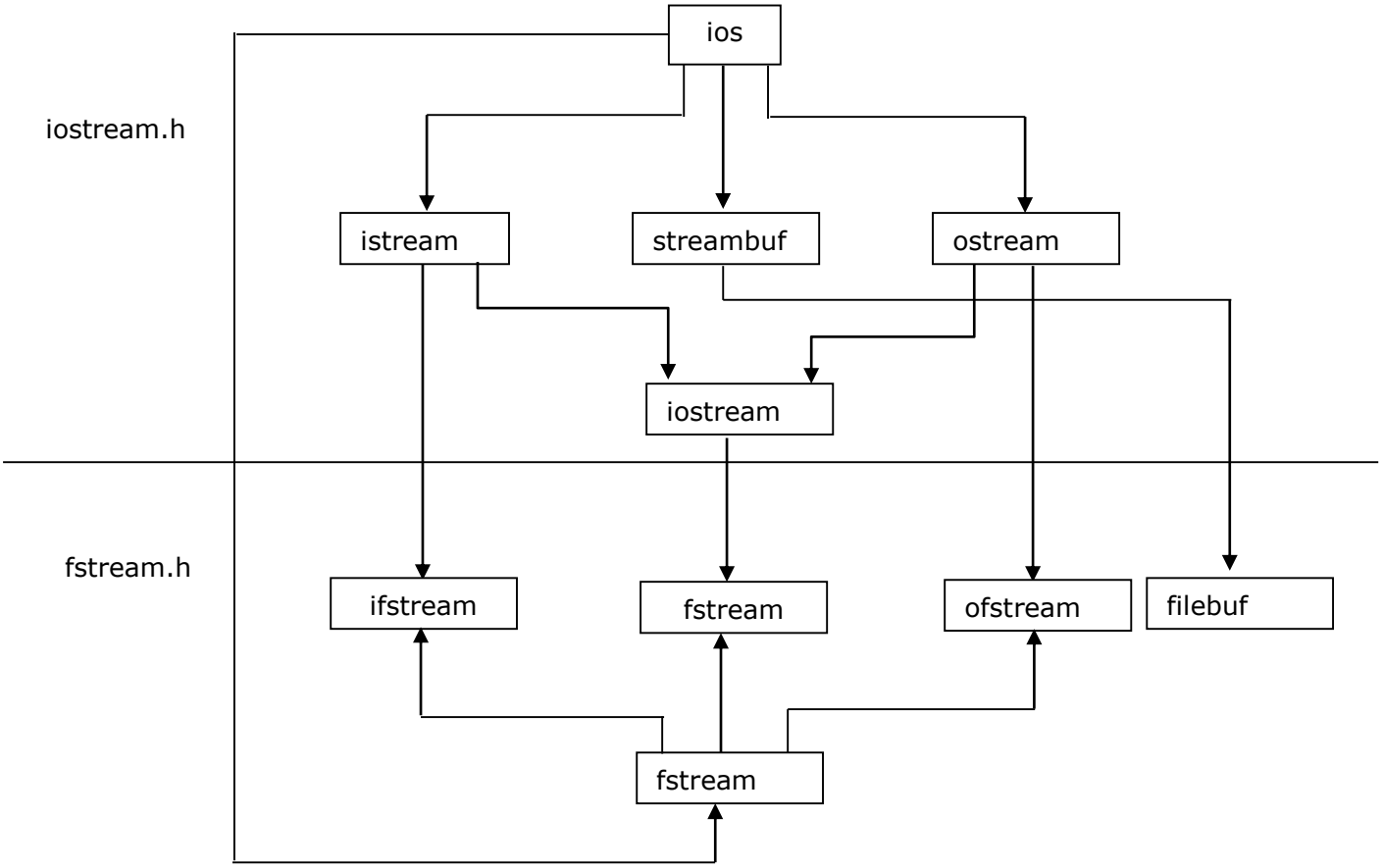
```
#include<iostream.h>
#include<conio.h>
#include<string.h>
#include<iomanip.h>
ostream & currency(ostream & output)
{
 cout<<setw(10);
 cout<<setbase(16);
 cout<<setiosflags(ios::uppercase);
 cout<<setiosflags(ios::showbase);
 return output;
}
void main()
{
 int k;
 clrscr();
 cin>>k;
 cout<<currency<<k;
 getch();
}

output:
28
      OX3D
```

File streams

A file is a collection of related data stored in a particular area on the disk.

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.

Classes for file stream operations



filebuf

Its purpose is to set the file buffers to read and write. Contains openprot constants 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.

Opening a file

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.

Opening file 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 file name.

   Syntax:

```
File_stream_class stream_object(file name);
```

Opening files using open()

Open() function is used to create new files as well as to open the existing file.

Syntax:

```
File_stream_class stream_object;
Stream_object.open("filename",mode);
```

Here, filename is the name of the file, which may include a path specifier. The value of mode determines how the file is opened. It must be one(or more) of these values.

ios::app :- appends to end of file
ios::ate :- seek to end of file when file is opened.
ios::binary :- Open file in the binary mode
ios::in :- Open file for reading only
ios::nocreate :- Causes open() function to fail if the file does not exist
ios::noreplace:- causes open() function fail if file exists.
ios::out :- Open file for writing only
ios::trunc :- Delete the contents of the file if it exists.

Note:

1. Opening a file in ios::out mode also opens it in the ios::trunc mode by default.
2. Both ios::app and ios::ate take us to the end of the file when it is opened. The difference between the two parameters is that the ios::app allows us to add data to the end of the file only, while ios::ate mode permits us to add data or to modify the existing data anywhere in the file. In both the cases, a file is created by a specified name, if it does not exist.
3. Creating a stream using ifstream implies input and creating a stream using ofstream implies output. So in these cases it is not necessary to provide the mode parameters.
4. The mode can combine two or more parameters using the bitwise OR operator (Symbol **|**).

Close a file

To close a file, the member function close() must be used. The close function takes no parameters and returns no value.

Syntax:

```
Stream_object.close();
```

Reading and writing files

Reading from the file and writing to a text file is very easy. The << and >> operators are used the same way the console I/O operations are performed, except that instead of using cin and cout, the stream that is linked to the file must be substituted.

Opening file using constructors

Program: Writing character into a file:

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
 ofstream outf("emp");
 char ch;
 clrscr();
 cin>>ch;
 outf<<ch;
 outf.close();
 getch();
}
```

Program: Reading character from the file

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
 ifstream inf("emp");
 char ch;
 clrscr();
 inf>>ch;
 cout<<ch;
 inf.close();
 getch();
}
```

Program: Opening file using open function

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
 ofstream outf;
 char ch;
 clrscr();
 outf.open("sen",ios::out);
 cin>>ch;
 outf<<ch;
 outf.close();
 getch();
}
```

Program:

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
 ifstream inf;
 char ch;
 clrscr();
 inf.open("sen",ios::in);
 inf>>ch;
 cout<<ch;
 inf.close();
 getch();
}
```

Program: Writing a number into a file:

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
 ofstream outf("emp");
 int n;
```

```
 clrscr();
 cin>>n;
 outf<<n;
 outf.close();
 getch();
}
```

Program: Reading number from the file:

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
 ifstream inf("emp");
 int n;
 clrscr();
 inf>>n;
 cout<<n;
 inf.close();
 getch();
}
```

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.

Syntax for put() function:

```
 Stream_object.put(variable_name);
```

Program:

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
 ofstream outf("sen");
 char ch;
 clrscr();
 cin>>ch;
 outf.put(ch);
 outf.close();
 getch();
}
```

Syntax for get() function:

```
 Stream_object.get(variable_name);
```

Program:

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
 ifstream inf("sen");
 char ch;
```

```
 clrscr();
 inf.get(ch);
 cout<<ch;
 inf.close();
 getch();
}
```

read() and write() functions

The functions read() and write(), unlike the function put() and get(), handle the data in binary form. This means that the values are stored in the disk file in the some format in which they are stored in the internal memory. The read() and write() functions are string oriented. These functions are called with two parameters.

Syntax for write function:-

```
Outfile.write((char *) & variable,size(var));
```

Program:

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
 ofstream outf("sen");
 int a[5],i;
 clrscr();
 for(i=0;i<5;i++)
 {
  cin>>a[i];
 }
 outf.write((char *) &a,sizeof(a));
 outf.close();
 getch();
}
```

Syntax for read function:

```
Infile.read((char *) & variable,size(var));
```

Program:

```
#include<iostream.h>
#include<conio.h>
#include<fstream.h>
void main()
{
 ifstream inf("sen");
 int a[5],i;
 clrscr();
 inf.read((char *) &a,sizeof(a));
 for(i=0;i<5;i++)
 {
  cout<<a[i]<<"\t";
 }
 inf.close();
 getch();
}
```

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). We can use these pointers to move through the files while reading or writing. 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.

Function for manipulation of file pointer

1. seekg():- Moves get pointer(input) to a specified location.
2. seekp():- Moves put pointer(output) to specified location.
3. tellg():- Gives the current position of the get pointer.
4. tellp():- Gives the current position of the put pointer.

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 take one of the following three contents defined in the ios class.

1. ios::beg Start of the file
2. ios::cur Current position of the pointer.
3. ios::end End of the file

Seek call	Meaning
Fout.seekg(0,ios::beg);	Go to start
Fout.seekg(0,ios::cur);	Stay at the current position.
Fout.seekg(0,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 bytes from 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 from the end

Program:

```
#include<iostream.h>
#include<conio.h>
#include<stdlib.h>
#include<stdio.h>
#include<fstream.h>
```

```
#include<iomanip.h>
class group
{
 private:
 struct person
 {
  int regno,m1,m2,m3;
  char name[20],flag;
 }p;
 fstream file;
 public:
  group();
  void addrec();
  void listrec();
  void modirec();
  void delrec();
  void recallrec();
  void packrec();
  void exit();
};
void main()
{
 char choice;
 group g;
 do
 {
  clrscr();
  cout<<"1.add records "<<endl;
  cout<<"2.list records"<<endl;
  cout<<"3.modify records"<<endl;
  cout<<"4.delete records"<<endl;
  cout<<"5.recall records"<<endl;
  cout<<"6.pack records"<<endl;
  cout<<"0.exit records"<<endl;
  cout<<"your choice?"<<endl;
  cin>>choice;
  clrscr();
  switch(choice)
  {
   case '1':
       g.addrec();
       break;
   case '2':
      g.listrec();
      break;
   case '3':
      g.modirec();
      break;
   case '4':
      g.delrec();
      break;
   case '5':
      g.recallrec();
      break;
   case '6':
      g.packrec();
      break;
   case '0':
       g.exit();
       exit(1);
  }
```

```
 }while(choice!='0');
}
void group :: group()
{
 file.open("emp.dat",ios::binary|ios::in|ios::out);
 if(!file)
 {
  cout<<endl<<"unable to open file";
  exit();
 }
}

//ADD RECORDS TO THE FILE AT END

void group::addrec()
{
 char ch;
 file.seekp(0,ios::end);
 do
 {
      cout<<endl<<"entern student regno,name,and
      marks"<<endl;
      cin>>p.regno>>p.name>>p.m1>>p.m2>>p.m3;
      p.flag=' ';
      file. write((char*)&p,sizeof(p));
      cout<<"add another record?(y/n)";
      cin>>ch;
 }while(ch=='y');
}

//LIST ALL RECORDS IN THE FILES

void group :: listrec(void)
{
 int a;
 file.seekg(0L,ios::beg);
 while (file.read((char*)&p,sizeof(p)))
 {
      cout<<endl<<setw(2)<<p.flag<<setw(6)<<p.regno<<
      setw(15)<<p.name<<setw(3)<<p.m1<<setw(3)<<p.m2<<setw(3)<<p.m
      3;
 }
 file.clear();
 cout<<endl<<"press any key......";
 getch();
}

//MODIFILES A GIVEN RECORD FROM THE FILE

void group::modirec()
{
 int reg;
 int count=0;
 long int pos;
 cout<<"enter the register no";
 cin>>reg;
 file.seekg(0,ios::beg);
 while(file.read((char*) &p,sizeof(p)))
 {
  if(p.regno==reg)
  {
```

```
        cout<<endl<<"enter new record"<<endl;
        cin>>p.regno>>p.name>>p.m1>>p.m2>>p.m3;
        p.flag=' ';
        pos=count*sizeof(p);
        file.seekp(pos,ios::beg);
        file.write((char*)&p,sizeof(p));
        return;
       }
       count++;
      }
      cout<<endl<<"press any key .....";
      getch();
      file.clear();
     }

     //MARKS A RECORD FOR DELETION

     void group::delrec()
     {
      int reg;
      long int pos;
      int count=0;
      cout<<"enter student register no";
      cin>>reg;
      file.seekg(0,ios::beg);
      while(file.read((char*)&p,sizeof(p)))
      {
       if(p.regno==reg)
       {
        p.flag='*';
        pos=count*sizeof(p);
        file.seekp(pos,ios::beg);
        file.write((char*)&p,sizeof(p));
        return;
       }
       count++;
      }
      cout<<endl<<"press any key....";
     }
     getch();

//RECALLS THE RECORDS WHICH WAS EARLIER MARKED FOR DELETION

     void group ::recallrec()
     {
     int reg;
     long int pos;
     int count=0;
     cout<<"enter student registerno";
     cin>>reg;
     file.seekg(0,ios::beg);
     while(file.read((char*)&p,sizeof(p)))
     {
     if(p.regno==reg)
     {
     p.flag='  ';
     pos=count*sizeof(p);
     file.seekp(pos,ios::beg);
     file.write((char*)&p,sizeof(p));
     return;
     }
```

```
        count++;
        }
        cout<<endl<<"press any key";
        getch();
        file.clear();
        }

//REMOVES ALL RECORDS WHICH HAVE BEEN MARKED FOR DELETION.

        void group::packrec()
        {
        ofstream outfile;
        outfile.open("temp",ios::out);
        file.seekg(0,ios::beg);
        while(file.read((char*) &p,sizeof(p)))
        {
        if(p.flag!='*')
         outfile.write((char*)&p,sizeof(p));
        }
        outfile.close();
        file.close();
        remove("emp.dat");
        rename("temp","emp.dat");
        file.open("emp.dat",ios::binary|ios::in|ios::out|ios::nocreate);
        }
        void group::exit()
        {
        file.close();
        }
```

Error handling during a file operations

The following are some of the situations that arise while manipulating a file.

1. Attempting to open a non-existent file in read mode.
2. Trying to open a read only file in write mode.
3. Trying to open a file with invalid name.
4. Attempting to read beyond the end of the file.
5. There may not be any space in the disk for storing more data.
6. We may attempt to perform an operation when the file is not opened for the purpose.

The class ios supports several member functions that can be used to read the status recorded in a file stream.

eof():- Returns true(non-zero value) if end of file is encountered while reading. Other wise 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.

Command line arguments

Like C, C++ too supports a feature that facilitates the supply of arguments to the main() function. These arguments are supplied at the time of invoking the program.

The command line arguments are typed by the user and are delimited by a space. The first argument is always the file name (command name) and contains the program to be executed.

The main function can take two arguments by shown below:

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

The first argument argc (known as argument counter) represents the number of arguments in the command line. The second argument argv (known as argument value) is an array of char type pointers that points to the command line arguments.

Program:

```
#include<stdio.h>
#include<conio.h>
main(int argc,char *argv[])
{
 clrscr();
 cout<<argc<<endl;
 cout<<argv[1]<<endl;
 getch();
}
```

14. TEMPLATES

Generic functions

A generic function defines set of operations that will be applied to various types of data. A generic function has the type of data that it will operate upon passed to it as a parameter.

Templates

A template can be used to create a family of classes or functions. For example, a class template for an array class would enable us to create arrays of various data types such as int array and float array. Similarly, we can define a template for a function, say mul(), that would help us create various versions of mul() for multiplying int, float and double type values.

A template can be considered as a kind of macro. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type. Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized classes or functions.

Syntax for creating generic function:

```
template<class Type>
return_type function_name(parameter list)
{
  Body of the function
}
```

Type is a placeholder name for the data type used by the function. This name may be used within the function definition. However, it is only a placeholder that the compiler will automatically replace with an actual data type.

Program: Function with one generic types

```
#include<iostream.h>
#include<conio.h>
template<class T>
T mul(T a,T b)
{
 return(a*b);
}
void main()
{
 int a,b;
 float p,q;
 clrscr();
 cin>>a>>b;
 cout<<mul(a,b)<<endl;
 cin>>p>>q;
 cout<<mul(p,q)<<endl;
```

```
 getch();
}
output:
20 10
200
1.3 2.3
2.99
```

Program: Function with two generic types:

```
#include<iostream.h>
#include<conio.h>
template<class T1,class T2>
T1 mul(T1 a,T2 b)
{
 return(a*b);
}
void main()
{
 int a,b;
 float p,q;
 clrscr();
 cin>>a>>b;
 cin>>p>>q;
 cout<<mul(a,p)<<endl;
 cout<<mul(q,b)<<endl;
 getch();
}
2 3
1.3 2.3
2
6.9
```

Generic classes

In additions to generic functions, one can also define a generic class. In the generic class, the actual type of the data being manipulated will be specified as a parameter when objects of the class are created.

A class created from a class template is called a template class. The syntax for defining an object of a template class is:

```
Classname <type> objectname;
```

Program: Class with one generic data type

```
#include<iostream.h>
#include<conio.h>
template<class T>
class A
{
 public:
  T mul(T a,T b)
  {
   return(a*b);
  }
};
```

```
void main()
{
 int a,b;
 float p,q;
 A <int> obj1;
 A <float> obj2;
 clrscr();
 cin>>a>>b;
 cout<<obj1.mul(a,b)<<endl;
 cout<<obj2.mul(a,b)<<endl;
 getch();
}
output:
20 10
200
1.3 2.3
2.99
```

Class with two generic data types

A template class can have more than one generic data type. Simply declare all the data types required by the class in a comma-separated list within the template specification.

Program:

```
#include<iostream.h>
#include<conio.h>
template<class T1,class T2>
class A
{
 public:
  T1 mul(T1 a,T2 b)
  {
   return(a*b);
  }
};
void main()
{
 int a,b;
 float p,q;
 A <int,float> obj1;
 A <float,int> obj2;
 clrscr();
 cin>>a>>b;
 cin>>p>>q;
 cout<<obj1.mul(a,p)<<endl;
 cout<<obj2.mul(q,b)<<endl;
 getch();
}
output:
2 3
1.3 2.3
2
6.9
```

Templates and arrays

Using templates we can store different values into an array. **Program:**

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

```
#include<conio.h>
template<class T>
class A
{
 public:
  T a[5];
};
void main()
{
 clrscr();
 A <int> obj1;
 A <float> obj2;
 A <char> obj3;
 obj1.a[0]=45;
 obj1.a[1]=23;
 obj3.a[2]='e';
 obj2.a[3]=45.23;
 obj3.a[4]='y';
 cout<<obj1.a[0]<<endl;
 cout<<obj1.a[1]<<endl;
 cout<<obj3.a[2]<<endl;
 cout<<obj2.a[3]<<endl;
 cout<<obj3.a[4]<<endl;
 getch();
}
output:
45
23
e
45.23
y
```