Chapter – 9

Virtual Function and Run Time Polymorphism

9.1 Pointers

One important use for pointers is in the dynamic allocation of memory, carried out in C++ with the keyword new and delete.

9.1.1 New and Delete Operator

Pointer provides the necessary support for C++ powerful dynamic memory allocation system. Dynamic allocation is the means by which a program can obtain memory while it is running. C uses malloc( ) and calloc( ) functions to allocate memory dynamically at run time. Similarly, it uses the function free( ) to free dynamically allocated memory.

Although C++ supports these functions, it also defines two unary operators new and delete that perform the task of allocating and freeing the memory in a better and easier way. An object can be created by using new and destroyed by using delete as and when required.

new operator is used to allocate memory dynamically i.e., at run time. new operator obtains memory from the operating system and returns a pointer to the starting point.

The syntax for the *new* operator:

```
pointer_variable=new data-type;
```

Where pointer_variable is a previously declared pointer of type type_name. type_name can be any basic data type or user-defined object (enum, class, and struct included).

Example:

```
int *p;          //int *p = new int;
p = new int;

float *q;        //int *q = new float;
q = new float;
```

Type of variable mentioned on the left hand side and the type mentioned on the right hand side should match. When a data object is no longer needed, it should be destroyed to release the memory space for reuse. For this purpose, we use ***delete*** unary operator.

<u>The general syntax for delete operator:</u>

```
delete pointer_variable;
```

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

Example:

```
delete p;
delete q;
delete [] s;      //multiple objects.
```

8.2 Pointer to Objects

There are two ways to declare pointer to objects.

- **Usual way:** Just like pointers to normal variables, we can have pointers to class variables, i.e. objects.

  Syntax:

  class_name * ptr_name = & object_name;

  Example:

  A b;

  A *a = & b; **// Here, b is an object and a is pointer to it.**

- **Using new Operator: new operator can be used to define pointer to objects.**

  Syntax:

  class_name *ptr_name = new class_name;

  Example:

  A *a = new A; **// Here, a is pointer to unnamed object.**

9.3 Pointer to Derived Class

Base class object pointer can point any type of derived class objects. Pointers to objects of a base class are type compatible with pointers to objects of 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 declaration:

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

9.4 Accessing Class Member Using Pointer to Objects

We can access class members i.e., member functions and data members using pointer to object. This can be done by using object pointer and arrow operator(->).

Syntax:

object_pointer->class_member;

<u>Example:</u>

```
class Sample
{
   int a, b;
public:
   int p;
Sample(int x, int y )
   {
      a=x;
      b=y;
   }
   void display( )
   {
      cout<< " a, b, p :"<<a<<b<<p;
   }
};
int main( )
{
   Sample s(7,8);
   s.p=9;
   s.display();
   Sample *a =&s;
   a->p=10;                    //accessing public data member p using object pointer a.
   a->display();               //accessing public member function display() using object pointer a.
//Another way to define pointer to object.
   Sample * k = new Sample; //using new operator, here object pointer k points to unnamed object.
   k->p=80;
   k->display();
   return 0;
}
```

9.5 this Pointer in C++

Every object in C++ has access to its own address through an important pointer called this pointer. The this pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

this is a special pointer which points to the object that is currently invoking a particular member function. For example, the function call a.display() will set the pointer “this” to the address of the calling object a, and again b.display() will set the pointer "this" to the address of the object b. this pointer is automatically passed to a member function when it is called. Therefore, inside a member function, this pointer is used to refer to the invoking object.

It can be used to access the data in the object. Also, this pointer can be used to find out the address of the calling object, to return values.

Example:

```
#include <iostream>
using namespace std;
class thisimp
{
private:
        int i;
public:
void setdata (int num)
{
    i = num;            //one way to assign data.
    this ->i = num;     //another way to assign data, remember "this" is a pointer to the calling object.
}
void showdata( )
{
  cout<< "i:" <<i<<endl ;                   //one way to display data
  cout<< " address is:" <<this<<endl ;
  cout<< " i:"<< this->i;                   //another way to display
}
} ;
```

```
int main( )
{
        emp e1, e2;
        e1.setdata(10);
        e2.setdata(30);
        e1.showdata( ) ;
        e2.showdata( ) ;
}
```

9.6 Polymorphism

Polymorphism is one of the crucial features of OOP. Polymorphism means one name, multiple forms.

Classification of Polymorphism:

1. Compile time polymorphism
2. Run time polymorphism

9.6.1 Compile Time Polymorphism

Also called **early binding** or **static binding** or **static linking**. In compile time polymorphism compiler selects the appropriate member function for particular function call at the compile time. Member functions are selected for invoking at the compile time by matching arguments, both type and number. The information regarding which function to invoke that matches a particular call is known in advance during compilation. Function overloading, constructor overloading, operator overloading, all these are examples of compile time polymorphism.

Example:

```
#include <iostream>
void area(float r) ;
void area(float l, float b) ;
void main( )
{
        float r1, l1, b1 ;
        cout<< “enter value of r1:” ; cin>>r1 ;
        cout<< “enter value of l1:” ; cin>>l1 ;
        cout<< “enter value of b1:” ; cin>>b1 ;
        cout<< “Area of circle is”<<endl; area(r1) ;
        cout<< “Area of rectangle is”<<endl ; area(l1, b1) ;
}
void area (float r)
{
        float a=3.14*r*r ;
        cout<< “Area=”<<a<<endl ;
}
void area(float l , float b)
{
        float a1=l*b ;
        cout<< “Area=”<<a1<<endl ;
}
```

In above example two functions have the same name, but question is how compile differentiates these two. Actually, C++ compiler differentiates these two by argument. In one area function, there is only one argument which is float type but in second area function there are two arguments both are float type. If any program has two functions both have same name and number of argument same, then compiler differentiates these by type of arguments.

9.6.2 Run Time Polymorphism

In **run time polymorphism the** appropriate member function is selected while program is running. It is also called **late binding** or **dynamic binding** because the appropriate function is selected dynamically at run time. This type of binding requires virtual functions and base class pointers.

9.6.3 Virtual Functions

A virtual function a member function which is declared within base class and is re-defined (Overridden) by derived class. A virtual function is a member function in base class, which is overridden in the derived class (i.e., redefined), and which tells the compiler to perform late binding on this function.

When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class. A function can be made virtual by placing the keyword virtual before its normal declaration.

When we refer to a derived class object using a pointer or a reference to the base class, we can call a virtual function for that object and execute the derived class’s version of the function.

Virtual Functions are used to support "Run time Polymorphism". When the virtual function is called by using a Base Class Pointer, the Compiler decides at Runtime which version of the function i.e. Base Class version or the overridden Derived Class version is to be called. This is called Run time Polymorphism.

Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call. The resolving of function call is done at Run-time.

Virtual functions are useful when we have number of objects of different classes but want to put them all on a single list and perform operation on them using same function call.

When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

```
class base_class
{
     …….
      ……
public:
      virtual return_type function_name(arguments)
   {
       //function body
   }
};
```

Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) an Early binding(Compile time) is done according to the type of pointer.

To achieve run time polymorphism, we use functions having same name, same number of parameters, and similar type of parameters in both base and derived classes(i.e., it requires function overriding). The function in the base class is declared as virtual using the keyword virtual. When a function in the base class is made virtual, C++ determines which function to use at run time based on the type of object pointed by the base class pointer, rather than the type of the pointer.

Some Rules for Virtual functions:

- The virtual functions must be members of some class. And they must be defined.
- They cannot be static members.
- They are accessed by using object pointers.
- A virtual function can be a friend of another class.
- We cannot have virtual constructors, but we can have virtual destructor.
- The prototypes of the virtual in the base class and the corresponding member function in the derived class must be same. If not same, then C++ treats them as overloaded functions (having same name, different arguments) thereby the virtual function mechanism is ignored.
- The base pointer can point to any type of the derived object, but vice-versa is not true i.e. the pointer to derived class object cannot be used to point the base class object.

9.7 Normal Member (*Non-virtual*) Functions Accessed with Base Class Pointers

When base class and derived classes all have functions with the same name, and we want to access these derived class functions with single base class object pointer without using virtual function.

Example:

```
class Base
 {
    public:
       void display()
       {
           cout<<" This is base "<<endl;
       }
 };
class Derv1: public Base
{
   public:
      void display()
      {
           cout<<" this is derived 1 ";
      }
 };
```

```
class Derv2: public Base
{
   public:
   void display()
   {
      cout<<" This is derived2 ";
   }
};
int main()
{
   Base * ptr;          //pointer to base class declaration.
   Derv1 D1;            //object of derived class 1.
   Derv2 D2;            //object of derived class 2.
   ptr =&D1;            //ptr points to derived class object D1.
   ptr ->display();     //calling display() function.
   ptr =&D2;            //ptr now points to derived class object D1.
   ptr ->display();     //again calling display() function.
}
```

<u>OUTPUT:</u>

```
This is base
This is base
```

As we can see, the function display() in the base class is always executed. The compiler ignores the contents of pointer ***ptr*** and always chooses the member function that matches the type of the pointer, in this case Base. So it does not implement polymorphism i.e., single name multiple form.

The reason for the this output is that the call of the function display() is being set once by the compiler as the version defined in the base class. This is called static resolution of the function call, or static linkage - the function call is fixed before the program is executed. This is also sometimes called early binding because the display() function is set during the compilation of the program.

9.8 Virtual Member Functions Accessed with Base Class Pointers

Let’s make a single change in the above program. We will place the keyword **virtual** in the declaration for the **display()** function in the base class.

Consider the following simple program which is an example of runtime polymorphism.

The main thing to note about the program is, derived class function is called using a base class pointer. The idea is, virtual functions are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime.

Run time polymorphism: Example 0

```
#include <iostream>
 using namespace std;
 class Base
 {
     public:
        virtual void display()      //virtual function.
        {
            cout<<" This is base ";
        }
 };
class Derv1: public Base
{
    public:
       void display()
       {
            cout<<" this is derived 1 "<<endl;
       }
};

class Derv2: public Base
{
    public:
    void display()
    {
        cout<<" This is derived2 ";
    }
};
```

```
int main()
{
    Base *ptr;   //base class pointer declaration.
    Derv1 D1;   //object of derived class 1.
    Derv2 D2;   //object of derived class 2.
    ptr=&D1;     //ptr points to derived class object D1.
    ptr->display(); //calling display() function, invokes member function of derived1. Why?
    ptr=&D2; //ptr now points to derived class object D1.
    ptr->display();    //again calling display() function, now invokes member function of derived2. Why?
}
```

OUTPUT:

```
This is derived 1
This is derived 2
```

Now, the member functions of the derived classes, not the base class, are executed. We change the contents of ptr from the address of Derv1 to that of Derv2, and the particular instance of display() that is executed also changes. So the same function call,

```
ptr->display();
```

executes different functions, depending on the contents of ptr(i.e., object pointed by ptr).

Here, the compiler does not know what class the contents of ptr may contain. It could be the address of an object of the Derv1 class or of the Derv2 class. Which version of display() does the compiler call? In fact the compiler does not know what to do, so it arranges for the decision to be deferred until the program is running. At runtime, when it is known what class is pointing to by ptr, the appropriate version of display() will be called, exhibiting late binding.

Compile-time(early binding) VS run-time(late binding) behavior of Virtual Functions

```
// CPP program to illustrate
// concept of Virtual Functions
#include<iostream>
using namespace std;

class base
{
public:
   virtual void print ()
   { cout<< "print base class" <<endl; }

   void show ()
   { cout<< "show base class" <<endl; }
};

class derived:public base
{
public:
   void print ()
   { cout<< "print derived class" <<endl; }

   void show ()
   { cout<< "show derived class" <<endl; }
};
 int main()
```

```
{
    base *bptr;
    derived d;
    bptr = &d;
    bptr->print();                    //virtual function, binded at runtime
    bptr->show();                     // Non-virtual function, binded at compile time
}
```

Explanation: Runtime polymorphism is achieved only through a pointer (or reference) of base class type. Also, a base class pointer can point to the objects of base class as well as to the objects of derived class. In above code, base class pointer 'bptr' contains the address of object 'd' of derived class.

Late binding(Runtime) is done in accordance with the content of pointer (i.e. location pointed to by pointer) an Early binding(Compile time) is done according to the type of pointer, since print() function is declared with virtual keyword so it will be binded at run-time (output is print derived class as pointer is pointing to object of derived class) and show() is non-virtual so it will be binded during compile time(output is show base class as pointer is of base type).

Run time polymorphism: Example1

```
#include<iostream>
#include<string>
using namespace std;
class media
{
   protected:
         string title;
         float price;
public:
media(string a, float b)
{
      title=a;
      price=b;
}
virtual void show(){  } //Empty virtual function.
};
class book: public media
{
   int pages;
   public:
   book(string a, float b, int p): media(a, b)
   {
      pages=p;
   }
   void show()
   {
      cout<<" Title: "<<title<<endl;
      cout<<" Price: "<<price<<endl;
      cout<<" Pages: "<<pages;
   }
};
```

```
class tape: public media
{
   int length;
   public:
   tape(string a, float b, int p): media(a, b)
   {
      length=p;
   }
   void show()
   {
      cout<<" Title: "<<title<<endl;
      cout<<" Price: "<<price<<endl;
      cout<<" Length: "<<length;
   }
};
int main()
{
   media *p;
   tape t(" Rihana",420.23,45);
   book b("Othelo",1080.23,456);
   p=&t;
   p->show();
   p=&b;
   p->show();
}
```

Run time polymorphism: Example2

```
class Shape
{
   protected:
      int length;
      int breadth;
   public:
      Shape(int l, int b)
      {
         length=l;
         breadth=b;
      }
      void display()
      {
         cout<<"Length: "<<length<<"\t"<<"Breadth: "<<breadth<<endl;
      }
      virtual void area()
      {
         //empty body.
      }
};

class Rectangle: public Shape //derived from base class shape
{
   public:
      Rectangle(int a, int b): Shape(a, b)
      {

      }
      void area()
      {
        cout<<"Area of rectangle: "<<length*breadth<<endl;
      }
};
class Square: public Shape //derived from base class shape
{
   public:
      Square(int a, int b): Shape(a, b)
      {

      }
      void area()
      {
         cout<<"Area of square: "<<length*breadth;
      }
};
```

```
int main()
{
   Shape*s; //base class pointer declaration.
   Rectangle R(7, 8);
   Square S(7, 7);
   R.display();
   S.display();
   s=&R;      //s points to derived class object R.
   s->area();    //invokes area() function of derived class Rectangle.
   s=&S;       //s now points to derived class object S.
   s->area();    //invokes area() function of derived class Square.
   return 0;
}
```

9.9 Pure Virtual Functions

A pure virtual function (or abstract function) in C++ is a virtual function for which don't have implementation (body), we only declare it. A pure virtual function is declared by assigning 0 in declaration. See the following example.

```
class Test                          // An abstract class, since contains pure virtual function
{
   // Data members of class
public:
        virtual void show() = 0; // Pure Virtual Function
      /* Other members */
};
```

A pure virtual function is a virtual function that has no definition within the base class (i.e. with no function body). Pure virtual function is one with the expression = 0 added to the declaration (a virtual function equated to 0). Once we have virtual function in base class, then it must be redefined in every derived class.

Example:

```
class My
{
      ……………
      …………….
         public:
           virtual void show()=0;                 //Pure virtual function.
};
```

If we do not override the pure virtual function in derived class, then derived class also becomes abstract class. The following example demonstrates the same.

```
#include<iostream>
using namespace std;
class Base
{
public:
   virtual void show() = 0;
};
 class Derived : public Base { };
 int main(void)
{
  Derived d;                  //error, since class Derived does not have function named show() overridden so
return 0;                      also an abstract class.
 }
```

9.10 Abstract Base class

A base class that has at least one pure virtual function is called abstract base class. It designed only to act as a base class. An abstract base class cannot be used to create objects. But we can define base class pointers. It provides only an interface for its derived classes. The main objective of an abstract base class is to provide some attributes to the derived class and to create a base class pointer required for run time polymorphism.

Example:

```
class A
{
      protected:
            int data;
      public:
           A(){}
           A(int d)
           {
                 data = d;
           }
virtual void show() = 0; //pure virtual function
};
class B : public A       //derived class B.
{
public:
     B(int d) : A(d)
       {
           //empty body.
        }
 void show()
  {
       cout<<data<<endl;
   }
};
```

```
class C : public A   //derived class C.
 {
    public:
       C(int d) : A(d)
         {
             //empty body.
         }
  void show()
       {
             cout<<data;
       }
};
int main()
{
       A *a;
       A z;   // error (Why?)
       B b(5);
       C c(6);
      a = &b; a->show();
      a = &c; a->show();
}
```

Another Example:

```
class Shape
{
   protected:
      int length;
      int breadth;
   public:
      Shape(int l, int b)
      {
         length=l;
         breadth=b;
      }
      void display()
      {
         cout<<"Length: "<<length<<"\t"<<"Breadth: "<<breadth<<endl;
      }
      virtual void area()=0;      //pure virtual function.
};
class Rectangle: public Shape //derived from base class shape
{
   public:
      Rectangle(int a, int b): Shape(a, b)
      {
                  //empty body
      }
      void area()
      {
         cout<<"Area of rectangle: "<<
         length*breadth<<endl;
      }
};
class Square: public Shape     //derived from base class shape.
{
   public:
      Square(int a, int b): Shape(a, b)
      {
                  //empty body
      }
      void area()
      {
         cout<<"Area of square: "<<length*breadth;
      }
};
```

```
int main()
{
    Shape*s[2]; //array of base class pointer declaration.
    Rectangle R(7, 8);
    Square S(7, 7);
    R.display();
    S.display();

    s[0]=&R;  //s[0] points to derived class object R.
    s[0]->area(); //invokes area() function of derived class Rectangle.

    s[1]=&S;  //s[1] points to derived class object S.
    s[1]->area(); //invokes area() function of derived class Square.
    return 0;
}
```

9.11 Abstract vs. Concrete Classes

Concrete means "existing in reality or in real experience; perceptible by the senses; real", whereas abstract means "Not applied or practical; theoretical.

A concrete class is a class that can be used to create an object. An abstract class cannot be used to create an object.

An abstract class is one that has one or more pure virtual function. Whereas a concrete class has no pure virtual functions. A base class can be either abstract or concrete and a derived class can also be either abstract or concrete.

Example

```
class Alpha
{
        ………
        ………
        virtual void draw()=0;
        ………
        ………
};
```

```
class Beta
{
        ………
        ………
        void draw();
        ………
        ………
};
```

In above example, class Alpha is abstract class and class Beta is concrete class. Concrete classes do not have pure virtual functions.

9.12 Early & Late Binding

The differences between early and late binding are given below:

Early Binding	Late Binding
1. It is also known as compile time polymorphism. It is called so because compiler selects the appropriate member function for particular function call at the compile time.	1. It is also known as run time polymorphism. It is called so because the appropriate member functions are selected while the program is executing or running.
2. The information regarding which function to invoke that matches a particular call is known in advance during compilation. That is why it is also called as early binding.	2. The compiler doesn't know which function to bind with particular function call until program is executed so it is also called late binding.
3. The function call is linked with particular function at compiler time statically. So, it is also called static binding.	3. The selection of appropriate function is done dynamically at run time, so it is also called dynamic binding.
4. This type of binding can be achieved using function overloading and operator overloading.	4. This type of binding is achieved using virtual function and base class pointer.

Chapter-10

Templates

10.1 Introduction

Templates are a relatively new feature of the C++ programming language that allows functions and classes to operate with generic types. This allows a function or class to work on many different data types without being rewritten for each one. Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type. A template is a blueprint or formula for creating a generic class or a function.

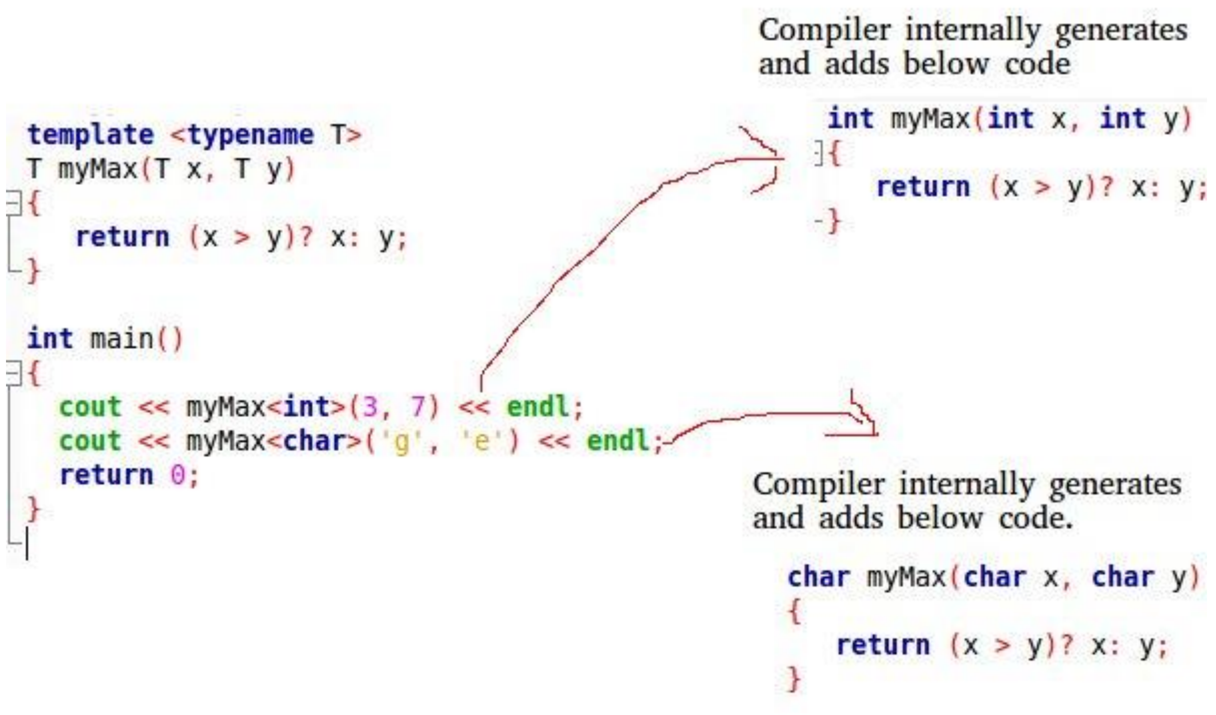
A template can be considered as a macro which helps to create a family of classes or functions. When an object of a specific type is defined for actual use, the template definition for that class is substituted with the required data type. Since a template is defined with a parameter that would be replaced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

Template is simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write same code for different data types. For example a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

C++ adds two new keywords to support templates: 'template' and 'typename'. The second keyword can always be replaced by keyword 'class'.

How templates work?

Templates are expended at compiler time. This is like macros. The difference is, compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of same function/class.



```
template <typename T>
T myMax(T x, T y)
{
    return (x > y)? x: y;
}

int main()
{
    cout << myMax<int>(3, 7) << endl;
    cout << myMax<char>('g', 'e') << endl;
    return 0;
}
```

Compiler internally generates and adds below code

```
int myMax(int x, int y)
{
    return (x > y)? x: y;
}
```

Compiler internally generates and adds below code.

```
char myMax(char x, char y)
{
    return (x > y)? x: y;
}
```

For examples:

- ❖ A class template for an array class enables us to create arrays of various types such as an integer array, a float array, a char array etc.
- ❖ Similarly, we can define a template for a function, say mul( ), that would help us to create various versions of mul( ) for multiplying int, float, double etc.

Advantages

- ❖ Template classes and functions eliminate code duplication for different types and thus make the program development easier and more manageable.

There are two types of templates:

1. Function Templates
2. Class Templates

10.2 Function Templates

Function Template is a way of writing a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray(). A function template behaves like a function except that the template can have arguments of many different types. In other words, a function template is a way of creating a family of functions with different argument types.

A specific function created from a function template is called a *template function*.

The general syntax of a function template:

```
template <class T>
return_type function_name (arguments of type T)
{
        // Body of function
        // with type T
        // wherever appropriate
}
```

OR

```
template < typename T>
return_type function_name (arguments of type T)
{
        // Body of function
        // with type T
        // wherever appropriate
}
```

This syntax shows that the function template definition is very similar to an ordinary function definition except the use of prefix template <class T> and use of type T. This prefix tells the compiler that we are going to declare a template and use "T" as a generic type name in the declaration. This T can be replaced by any built-in data type (int, float, char, etc) or a user-defined data type.

NOTE: The typename and class keywords can be used interchangeably to state that a template parameter is a type variable.

Calling Function Templates:

Function_template_name<type*1*,type*2*,…,type*n*> (Args.);   //Here, type can be int, float, char etc.

Or

Function_template_name (Args.);

Example1: The following program declares a mul() function template that will multiply two values of a given type.

```
#include <iostream>
using namespace std;
template <class T>          //function template with one generic type T.
T mul(T x, T y)             //remember here return type is generic type T.
{
        return x*y;
}
```

```
int main( )
{
cout<<" Result for int type: "<< mul <int>(4, 5)<<endl;       //creating int version of mul(),
                                                              generic type T is replaced by int at compile time.

cout<<" Result for float type: "<< mul(7.6, 88.3);            //creating float version of mul(),generic
                                                              type T is replaced by int at compile time.
        return 0 ;
}
```

When the compiler sees a function call mul(4, 5), it knows the type to use is *int*, because that is the type of the arguments passed. So, it generates a specific version of the mul() function for type *int*, substituting *int* whenever it sees the name T in the function template. This is called *instantiating* the function template, and each instantiated version of the function is called a ***template function***. That is template function is a specific version/instance of the function template.

Example2: The following program declares a max() function template that will find maximum value.

```
#include <iostream>
using namespace std;
template <class X>
X maxi(X a, X b)
{
        if (a<b)
                return b;
        else
                return a;
}
int main()
{
        cout<<"Max of integer: "<<maxi(4, 5)<<endl;              //displays 5.
        cout<<"Max of character: "<<maxi('f', 'e')<<endl;        //displays f.
        cout<<"Max of string: "<<maxi("xahul", "Xahul")<<endl;   //displays xahul.
        cout<<"Max of float: "<<maxi(4.2, 2.01);
        return 0;
}
```

Example2: The following program shows how to declare multiple generic types in a single function template.

```
#include <iostream>
using namespace std;
template <class T, class M>   //function template with two generic types
void Output(T x, M y)
{
   cout<<"x: "<<x<<endl;
   cout<<"y: "<<y<<endl;
}
int main( )
{
         cout<<" Result for one string and one int type: "<< endl;
                  Output("ram", 47);      //Generic type T is replaced by string, and M is replaced by int.

         cout<<" Result for two float types: "<<endl;
          Output(7.6, 88.3);              //Generic type T is replaced by float, and M is replaced by floatt.

         cout<<" Result for one char and one string type: "<<endl;
                   Output('c', "hello world"); //Generic type T is replaced by char, and M is replaced by string.
         return 0 ;
}
```

10.3 Overloading of function template

Just like normal functions we can overload function templates. Process of defining more than one function template with same name is called template function overloading.

Example:

```
#include <iostream>
using namespace std;
template <class T>
T mul(T x, T y)
{
         return x*y;
}
template <class T, class S>
T mul(T x, T y, S z)
{
         return x*y*z;
}
```

Here, we have defined two function templates with same name *mul*, one has one generic type T, and the other has two generic type T, and S.

Hence, in this example function template mul is overloaded.

```
int main( )
{
        cout<<" Two argument function template: "<< mul(4, 5)<<endl;
        cout<<" Three argument function template: "<< mul(7.6, 88.3, 2);
        return 0 ;
}
```

10.4 Class template

Class template is a template that helps us to create generic classes. The relationship between a class template and an individual class is like the relationship between a class and an individual object. An individual class defines how a group of objects can be constructed, while a class template defines how a group of classes can be generated.

The general syntax of a class template is:

```
template <class T>
class class_name
{
        // class member specification
        // with type T
        // wherever appropriate
} ;
```

OR

```
template < typename T>
class class_name
{
        // class member specification
        OR
        // with type T
        // wherever appropriate
} ;
```

This syntax shows that the class template definition is very similar to an ordinary class definition except the use of prefix template <class T> and use of type T. This prefix tells the compiler that we are going to declare a template and use "T" as a type name in the declaration. This T can be replaced by any built-in data type (int, float, char, etc) or a user-defined data type. A specific class created from a class template is called a template class.

The syntax for defining an object of a class template is:

```
template_class_name <type1, type2…typen> object_name(arglist);       //if constructor is used to
                                                                     initialize object.

template_class_name <type1, type2,….,typen> object_name;             //if member function is used to
                                                                     initialize object.
```

Here type indicates any data type (int, float, char, or any user defined type).

Example1:

```
template <class T>

class test

{

                T a, b;

        public:

                void getdata( )

                {

                        cout<<" Enter any thing: "<<endl;

                        cin>>a>>b ;

                }

                void putdata( )

                {

                        cout<< " You entered: "<<a<< " "<<b<<endl;

                }

};

int main( )

{

        test<int>t1 ;           //defining an object with int type data member.

        t1.getdata( );

        t1.putdata( );


        test<float>t2 ;         //defining an object with float type data member.

        t2.getdata( );

        t2.putdata( );

        test<char>t3 ;          //defining an object with char type data member.


        t3.getdata( );

        t3.putdata( );

        return 0 ;

}
```

Example2:

```
#include <iostream>
using namespace std;

template <class T>

class temp

{

                T a[10];

        public:

                void getdata()

                {

                        cout<<"Enter data: "<<endl;

                        for(int i=0; i<10;i++)

                        {

                                cin>>a[i];

                        }
                }
                void display()

                {

                        cout<<"Your data: "<<endl;

                        for(int i=0; i<10;i++)

                        {

                                cout<<a[i];

                        }
                }
};
int main()

{

        temp<int>t;             //defining an object with int type data member.

        temp<char>s;            //defining an object with char type data member.

        temp<float>p;           //defining an object with floata type data member.

        t.getdata();

        t.display();

        s.getdata();

        s.display();

        p.getdata();

        p.display();

        return 0;

}
```

Example3:

```
#include <iostream>
using namespace std;
template <class T, class M>                    //class template with two generic types T and M.
class test
{
        T a_data;               //member of generic type T.
        M b_data;               //member of generic type M.
public:
test(T a,  M b)                 //constructor
{
        a_data = a;
         b_data = b;
}
void putdata( )
{
        cout<< " You entered: "<<a_data<< " "<<b_data<<endl;
}
};
int main( )
{
   test<int, float>t1(4, 5.5) ;    //defining an object with one int type and one float type data member.
   t1.putdata();

   test<float, string>t2(4.44, "ram") ; //defining an object with one float type &one string type data member.
   t2.putdata();

   test<char, int>t3('c', 456) ;           //defining an object with one char and one int type data member.
   t3.putdata();
   return 0 ;
}
```

//Operator Overloading Using Class Template

```
template <class S>
class Test
{
   S Somedata;
public:
   Test()
   {
      Somedata= " ";
   }
   Test(S l)
   {
      Somedata=l;
   }
S operator +(Test T1)
{
   return (T1.Somedata+Somedata);
}
void display()
{
   cout<<"Length: "<<Somedata<<endl;
}
};
int main( )
{
   Test<int> g(2);             //Creating an object that hold type int as a data member, generic type S is
                               replaced by built in type int
   Test<int> h(77);            //Creating an object that hold type int as a data member
   Test<int>I=g+h;             //Calling operator function, g is the calling object.
   I.display();
```

```
    Test <string>A(" Class Template");          //Creating an object that hold type string as a data member
    Test <string>B(" Beauty of");               //Creating an object that hold type string as a data member
    Test<string> C=A+B;                         //Calling operator function, A is the calling object.
    C.display();
    return 0 ;
}
```

Here, + operator is overloaded using one generic type S, and is replaced by types *int*, and *string* at compile time. Thus, class template allows us to create different versions of data members, as well as class member functions.

Exercise: WAP to overload < operator using class template for string, char, int and float.

Chapter-11

Exceptions

11.1 Introduction

The most common types of error (also known as bugs) occurred while programming in C++ are Logic error and Syntactic error. The logic errors occur due to poor understanding of the problem and solution procedure. The syntactic errors arise due to poor understanding of the language. These errors are detected by using exhaustive debugging and testing.

There are some problems other than logic or syntax errors. They are known as exceptions.

- These are basically runtime errors.
- Exceptions are runtime anomalies or unusual conditions that a program may encounter while in execution.
- Exceptions might include conditions such as division by zero, access to an array outside of its bounds, running out of memory or disk space, not being able to open a file, trying to initialize an object to an impossible value etc.
- When a program encounters an exceptional condition, it is important that it is identified and dealt with effectively.
- C++ provides built-in language features to detect and handle exceptions.

Why Exception Handling?

The purpose of the exception handling mechanism is to provide means to detect and report an "exceptional circumstance" so that appropriate action can be taken.

Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try, and catch blocks, the code for error handling becomes separate from the normal flow.

11.2 Exception Handling Mechanism

The mechanism suggests a separate error handling code that performs the following tasks:

- ❖ Find the problem (Hit the exception/try).
- ❖ Inform that an error has occurred (throw the exception).
- ❖ Receive the error information (catch the exception).
- ❖ Take corrective action (Handle the exception).

The error handling code basically consists to two segments, one to detect errors and to throw exceptions, and the other to catch the exceptions and to take appropriate actions.

C++ exception handling is built upon three keywords: try, catch, and throw.

a) try

b) throw

c) catch

- ❖ **try:** The keyword "try" is used to refer a block of statements surrounded by braces which may generate exceptions. This block of statement is known as **try** block. A try block identifies a block of code for which particular exceptions will be activated. Try block is intended to throw exceptions, which is followed by catch blocks. Only one try block. It's followed by one or more catch blocks.

- ❖ **throw:** A program throws an exception when a problem shows up. This is done using a throw keyword. When an exception is detected, it is thrown using a ***throw*** statement in the try block or in functions that are invoked from within the try block. This is called throwing an exception and the point at which the throw is executed is called the ***throw point***. A throw expression accepts one parameter and that parameter is passed to handler.

- ❖ **catch:** Represents a block of code that is executed when a particular exception is thrown. A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The ***catch*** keyword indicates the catching of an exception. A ***catch*** block defined by the keyword **catch** catches the exception thrown by the **throw** statement in the try block, and handles it appropriately. Catch block is intended to catch the error and handle the exception condition. We can have multiple catch blocks.

The general syntax for handling exception is:

```
- - - -
try
{
        - - - -
        - - - -                         // block of statement which
       throw exception;                 //detects and throws an exception
        - - - -
        - - - -
}
catch(type arg)                         //catches exception
{
        - - - -
        - - - -                         // block of statements that handles the exception.
        - - - -
}
```

Figure below shows try-catch relationship.

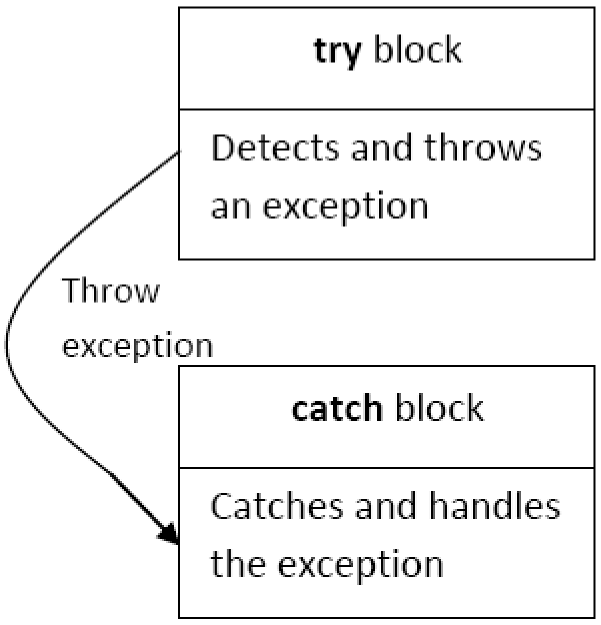

try block
Detects and throws an exception
Throw exception
catch block
Catches and handles the exception

Fig: The block throwing exception

Throwing Mechanism:

The exception is thrown by the use of throw statement in one of the following ways:

```
throw(exception) ;
throw exception ;
```

The object "exception" may be of any type or a constant. We can also throw an object not intended for error handling.

Catching Mechanism:

The catch block must immediately follow the try block that throws the exception. Code for handling exceptions is included in catch blocks. A catch block looks like a function definition and is of the form:

```
catch(type arg)
{
        // statements for
        // managing exceptions
}
```

The "type" indicates the type of exception that **catch** block handles. The parameter arg is an optional parameter name. The exception_handling code is placed between two braces. The catch statement catches an exception whose type matches with the type of catch argument. When it is caught, the code in the catch block is executed.

We can define a single catch, to handle all types of exceptions. It has the following form:

```
catch(…)                //to handle all types of exceptions.
{
        // statements for
        // managing exceptions
}
```

The figure below shows the exception handling mechanism, here try block invokes the function that contains the throw statement.

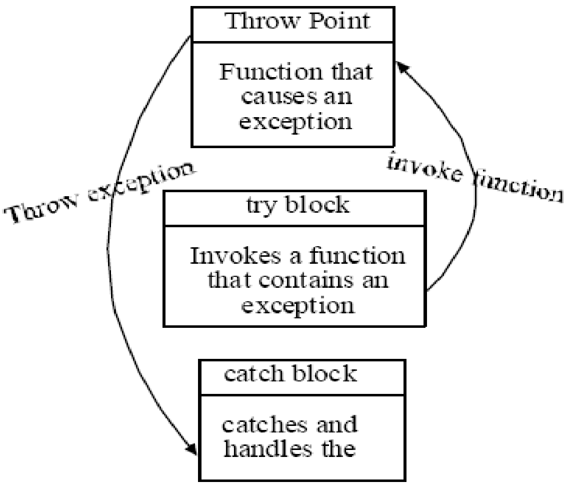


Fig : Function invoked by try block throwing exception

When the **try** block throws an exception, the program control leaves the **try** block and enters the **catch** statement of the **catch** block. Exceptions are objects, which are used to transmit information about a problem. If the type of object *thrown* matches the *arg type* in the **catch** statement, then catch block is executed for handling the execution. If they do not match, the program is aborted with the help of *abort()* function which is invoked by default.

When no exception is detected and thrown, catch block is skipped.

Example 1: Try block throwing exception

```
#include <iostream>
using namespace std;
int main()
{
    float a, b;
    cout<<"Enter values of a & b:\n";
    cin>>a>>b;
try
{
            if(b == 0)
                         throw b;
             else
                        cout<<"Result = "<<a/b;
}
catch(float)
   {
        cout<<"Divide by zero exception:= "<<endl;
   }
  return 0;
}
```

Example2: Function invoked by try block throwing exception

```
#include <iostream>
using namespace std;
void divide(int a, int b)
{
        if(b == 0)
                throw b;
        else
                cout<<"Result = "<<(float)a/b;
}
int main()
{
        int a, b;
        cout<<"Enter values of a & b:\n";
        cin>>a>>b;
        try
        {
                divide(a, b);           //invokes a function.
        }
        catch(int i)
        {
                cout<<"Divide by zero exception: b = "<<i;
        }
        return 0;
}
```

In the first example, if exception occurs in the try block, it is thrown and the program control leaves from the try block and enters the catch block. In the second example, try block invokes the function divide(). If exception occurs in this function, it is thrown and control leaves this function and enters the catch block.

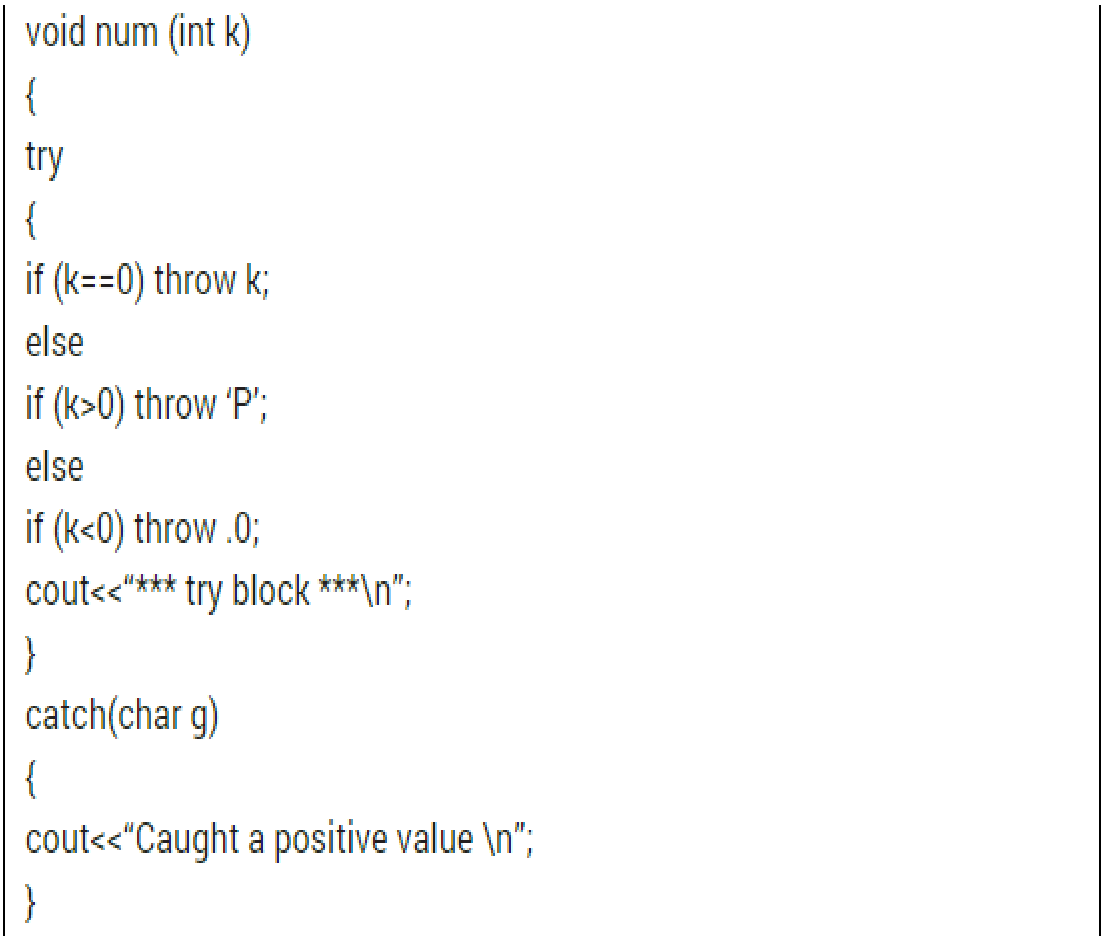
11.3 Multiple catch statements:

We can also define multiple catch blocks; in the try block, such programs also contain multiple throw statements based on certain conditions. The format of multiple catch statements is as follows:

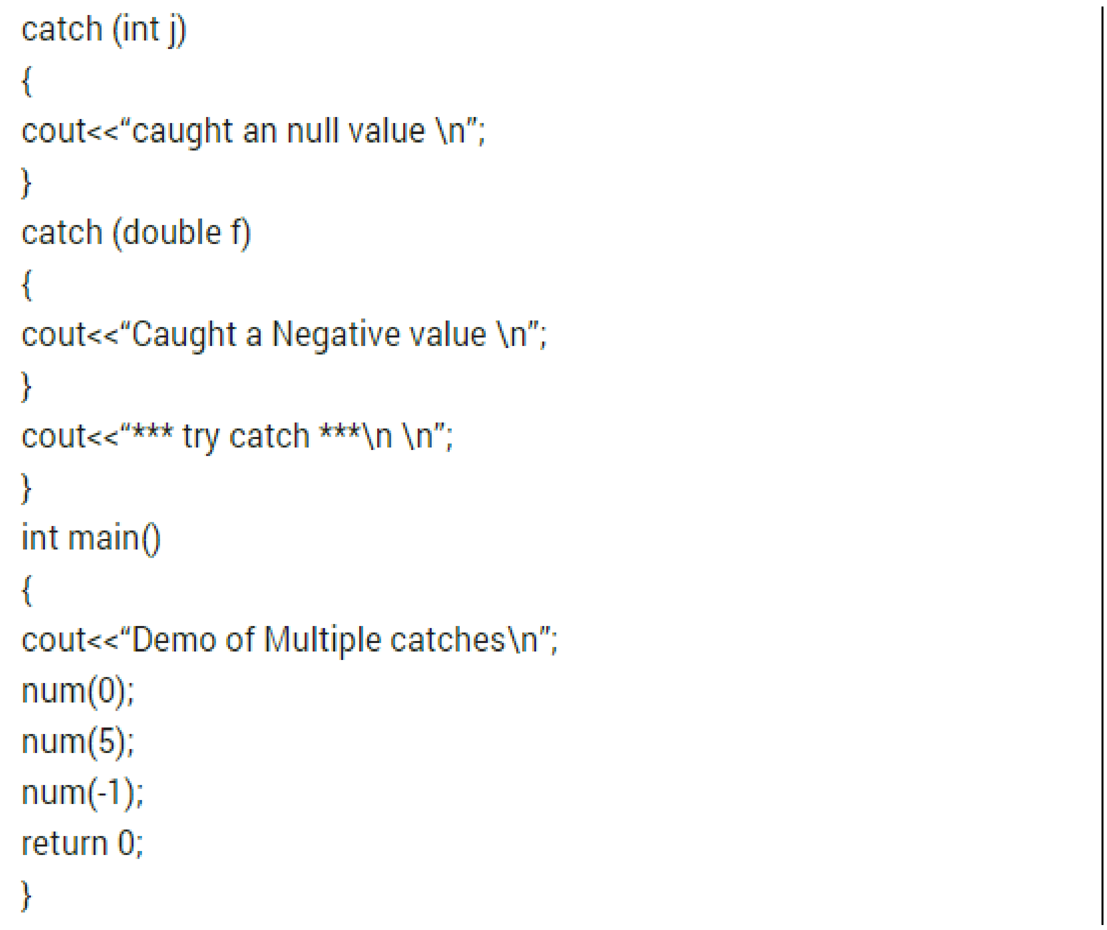
```
try
{
 // try section
}
catch (object1)
{
 // catch section1
}
catch (object2)
{
 // catch section2
}
. . . . . . .
. . . . . . .
catch (type n object)
{
 // catch section-n
}
```

As soon as an exception is thrown, the compiler searches for an appropriate matching catch block. The matching catch block is executed, and control passes to the successive statement after the last catch block. In case no match is found, the program is terminated. In a multiplecatch statement, if objects of many catch statements are similar to the type of an exception, in such a situation, the first catch block that matches is executed.

Consider the following example.



```
void num (int k)
{
try
{
if (k==0) throw k;
else
if (k>0) throw 'P';
else
if (k<0) throw .0;
cout<<"*** try block ***\n";
}
catch(char g)
{
cout<<"Caught a positive value \n";
}
```



```
catch (int j)
{
cout<<"caught an null value \n";
}
catch (double f)
{
cout<<"Caught a Negative value \n";
}
cout<<"*** try catch ***\n \n";
}
int main()
{
cout<<"Demo of Multiple catches\n";
num(0);
num(5);
num(-1);
return 0;
}
```

Chapter-12

File Handling in C++

12.1 C++ Class Hierarchy

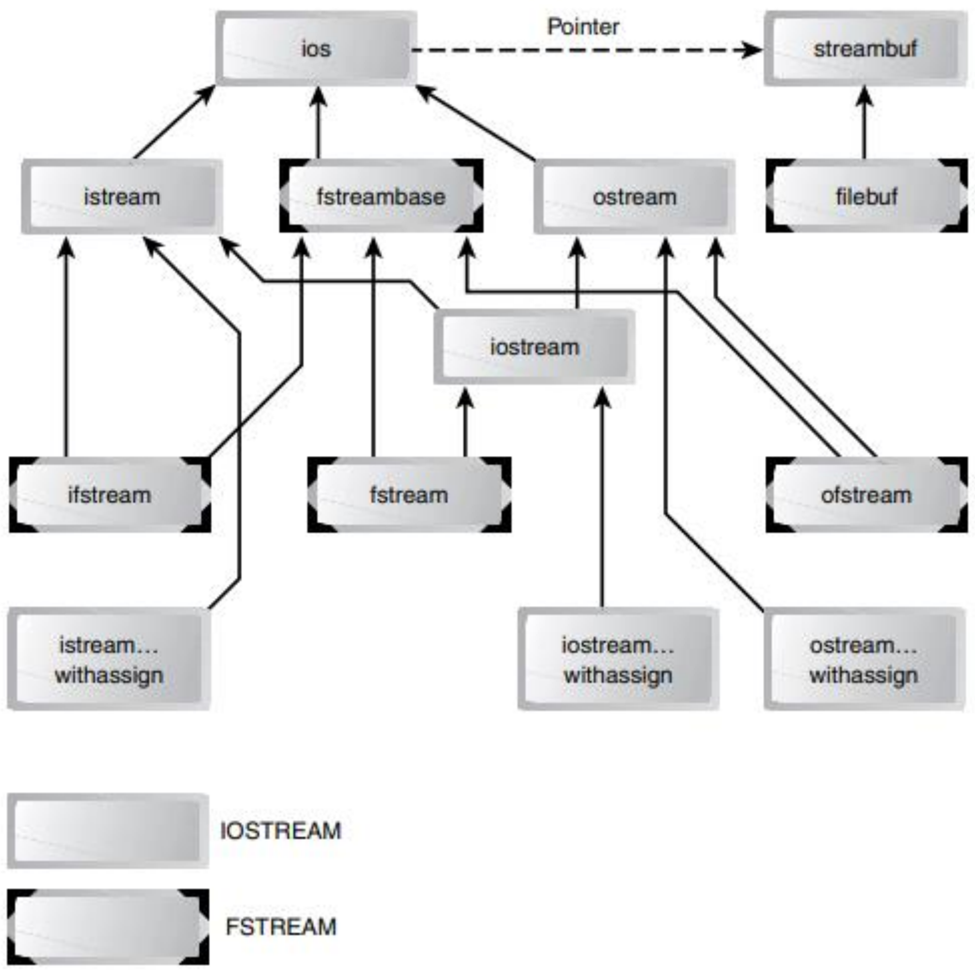


Figure: The Stream Class Hierarchy

The stream classes are arranged in a rather complex hierarchy. Figure above shows the arrangement of the most important of these classes.

12.2 File I/O

12.2.1 Introduction

The I/O system of C++ handles file operations which are very much similar to the console input and output operations. A file stream is an interface between the programs and the files. The stream which supplies data to the program is called **input stream** and that which receives data from the program is called **output stream**. That is, the input stream reads or receives data from the file and supplies it to program while the output stream writes or inserts data to the file. This is illustrated in the figure.

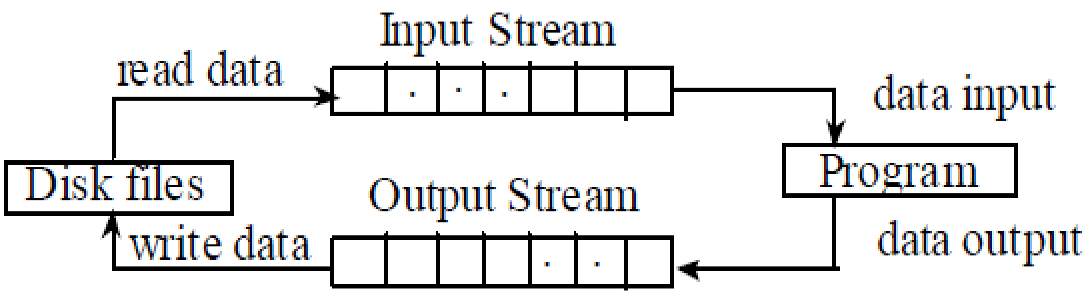


Fig : File Input and Output Stream

12.2.2 Classes for File Stream Operations

The I/O system of C++ contains several stream classes for handling file I/O operations:

❖ ***ifstream***(input file stream) class provides input operations on files. This class represents the input file stream and is used to read information from files. It contains open( ) with default input mode. Inherits the function get( ), getline( ), read( ), seekg( ) and tellg( ) function from istream class.

❖ ***ofstream****(output file stream)* class provides output operations on files. This class represents the output file stream and is used to create files and to write information to files. It Contains open( ) with default output mode. Inherits put( ), seekp( ), tellp( ) and write( ) function from ostream class.

❖ ***fstream****(file stream)* class supports for simultaneous input and output operations on files. This class represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files. Inherits all the functions from istream and ostream classes.

Summary:

❖ ofstream: Stream class to write on files

❖ ifstream: Stream class to read from files

❖ fstream: Stream class to both read and write from/to files.

12.3 General File I/O Steps

1. Declare a file stream object using **ifstream, ofstream or fstream**. For reading only: **ifstream** is used, for writing only**: ofstream** is used, and for simultaneous reading-writing**: fstream** is used.
2. Connect the file stream object with the file name.
3. Open the file.
4. Use the file: **read/write or both**.
5. Close the file.

12.4 File Operations

There are different types of operations that can be performed on a file, like opening, reading, writing, closing, etc. A file stream can be defined using the classes **ifstream**, **ofstream**, and **fstream** that are contained in the header file **fstream**.

To perform read-write operations in the file, first we need to open that file. A file can be opened in two ways:

- **Using the constructor of the class**: suitable when we use only one file in the stream.
- **Using the member function open( ):** suitable when we want to manage multiple files using one stream.

12.4.1 Opening files Using Constructor

This method is used when we use only one file in a stream. While opening the file using constructor, we need to pass the desired filename as a parameter to the constructor. This involves following steps:

1. Create a file stream object for read or write operation.
2. Initialize the file object with desired filename in appropriate mode.

Syntax:

```
file_stream_class object_name(“file_name”, mode);
```

Here file_stream_class can be ifstream, ofstream, or fstream.

For example:

```
ofstream fout(“results.txt”) ;            //output only
```

This statement creates an object fout (which can be any valid name like myfile, outfile, o_file, etc) of ofstream class and attaches this object with a file “results.txt” which can be any valid name. In this results file, we can only write data because it is defined using **ofstream** class.

Similarly, a file can be opened for input or reading as,

```
ifstream fin(“test”) ;                    //input only
```

This creates an object fin(it may be any valid name like infile, myfile, etc) of ifstream and attaches a file “test” with it. In file “test” we can perform only input operation i.e. we can only read data because it is created using **ifstream** class.

Your Very First Program

The first program, will create a file, and write some text into it.

```
#include <fstream>
using namespace std;
int main()
{                                          // ofstream constructor opens file.
        ofstream fout("cpp.txt");      //creating output file stream object fout and connecting it to cpp.txt.
        fout << "Hello World, from Sagarmatha College!";        //writing to the file, same as cout.
        fout.close();    //closing file after writing.
        return 0;
}
```

Program Description:

This program will create the file cpp.txt in the directory from where you are executing it, and will put "Hello World, from Sagarmatha College!" into it.

- #include <fstream> - We need to include this file in order to use C++'s functions for File I/O. In this file, there are several classes, including ifstream, ofstream and fstream, which are all derived from istream and ostream.
- ofstream fout("cpp.txt"); : This statement creates an object fout (*which can be any valid name like myfile, outfile,o_file, etc*) of ofstream class and attaches this object with a file "cpp.txt" which can be any valid name. In this "cpp.txt" file, we can only write data because it is created by ofstream class. A filename is used to initialize the file stream object as fout.
- fout.close(); : As we have opened the stream, when we finish using it, we have to close it. fout is an object from class ofstream, and this class (ofstream) has a function that closes the stream. That is the close() function. So, we just write the name of the object, dot operator and close(), in order to close the file stream.

File Modes:

- File modes specifies how file will be opened. Different file opening modes in C++ are as follows:

Modes	Functions
ios: :app	Start reading or writing at end of file (Append)
ios: : in	Open file for reading only (default for ifstream)
ios: : out	open file for writing only (default for ofstream)
ios: : binary	open file in binary (not text) mode
ios :: ate	go to end of file on opening
ios :: trunc	If file is present erases its content, otherwise creates a new file.

We can combine two or more modess by **bitwise OR (|)**ing them together. For example if we want to open a file in write mode and want to truncate it in case it already exists, following will be the syntax:

```
ofstream outfile("file.dat", ios::out | ios::trunc );
```

Similarly, following statement will open a file student.dat in read, and binary mode.

```
ifstream file(("student.dat", ios::in | ios::binary);
```

Reading from file:

Sample Program0:

```
#include<fstream>
#include<iostream>
#include<string>
using namespace std;
int main()
{
   ifstream fin("data.txt");     //file is opened using ifstream class so can perform read operation only.
if(!fin)             //check for file.
{
   cout << "File could not found" << endl;
}
string item;
while(!fin.eof())                //read from file until end of file.
{
         fin>>item;              //Remember it cannot read whitespaces.
         cout<<item;           //displays the word to the monitor.
}
fin.close();
return 0;
}
```

To read one entire line with embedded blanks use following:

```
ifstream fin("data.txt");
while(!fin.eof())                //read from file until end of file.
{
         getline(fin, item);      //Can read one line of text with whitespaces.
         cout<<item;           //displays on the screen.
}
```

To read multiple lines with embedded blanks use following:

```
ifstream fin("data.txt");
while(!fin.eof())                  //reads the file until end of file.
{
        getline(fin,item,'$');
        cout<<item;              //displays on the screen.
}
```

Sample Program1:

```
using namespace std;
int main()
{
   ifstream fin("kk.txt");    //file is opened using ifstream object so can perform read operation only.
if(!fin)                          //check for file.
{
   cout << "File could not found" << endl;
}
string item;
while(!fin.eof())                 //read till end of file.
{
        fin>>item;               //read one word at a time from file and assigns it to item.
        cout<<item<<endl;          //displays the string on the screen.
}
fin.close();
return 0;
}
```

12.4.2 Opening a file using open( ) function

The open( ) function can also be used to open files.

General syntax:

```
        file_stream_class stream_object_name;
        stream_object_name.open ("filename", mode);
```

The second argument (called file mode parameter) specifies the purpose for which the file is opened. The mode can combine two or more parameters using the bitwise OR operator (|).

Examples:

```
1)      fstream file ;
        file.open("Person.Dat", ios: : app|ios: : out|ios: : in);
```

It creates a Person.Dat file for input and output in append mode.

```
2)      ifstream infile ;
        infile.open("Test.txt", ios: : in) ;  //This opens Test.txt file for reading only.
```

```
3)      ofstream outfile ;
        outfile.open("Test.txt", ios: : out) ; //This opens Test.txt for writing only.
```

Example:

```
#include <fstream>
#include<iostream>
using namespace std;
int main()
{
    string data, rdata;
    cout<<"Enter a string: "<<endl;
    getline(cin, data)
     ofstream fout;                    //can perform write operation.
    fout.open("String.txt", ios::out|ios::app);
    fout<<data<<endl;
    fout<<"Well done!! You entered some text";
    fout.close();
    ifstream fin;
   fin.open("String.txt", ios::in)
    while(!fin.eof())
    {
        getline(fin, rdata);   //read a line from file and assigns it to rdata.
        cout<<rdata;           //displaying it on screen.
    }
   fin.close();
    return 0;
}
```

12.5 File Pointers and Their Manipulations

Each file has two automatic 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**). When input and output operation takes place, the appropriate pointer is automatically set according to mode. For example when we open a file in reading mode file pointer is automatically set to start of file. And when we open in append mode the file pointer is automatically set at the end of file.

In C++ there are some manipulators by which we can control the movement of pointer. The available manipulators in C++ are:

1. seekg( )
2. seekp( )
3. tellg( )
4. tellp( )

Following section shows their details:

1. **seekg(n):** Moves get pointer(input) to a specified location from the start.
    a. **fileObject.seekg(0)**: Goes to the start of the file for reading.
    b. **fileObject.seekg(n)**: Goes to nth byte from beginning for reading.
2. **seekp(n):** Moves put pointer(output) to a specified location from the start.
    a. **fileObject.seekp(0)**: Goes to the start of the file for writing.
    b. **fileObject.seekp(n)**: Goes to nth byte from beginning for writing.
3. **tellg():** Gives the current position of the get pointer. It returns an integer value that specifies the current location of the get pointer.
    a. int p = fileObject.tellg();   **//returned value is assigned to p.**
4. **tellp():** Gives the current position of the put pointer. It also returns an integer value that specifies the current location of put pointer.
    a. int p = fileObject.tellp();   **//returned value is assigned to p.**

Consider following Example:

```
int main()
{
        ifstream fin;
        fin.open("cpp.txt");    //open already created file for reading.
        fin.seekg(4);           //moving get pointer to the 4th position in the file
        string data;
        while(!fin.eof())       //it will read all the data after 4th position in the file.
        {
                fin>> data;
                cout<<data<<endl;
        }
        fin.close();
}
```

Consider the following statements:

```
ifstream fin;
        fin.open("cpp.txt");    //while opening, get pointer is at the start of the file automatically.
        int p=fin.tellg();      //determines the current position of the get pointer, i.e., 0.
        cout<<" current position of the get pointer is: "<<p<<endl;         //displays 0.
        fin.seekg(4);                   //now get pointer is moved to 4th position.
        p=fin.tellg();                  //determines the current position of the get pointer, i.e., 4.
cout<<" current position of pointer is: "<<p<<endl<<endl;                   //displays 4.
```

<u>**Try this, analyze the output:**</u>

```
int main()
{
        ofstream file;
        file.open("test.txt", ios::out);
        int p=file.tellp();
        cout<<" current position of put pointer is: "<<p<<endl;     //displays 0.
        file<<"C++ is better than C ";//writing some text to file.
        cout<<"current position of put pointer is: "<<file.tellp()<<endl;
        file.seekp(4);          //moving put pointer to 4th position.
        file<<"hello world ";           //writing some text again.
        cout<<" current position of pointer is: "<<file.tellp()<<endl;
        file.close();
        string text;
        ifstream fin;
        fin.open("test.txt", ios::in);
        while(!fin.eof())               //reading file until, end of file.
        {
                fin>>text;
                cout<<text;
        }
        fin.close();
        return 0;
}
```

Specifying the Reading and Writing Direction

The other form for seekg(), and seekp()functions:

seekg (offset, direction);

seekp (offset, direction);

Direction can be:

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

For Example:

FileObject.seekg(10,ios::beg): Go 10 characters forward from the start(for reading).

FileObject.seekg(-5,ios::end): Go 5 characters backward from the end(for reading).

FileObject.seekg(12,ios::cur): Go 12 characters forward from the current position(for reading).

FileObject.seekp(10,ios::beg): Go 10 characters forward from the start of the file(for writing).

FileObject.seekp(-5,ios::end): Go 5 characters backward from the end(for writing).

FileObject.seekp(-6,ios::cur): Go 6 characters backward from the current position (for writing).

12.6 Stream Errors

Errors that occur during input(read)/output(write) are called stream errors. **The stream error-status flags** are used to report errors that occurred in an input or output operation.

Error-Status Flags

Name	*Meaning*
`goodbit`	No errors (no flags set, value = 0)
`eofbit`	Reached end of file
`failbit`	Operation failed (user error, premature EOF)
`badbit`	Invalid operation (no associated streambuf)
`hardfail`	Unrecoverable error

Functions for Error Flags

Function	*Purpose*
`int = eof();`	Returns true if EOF flag set
`int = fail();`	Returns true if `failbit` or `badbit` or `hardfail` flag set
`int = bad();`	Returns true if `badbit` or `hardfail` flag set
`int = good();`	Returns true if everything OK; no flags set
`clear(int=0);`	With no argument, clears all error bits; otherwise sets specified flags, as in `clear(ios::failbit)`

12.7 Reading and Writing a Class Object (Object I/O)

The binary input and output functions read() and write() are can be used for reading and writing a class object into the file. These functions handle the entire structure of an object as a single unit. But only data members can be written to the disk file, and member functions are not.

Syntax of read(), and write() functions:

- *stream_object_name.read((char*) &V, sizeof(V));* ***//to read from file***
- *stream_object_name.write(char*) &V, sizeof(V));* ***//to write into file***

These functions take two arguments. The first is the address of the variable or object V, and the second is the length of that variable in bytes. The address of the variable must be cast to type char*(i.e., pointer to character type.).

Following programs shows, how class objects can be written to/and read from the file. The length of the object is obtained by using the **sizeof()** operator.

Example Program for Writing an Object to File:

```
#include <iostream>
#include <fstream>
using namespace std;
class Student
{
        char name[20];
        int roll ;
public:
        void getdata( )
        {
                cout<< " Enter name and roll";
                cin>>name>>roll ;
        }
        void display()
        {
                cout<< "Name\t"<< "Roll"<<endl;
                cout<<" "<<name<<"\t"<<" "<<roll;
        }
} ;
int main( )
{
Student X;
X.getdata( ) ;
fstream file;
file.open("data.txt",ios::binary|ios::out);    //opened in out mode so we can perform write operation only.
file.write ((char*) &X, sizeof (X));
file.close( ) ;
return 0 ;
}
```

Example Program for Reading an Object from File:

```
#include <iostream>
#include <fstream>
using namespace std;
class Student
{
        char name[20];
        int roll ;
public:
        void getdata( )
        {
                cout<< " Enter name and roll";
                cin>>name>>roll ;
        }
        void display( )
        {
                cout<<" Name"<<"\t"<<" Roll" <<endl;
                cout<<" "<<name<<"\t"<<" "<<roll;
        }
};
int main( )
{
        Student Y;
        fstream file;
        file.open(" data.txt", ios::binary|ios::in|ios::trunc);    //remember fstream is used.
        file.read((char*)&Y, sizeof (Y));
        Y.display( ) ;
        file.close( ) ;
        return 0 ;
}
```

Simultaneous Reading and Writing:

```
#include <iostream>
#include <fstream>
using namespace std;
class Student
{
   char fname[100];
   char lname[100];
   int roll ;
public:
   void getdata( )
{
   cout<< " Enter first_name, last_name and roll";
   cin>>fname>>lname;
   cin>>roll ;
}
void show( )
{
          cout<<" "<<fname<<"\t"<<" "<<lname<<"\t"<<" "<<roll;
}
} ;
int main( )
{
         Student X, Y;
         X.getdata( ) ;
         fstream file;                    //declaring fstream object for both reading and writing.
         file.open("sample.txt", ios::binary|ios::in|ios::out|ios::trunc);          //multiple modes.
         file.write ((char*) &X, sizeof (X));                    //writing to the file.
         file.seekg(0);                                 //Reset to the start, for reading.
         cout<<" FName "<<"\t"<<" Lname "<<"\t"<<" Roll " <<endl;
         cout<<"---------------------------------------------------"<<endl;

         while (file.read((char*)&Y, sizeof(Y)) )       //reading one object at a time from file.
         {
                  Y.show();                             //displaying on the screen.
                  cout<<endl;
         }
```

```
        file.close();
        return 0;
}
```

Reading and Writing Multiple Objects:

```
#include <iostream>
#include <fstream>
#include <iomanip>
using namespace std;
class Item
{
        int code ; // item code
        char name[100]; //item name
        float cost ; // cost of each item.
public:
        void readdata(void) ;
        void writedata(void) ;
} ;
void Item:: readdata(void)  // read from keyboard.
{
cout<< " Enter code: "; cin>>code ;
cout<< " Enter name: "; cin>>name ;
cout<< " Enter cost: "; cin>>cost ;
}
void Item:: writedata(void) // for display.
{
cout<<setw(10)<<code<<setw(10)<<name
<<setw(10)<<cost
<<endl ;
}
```

```
int main( )
{
Item i;
char ch;
fstream file ;
file.open (" item.dat", ios:: in|ios:: out|ios::app);
do
{
i.readdata( ) ;
file.write((char*) &i, sizeof (i)) ;
cout<< " Enter another Item (y/n)?\n" ;
cin>>ch;
} while(ch=='y');
file.seekg(0) ;                // reset to start.
cout<< " \n Output: \n\n" ;
while(file.read((char*) &i , sizeof (i)))
{
   i.writedata( );
}
file.close( ) ;
return 0 ;
}
```

1. SEARCHING IN THE FILE

//Displaying Items having price less than 90.

```
#include <iostream>
```

```
int main( )
{
Item i;
char ch;
```

```
#include <fstream>
#include <iomanip>
using namespace std;
class Item
{
        int code ; // item code
        char name[100]; //item name
        float cost ; // cost of each item.
public:
void readdata(void) ;
void writedata(void) ;
void search();
} ;
void Item:: readdata(void) // read from keyboard
{
cout<< " Enter code: "; cin>>code ;
cout<< " Enter name: "; cin>>name ;
cout<< " Enter cost: "; cin>>cost ;
}
void Item:: writedata(void) // for display.
{
cout<<setw(10)<<code<<setw(10)<<name
<<setw(10)<<cost
<<endl ;
}
void search()   //searching for items, having price less than 90.
{
        if(cost<90)
        {
                writedata();
        }
}
```

2. Displaying nth item

For this, first find the location of the first byte of nth object in the file.

General formula for obtaining the desired object;

*int location= (n-1)*sizeof(object);*

Then set the file pointer to reach this byte with the help of seekg() or seekp().

Example:

```
using namespace std;
class Item
{
        int code ;        // item code
        char name[10] ;        //item name
        float cost ;      // cost of each item.
public:
        void writedata(void) ;
};
void Item:: writedata(void)     // formatted display on screen.
{
        cout<<setw(10)<<code<<setw(10)<<name <<setw(10)<<cost <<endl ;
}
int main( )
{
        Item i;
        int n;
        fstream file ;
        file.open (" item.dat", ios:: in|ios:: out);
        cout<<" Enter Item number to be displayed: ";
        cin>>n;
        int a=sizeof(i);          //finding the size of individual object.
        int pos=a*(n-1);          //finding position of nth object.
        file.seekg(pos);          //moving file pointer to that position.
        file.read((char*)&i, sizeof(i));  //reading object which is at the position pos.
         i.writedata( );                  //displaying that item.
        file.close( ) ;
        return 0 ;
}
```

COMPLETE MENU PROGRAM:

//Use this concept in your project, but do not copy the code.

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
using namespace std;
class Item
{
int code ; // item code
char name[10] ; //item name
float cost ; // cost of each item.
public:
void readdata(void) ;
void writedata(void) ;
int retCode()
{
   return code;
}
} ;
void Item:: readdata(void)  // read from keyboard.
{
cout<< " Enter code: "; cin>>code ;
cout<< " Enter name: "; cin>>name ;
cout<< " Enter cost: "; cin>>cost ;
}
void Item:: writedata(void) // for display.
{
   cout<<code<<"\t"<<name<<"\t\t"<<cost<<endl ;
}
void Insert()             // function to write in a binary file.
{
         ofstream file;
         file.open("Item.dat ", ios::binary | ios::app);
         Item obj;
         obj.readdata();
         file.write((char*)&obj, sizeof(obj));
```

```
            file.close();
}
void Display()          // function to display records of file on screen, formatted output.
{
            ifstream file;
            file.open("Item.dat ", ios::binary);
            Item obj;
            cout<<"Code\t"<<"Item Name\t"<<"Price"<<endl;
            cout<<"..............................."<<endl;
            while(file.read((char*)&obj, sizeof(obj)))
              {
                  obj.writedata();
              }
            file.close();
            cout<<"..............................."<<endl;
            cout<<"..............................."<<endl;
}

void Delete(int n)              // function to delete a record.
{
    Item obj;
    ifstream inFile;
    inFile.open("Item.dat ", ios::binary);
    ofstream outFile;
    outFile.open("temp.dat", ios::out | ios::binary);
    while(inFile.read((char*)&obj, sizeof(obj)))
    {
        if(obj.retCode()!= n)
        {
            outFile.write((char*)&obj, sizeof(obj));
        }
    }
    inFile.close();
    outFile.close();
    remove("Item.dat "); //deletes a file from memory.
    rename("temp.dat", "Item.dat");
// renames the file temp.dat as Item.dat.
}
```

```
void Modify(int n)                  // function to modify a record.
{
    fstream file;
    file.open("Item.dat",ios::in | ios::out);
    Item obj;
    while(file.read((char*)&obj, sizeof(obj)))
    {
        if(obj.retCode() == n)
        {
            cout << "\nEnter the new details of item";
            obj.readdata();
            int pos = sizeof(obj);
            file.seekp(-pos, ios::cur);      //moving put pointer to the pos position for writing new detail.
            file.write((char*)&obj, sizeof(obj));
        }
    }
    file.close();
}
 void Search(int n)                 //function to search and display from binary file.
{
    ifstream inFile;
    inFile.open("Item.dat ", ios::binary);
    Item obj;
    while(inFile.read((char*)&obj, sizeof(obj)))
    {
        if(obj.retCode() == n)
        {
            obj.writedata();
        }
    }
   inFile.close();
}
```

```
int main( )
{
   int r,choice;
   char ch;
   do
   {
      cout<<"******MENU***********"<<endl;
      cout<<"1.INSERT"<<endl;
      cout<<"2.DISPLAY"<<endl;
      cout<<"3.MODIFY"<<endl;
      cout<<"4.DELETE"<<endl;
      cout<<"5.SEARCH"<<endl;
      cout<<"6.EXIT"<<endl;
      cout<<"Enter your choice: "<<endl;
      cin>>choice;
      switch(choice)
      {
         case 1:
            Insert();
            break;
         case 2:
            Display();
            break;
         case 3:
            int c;
            cout<<"Enter item code to  modify(1-100): "<<endl;
            cin>>c;
            Modify(c);
            break;
         case 4:
            int m;
            cout<<"Enter item code to delete(1-100): "<<endl;
            cin>>m;
            Delete(m);
            break;
        case 5:
         int s;
```

```
            cout<<"Enter item code to search(1-100): "<<endl;
            cin>>s;
            Search(s);
            break;
        case 6:
            exit(0);
        }
    } while(choice<7);
}
```

12.8 Exercise

1. A file named "**Info.dat**" contains a list of items with SN, Items, and Price. Write a program to read the file and output the item list in the following form:

SN	**Items**	**Price**
1	STAX	100
2	DVD	15
3	SUGAR	80
4	DOSA	150
5	APPLE	70

2. Write an interactive, menu driven program that will access the file "**Info.dat**" and implement the following task:

    a. Determine the Price of the specified item.

    b. Update the price, whenever there is a change.

    c. Add the new item.

    d. Find items with price less than 100.

3. Create a file named "**Employee.dat**", which contains a list of Employees with Name, ID, Salary, Phone Number, and Address. Write an interactive, menu driven program to implement the following task:

    a. Determine the phone number of the specified person.

    b. Determine the salary of the specified person.

    c. Determine the name if telephone number is known.

    d. Update/Modify the phone number, whenever there is change.

    e. Add new employee.