UNIT-4

Inheritance and Polymorphism

ILO1: Basics of inheritance

Inheritance:

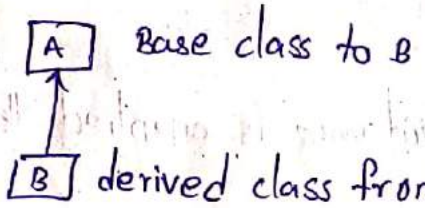
When the properties of one class can be aquired by another class without writing the code again. It is called inheritance.

Types of inheritances:

1. single inheritance
2. Multi level inheritance
3. Multiple inheritance
4. Hirarchical inheritance.
5. Hybrid inheritance.

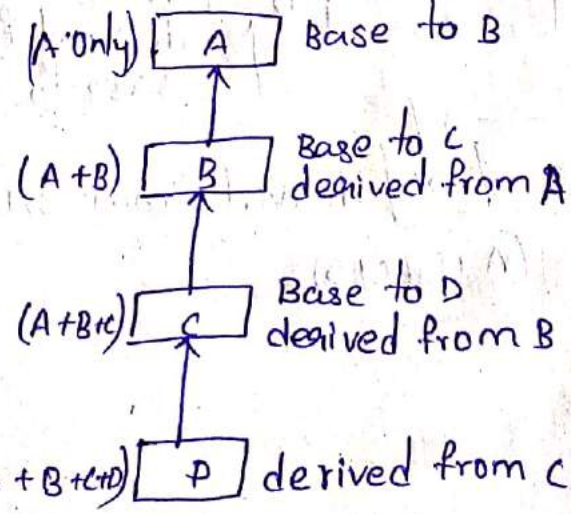
Single inheritance:

When a class is derived from single base class is called single inheritance.



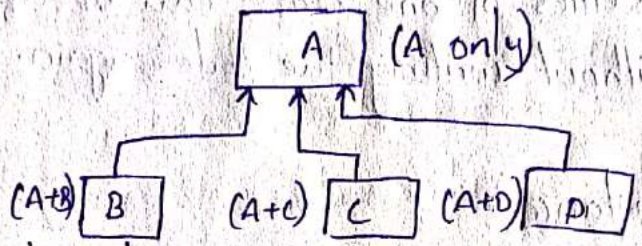
Multilevel inheritance:

When there is more than one level of inheritance then it is called multilevel inheritance.



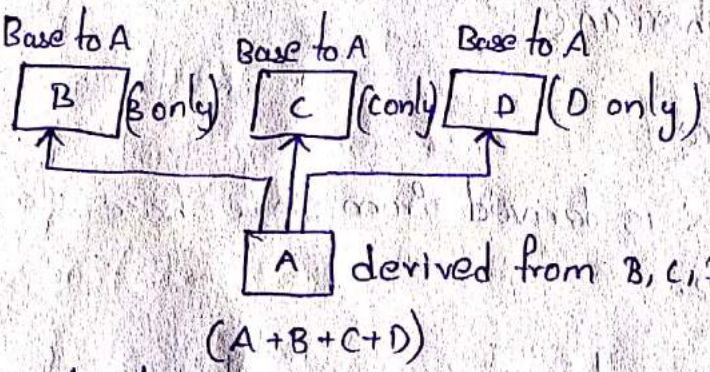
3. Hirarchical inheritance:

In hirarchical inheritance, more than one class can be derived. That means one base class is having more number of derived classes.



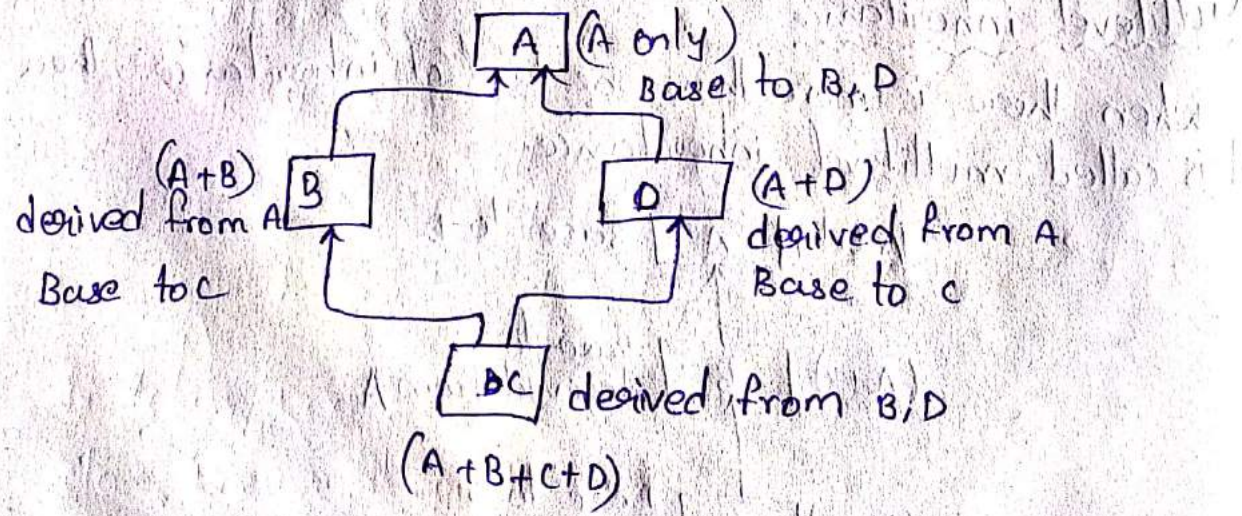
4. Multiple inheritance:

More than one base class to a derived class.



5. Hybrid inheritance:

If more than one inheritance is applied then it is called hybrid inheritance.



* Code reusability is the main feature of inheritance in OOP. This will reduce the code and time for the program so the time complexity and space complexity will be less when we are using inheritance.
* The mechanism of deriving a new class from an old one is called inheritance. The old class is refered as base class and the new class is refered as derived class.

Defining derived classes:

A derived class can be defined by specifying its relationship with the base class in addition to its own details.

The General form

```
class Derived-class : visibility-mode Base-class
{
    ----
    ----  //members of derived class
    ----
};
```

Example:

```
class ABC : private xyz
{
    ----
    ----  //member of ABC
    ----
};
class ABC: public xyz
{
    ----
    ----  //members of ABC
    ----
};
class ABC: xyz
{
    ----  //members of ABC
};
```

single inheritance:

```
#include<iostream.h>
class Box
{  private:
     double len, bre, hei;
   public:
     Box()
     { cout<<"enter len, bre, hei"; vol of ob1 : 1080
       cin>>len>> bre >>hei;        weight = 16
     }                              details of ob2
     Box(double l, double b, double h)  len = 1.1
     { len=l;                       bre = 2.2
       bre=b;                       hei = 3.3
       hei=h;                       vol of ob2 : 7.986
     }                              weight = 4.4
     Box (double x)                 details of ob3 :
     { len=bre=hei=x;               length = 5.5
     }                              bre = 5.5
     Box(Box &ob)                   hei = 5.5
     { len=ob.len;                  Vol of ob3 : 166.375
       bre=ob.bre;                  details of ob4 :
       hei=ob.hei;                    len = 1.1
     }                                bre = 2.2
     double vol()                     heig = 3.3
     { cout<<"length = "<<len<<endl;  vol of ob4 : 7.986
       cout<<"bredth = "<<bre<<endl;  weight = 4.4
       cout<<"height = "<<hei<<endl;
       return len*bre*hei;
     }
};//class Box
class Boxweight : public Box
{    private:
       double wei;
     public:
       {
```

output

enter len, bre, hei:
10, 12, 14
enter weight: 16
details of ob1:
len = 10
bre = 12
hei = 14

```
Boxweight(): Box()   optional for only default constructor
{ cout<<"enter a weight:";
  cin>>wei;
}
Boxweight(double l, double b, double h, double w): Box(l,b,h)
{  wei=w;
}
Boxweight (double x): Box(x)
{  wei = x;
}
Boxweight (Boxweight &ob) : Box(ob)
{  wei = ob.wei;
}
void displayweight()
{  cout<<"weight = "<<wei<<endl;
}
}; //end of Boxweight.
void main()
{ Boxweight ob1, ob2(1.1, 2.2, 3.3, 4.4), ob3(5.5), ob4(ob2);
 cout<<"\n ob1 details\n";
  double v = ob1.vol();
  cout<< "volume of ob1 = "<<v<<endl;
  ob1.displayweight();
  cout<<" Details of ob2 = "<<endl;
    v = ob2.vol();
    cout<<" vol of ob2 = "<<v<<endl;
    ob2.displayweight();
   Cout<<"\n ob3 details \n";
    v = ob3.vol();
    Cout<<" vol of ob3 = "<<v<<endl;
    ob3.displayweight();
    cout<<"\n ob4 details\n";
    v = ob4.vol();
```

```
    cout<<"In val of ob4 = " <<v<<endl;
    ob4.displayweight();
}
```

MULtilevel inheritance

```
#include<iostream.h>
class stu
{ private:
    int rno;
    char name[20];
  public:
    stu()
    { cout<<"enter rno:";
      cin>>rno;
      cout<<"enter name:";
      cin>>name;
    }
    void display()
    { cout<<"Rollno= "<<rno<<endl;
      cout<<"Name = "<<name<<endl;
    }
}; // end of class stu
class marks : public stu
{ protected:
    int m[6];
  public:
    marks() : stu()
    { cout<<"enter 6 sub marks:";
      for(int i=0;i<6;i++)
        cin>>m[i];
    }
    void dispmarks()
    { cout<<"In display 6 sub marks\n";
```

output

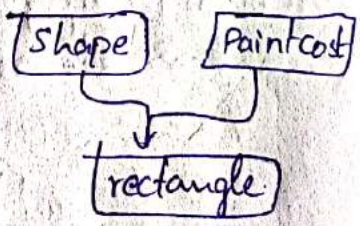
```
enter rno : 1
enter name : rani
enter 6 subj marks
67
78
79
81
83
90
rno=1
name=rani
display 6 sub marks
67 78 79 81 83 90
total marks:478
percent:79.66667
```

```
    for(i=0;i<6;i++)
        cout<<" "<<m[i];
    }
}; // end of marks
class result : public marks
{   private:
        int tot;
        float per;
    public :
        void calc( )
        { tot =0;
          for(i=0 ;i<6; ;++)
              tot += m[i];
          per = (float) tot/6.0;
        }
        void dispresult( )
        { cout<<"Total marks : "<<tot<<endl;
          cout<<"percent = "<<per<<endl;
        }
}; // end of result
void main( )
{ result r;
  r.calc( );
  r.dispname( );
  r.dispmarks( );
  r.dispresult( );
}
```

Multiple Inheritance:

①
```
#include<iostream.h>
{  protected:
       int width, height;
   public:
       void setwidth(int w)
       {  width = w;
       }
       void setheight(int h)
       {  height = h;
       }
};
class paintcost
{  public:
       int getcost(int area)
       {  return area*70;
       }
};
class rectangle : public shape, public paintcost
{     public:
          int getAreac)
          {  return(width * height);
          }
};
int mainc )
{  rectangle r;
   int a;
   r.setwidth(5);
   r.setheight(7);
   a = r.getareac );
   cout<< "total area = "<<a<<endl;
   cout << "cost of paint = "<<r.getcost(a) << endl;
}
```

total area = 35
cost of Paint = 2450



```
#include<iostream.h>
class stud
{  protected:
       int rno, m1, m2;
   public:
       void get()
       {  cout<<"enter rno:";
          cin>>rno;
          cout<<"enter two highest marks:";
          cin>>m1>>m2;
       }
};
class extracur
{  protected:
       int xm;
   public:
       void getsm()
       {  cout<<"enter extracur marks:";
          cin>>xm;
       }
};
class output : public stud, public extracur
{  int tot, avg;
   public:
       void display()
       {  tot=(m1+m2+xm);
          avg=tot/3;
          cout<<"rno="<<rno<<endl;
          cout<<"total="<<tot<<endl;
          cout<<"avg="<<avg<<endl;
       }
};
int main()
{  output o;
   o.get();
   o.getsm();
   o.display();
}
```

③
```
#include<iostream.h>
class stu
{   Protected:
      int rno;
      Char name[20];
    public:
      stu()
      { cout<<"enter rno: ";
        cin>>rno;
        cout<<"enter name:";
        cin>>name;
      }
      void dispname()
      { cout<<"roll no:"<<rno<<endl;
        cout<<"name:"<<name<<endl;
      }
};
class marks
{   Protected:
      int m[6];
    public:
      marks() : stu()
      { cout<<"enter 6 subject marks:";
        for(int i=0;i<6;i++)
          cin>>m[i];
      }
};
class result : public stu, public marks
{   private:
      int tot;
      float per;
    public:
      void calc()
      {  tot=0;
         for(int i=0;i<6;i++)
           tot+=m[i];
      } per=(float)tot/6.0;
```

```
void dispresult()
 { cout<<"total marks: "<<tot<<endl;
    cout<<" percent = "<<per<<endl;
 }
};
void main()
{ result r;
  r.calc();
  r.dispname();
  r.dispmarks();
  r.dispresult();
}
```

Hirarchical inheritance:

```
#include<iostream.h>
class member
{ char gender[10];
  int age;
  public:
    void get()
     { cout<<"enter age:";
       cin>>age;
       cout<<"enter gender:";
       cin>>gender;
     }
     void disp()
     { cout<<"Age: "<<age<<endl;
       cout<<"Gender:"<<gender<<endl;
     }
};
class std: public member
{  char level[20];
   public:
     getdata()
     { //member::get();
```

```
        get();
        cout << "class : ";
        cin >> level;
    }
    void disp2()
    { member :: disp();
      cout << "level :" << level << endl;
    }
};
class staff : public member
 { float salary;
   public:
     void getdata()
     { member :: get();
       cout << "enter salary :";
       cin >> salary;
     }
     void disp3()
     { member :: disp();
       cout << "salary is :" << salary << endl;
     }
};
int main()
{   stud st;
    staff s;
    cout << "enter student details \n";
    st.getdata();
    cout << "\n";
    cout << " display student details \n";
    st.disp2();
    cout << "enter staff details \n";
    s.getdata();
    cout << endl;
    s.disp3();
    return 0;
}
```

output:
Enter student details
enter age : 16
enter gender : female
enter level : 12
enter staff details
age : 45
gender : male
enter salary : 30

Hybrid inheritance:

```
#include<iostream.h>
#include<conio.h>
class sports;
class std
{ int rno;
  char name[20];
public:
  void getc()
  { cout<<"enter rno:";
    cin>>rno;
    cout<<"enter name:";
    cin>>name;
  }
  void disp()
  { cout<<"rno:"<<rno<<endl;
    cout<<"name:"<<name<<endl;
  }
};
class marks: public std
{ protected:
    int s[6];
public:
  void get marks()
  { getc();
    cout<<"enter marks:";
    for(int i=0;i<6;i++)
      cin>>s[i];
  }
  void dispmarks()
  { disp();
    cout<<" marks:"<<endl;
    for(int i=0;i<6;i++)
      Cout<<s[i]<<"\t";
  }
};
```

output:

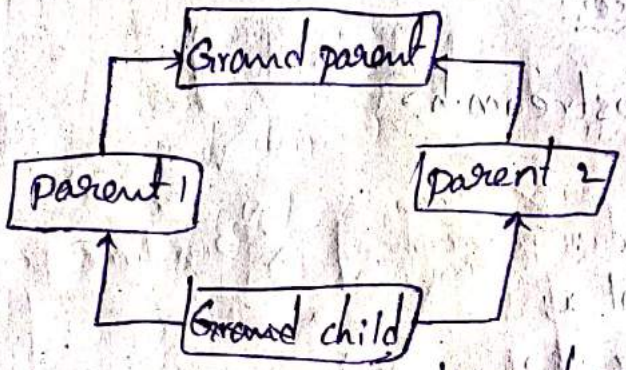
```
enter rno:1
enter name:ravi
enter marks:
67 78 79 80 80 80
enter sports marks:
67 78 79 80 80 80
rno:1
name:ravi
marks:
67 78 79 80 80 80
Sports marks:
total marks:407
percent:58.142857
```

```
class sports
{ protected:
    int sm;
  public:
    void getsports()
    { cout<<"enter sports marks: ";
      cin>>sm;
    }
    void dispsm()
    { cout<<"sports marks:"<<sm<<endl;
    }
};
class result: public marks, public sports
{ protected:
    int tot;
    float per;
  public:
    void total()
    { tot=sm;
      for(int i=0; i<6; i++)
        tot+=s[i];
      per=(float)tot/7.0;
    }
    void display()
    { dispmarks();
      dispsm();
      cout<<"total marks:"<<tot<<endl;
      cout<<"percent:"<<per<<endl;
    }
};
void main()
{ result r;
  clrscr();
  r.getmarks();
```

```
r.getsports();
r.total();
r.display();
}
```

Virtual Base class:

When we are using Multilevel, multiple and Hyrarchical inheritance are Combindly used we may get problem called ambiguity error for the compiler because two copies of same members are inherited to a class in two directions. The compiler can not decide which copy of the data inherited to be considered.



The child class is having two base classes called parent1 and parent2 which themselves are having a common base class called grand parent. The child inherits the same data members and member functions from grand parent in two paths. The grand parent is also called as indirect base class.

By using this multipath inheritance the compiler get ambiguity error because it is not able to decide which path is to be considered.

```
class Grandparent
{
  ---
  ---
};
class parent1 : public virtual Grandparent
{
  ---
  ---
};
```

```
class parent2 : virtual public Grandparent
{
    ----
    ----
};

class child : public parent1, public parent2
{
    ----
    ----
};
```

This is also called as multipath inheritance.

Qualified classes:

```
#include<iostream.h>
class A
{ public:
    int x;
    A(){};
  class B
  { public:
      int y;
      B(){}
  };
};
class C: public A, A::B
{ public:
    int z;
    void show()
    { cout<<"x="<<x<<"y="<<y<<"z="<<z;
    }
    c(int j, int k, int l)
    { x=j;
      y=k;
```

```
        z=i;
    }
};
int main()
{   C   c(4,7,1)
    c.show();
    return 0;
}
```

In the above program the class B defined inside the class A. class A is qualifier class for class B. The class C is the inherited class from classes A and B.

In the statement class c we are inherited class A publicly and also B privately.

To access the class B it is preceded by the qualifier class A and the scope resolution operator if we mention only class A the class B will not be considered for inheritance. similarly, if we mention only class B the qualified class A will not be considered for inheritance.

Constructor in derived class in C++:

Base class constructors are always called in the derived class constructors whenever we create derived class objects first the base class default constructor is executed and then derived class constructor finishes execution.

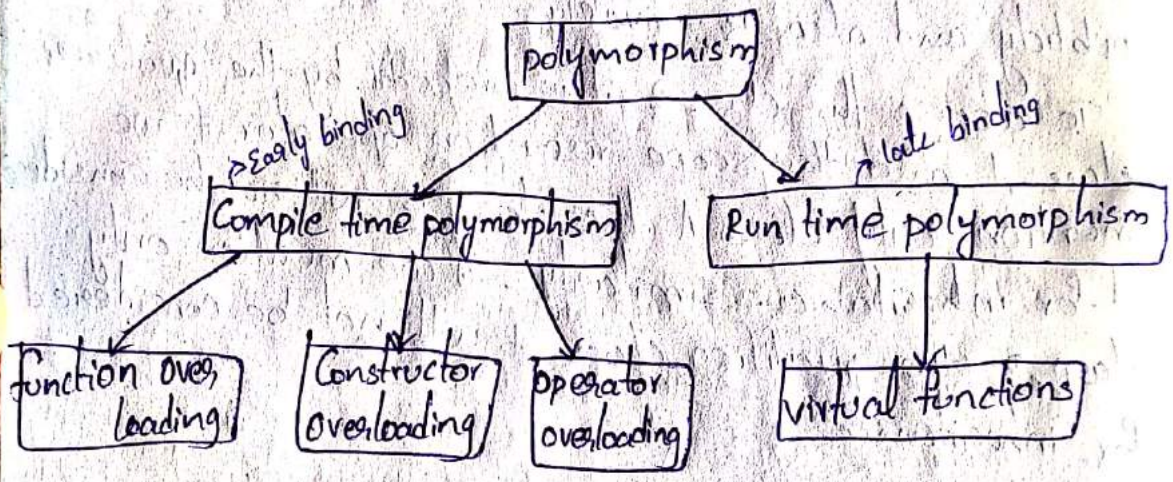
Program:

```
#include<iostream.h>
class parent
{   public:
        parent()
        {  Cout<<" Inside base class\n";
};
```

```
class child : public parent
{ public :
    child()
    { cout << "Inside derived class \n";
    }
};
int main()
{ child c;
  return 0;
}
```

output
Inside base class
Inside derived class.

Virtual functions:



polymorphism is one of the important feature of oop we already know about compile time polymorphism, by using function overloading, constructor overloading and operator overloading. Now we concern about run time polymorphism which also called as virtual functions.

Compile time polymorphism is also called as early binding. Run time polymorphism can be termed as late binding. That means early binding occurs at the compile time and late binding occurs during run time.

For runtime polymorphism we are going to use pointers.

Example

```
#include<iostream.h>
class A
{   private :
        int x;
    public:A(){x=10;}
        void show()
        {   cout<<"x = "<<x<<endl;
        }
};
class B: public A
{   private :
        int y;
    public:B(){y=20;}
        void show();
        {   A::show();
            cout<<"y = "<<y<<endl;
        }
};
int main()
{   B b;
    b.show();
    return 0;
}
```

In the above program when we are using the same function with same signatures in both the base and derived classes then only one function is going to be executed that is only the derived class function will be called. We are having one solution with this problem is by using scope resolution operator with the name of base class.

2.

```
#include<iostream.h>
class Base
{ public:
    void display()
    { cout<<"\n display base\n";
    }
    virtual void show()
    { cout<<"\n show base\n";
    }
};
class derived : public base
{ public:
    void display()
    { cout<<"\n display derived\n";
    }
    void show()
    { cout<<"\n show derived\n";
    }
};
int main()
{ Base B;
  Derived D;
  Base * bptr;
  cout<<"\n bptr points to Base\n";
  bptr = &B;
  bptr -> display();
  bptr -> show();
  bptr = &D;
  bptr -> display();
  bptr -> show();
}
```

bptr points to base
display base
show base
display base
show derived

void virtual display()

bptr points to base
display base
show base
display derived
show derived

Pure virtual functions:

It is a normal practice to declare a function virtual inside the base class and redefining in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a place holder.

Ex: virtual void display() = 0;
// do-nothing function.

The above function is called pure virtual function. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases the compiler requires each derived class to either define the function or redeclare it as a pure virtual function.

pure virtual functions can not be instantiated that means we can not create any object for a class having pure virtual function. A class having pure virtual function can also be called as abstract class.

Program

```
#include<iostream.h>
class Base
{   protected:
        int a, b;
    public:
        Base()
        { cout<< "enter a,b:";
          cin>>a>>b;
        }
        void display()
        { cout<<"a="<<a<<endl;
          cout<<"b="<<b<<endl;
        }
        virtual int sum() = 0; // pure virtual functi
};
```

```
class derived : public Base
{   private:
        int c;
    public:
        Derived()
        {   cout<<"enter c:";
            cin>>c;
        }
        void dispc()
        {   cout<<"c="<<c<<endl;
        }
        int sum()
        {   return(a+b+c);
        }
};
int main()
{   derived d;
    d.display();
    d.dispc();
    cout<<"\n sum="<<d.sum();
}
```

enter a,b
20 10
enter c
30
a=10
b=20
c=30
sum=60

Virtual destructor:

Deleting a derived class object using a pointer to a base class that has a non-virtual destructor results in undefined behaviour. To correct this situation the base class should be defined in the virtual destructor.

Example

```
#include<iostream.h>
class Base
{   public:
        Base()
        { cout<<"In constructing Base In";
        }
        virtual ~Base()
        { cout<<"In Destructing Base In";
        }
};
class Derived : public Base
{   public:
        Derived()
        { cout<<"In constructing derived In";
        }
        ~Derived()
        { cout<<"In Destructing derived In";
        }
};
int main()
{   Derived *d = new Derived();
    base *b = d;
    getchar();
    return 0;
}
```

UNIT-5

Exception Handling

Introduction:

In any programming language we are having two types of errors.

1. syntax errors.
2. logical errors.

The compiler identifies the syntax errors. The main reason for syntax error is poor understanding about the programming construct. The compiler takes care about syntax errors. But logical errors are not going to be identified by any device. Some examples of logical error are

division by zero error (or) array index out of bound error and overflow error etc.

In ANSI C programming there is no mechanism for finding these kind of logical errors. In ANSI C++ the identification of logical errors are incorporated.

Basics of Exception handling:

Exceptions are of two kinds namely sy.

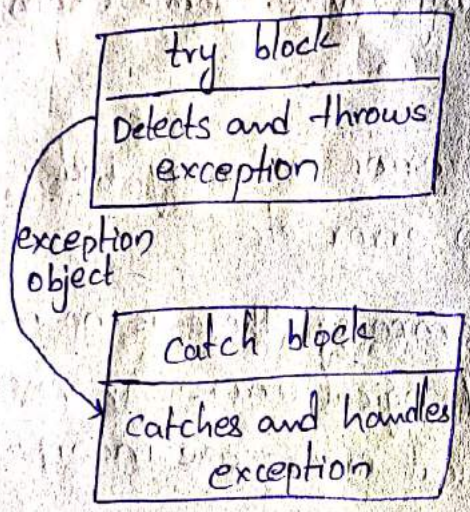
1. Synchronous Exceptions and
2. Asynchronous Exceptions

Errors such as out of range index and overflow belong to the synchronous type exception. The errors that are caused by events beyond the control of program such as keyboard interupts are called asynchronous exception. We concern about synchronous exception.

The purpose of exception handling mechanism is to provide the means of exception and to detect report an exceptional circumstance as to take appropriate action. The exception handling mechanism is described as follows :

1. Find the problem (Hit the exception)
2. Inform that an error has occured ('throw' the exception)
3. Recieve the error information (catch the exception)
4. Take corrective actions (handle the exception)

Exception Handling Mechanism:

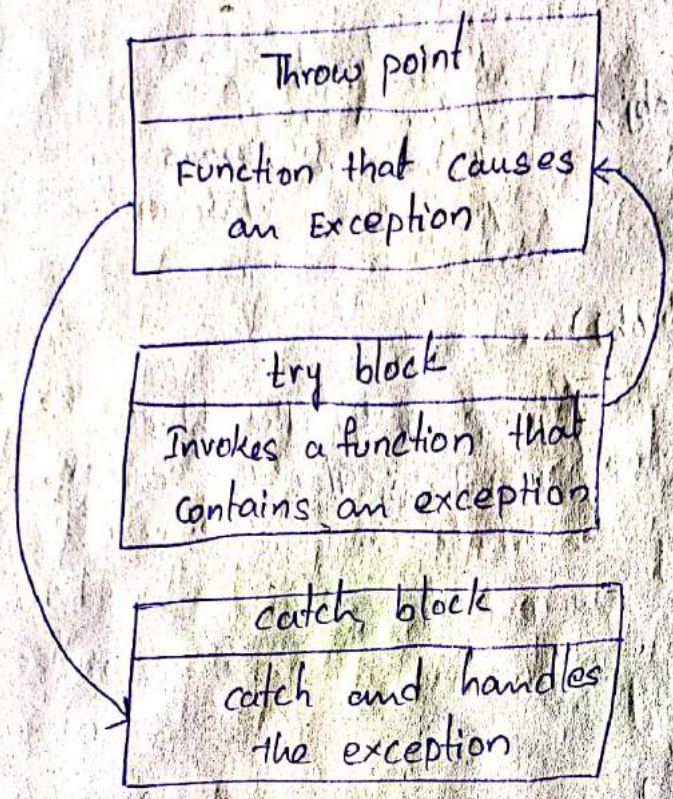


C++ exception handling mechanism is basically built upon three keywords namely, try, throw and 'catch'. The keyword try is used to preface a block of statements which may generate exceptions. This block of statement is called try block. When an exception is detected it is thrown using a throw statement in the try block. A catch block defined by the keyword catch catches the exception thrown by the throw statement in the try block and handles it appropriately.

```
try
{
    - - - -
    - - - -
    - - - -
    throw obj;      // monitor the block
    - - - -         // for errors if
    - - - -         // any error throws
    - - - -         // by throw statement
}
catch (type obj)
{
    - - - -         // catches by the block
    - - - -         // and handles the error
    - - - -
}
```

Program:

```
int main()
{ int a,b;
  cout<<"enter a,b:";
  cin>>a>>b;
  try
  {
      if(b!=0)
      { cout<<(float) a/b;
      }
      else
          throw(b);
  }
  catch(int x)
  {
      cout<<"exception caught:"<<x;
  }
}
```

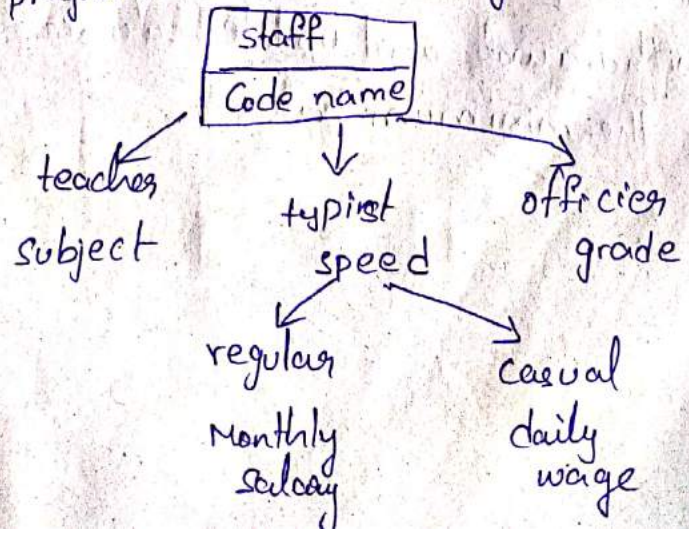



```
type function(arg list) // function with exception
{
    - - - - -
    - - - - - // throws exception
    - - - - -
}
- - -
- - -
try
{
    - - - - -
    - - - - //invoke function here
}
catch(type arg)
{
    - - - -
    - - - - // Handles exception here
    - - - -
}
```

Example

```
void divide(int x, int y, int z)
{ cout<<"\n we are inside the function\n";
  if((x-y)!=0)
  { int R=z/(x-y);
    cout<<"Result = "<<R<<endl;
  }
  else
    throw(x-y);
}
int main()
{ try
  { cout<<"\n we are inside try\n";
    divide(10,20,30);
    divide(10,10,20);
  }
  catch(int i)
  { cout<<" caught the exception\n";
  }
}
```

* what does inheritance mean in C++
* when do we make a class virtual
* Explain about abstract classes.
* Explain the problem of multipath inheritance
* write a program for the following inheritance.



An educational institution wishes to maintain a data base of its employees. A data base is divided into a number of classes whose hirarchical relationships are shown in the above figure. The figure also shows the minimum information required for each class. Specify all the classes and define functions to create the database and retrive individual information as and when required.

* Assume that a bank maintaince two kinds of account for customers one called as savings account and the other as current account. The savings account provides compound intrest and with drawl facilities but no cheque book facility. The current account provides cheque book facility but no interest. Current account holders should also maintain a minimum balance and if the balance falls below this level a service charge is imposed.

Create a class account that stores customer name account number and type of account. From this derive the classes current account and saving's account to make them more specific to their requirement include necessary member functions inorder to achieve the following tasks

a) Accept deposit from a customer and update the balance
b) Display the balance
c) Compute and deposite intrest
d) permit withdrawal and update balance
e) Check for the minimum balance

Throwing Mechanism:

When an exception that is desired to be handled is detected its is thrown using the throw statement in one of the following forms.

```
throw(exception);
throw exception;
throw; //used for rethrowing an exception
```

Exception may be any type (or constant. It is also possible to throw objects not intended for error handling.

When an exception is thrown it will be caught by the catch statement ossociated with the try block. That is the control exits the current try block and is transfered to the catch block after the try block.

Catching Mechanism:

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

The catch statement looks like a function block with type mentioned within the paranthesis the type indicates the type of exception that catch block handles. 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.

Multiple catch statements:

```
try
{
}
catch(type1 arg)
{
}
```

```
catch (type2 arg)
{
  - - -
}
catch (type3 arg)
{
  - - -
}
catch (typeN arg)
{
  - - -
}
```

we can write multiple catch statements for a single try whenever the try block is executed we may have more than one exception may arraised we don't know which type of exception is going to occur in runtime. we are having facility to handle more than one exception for a single try. we can define catch statements with different type of variables and also the arguments. When an exception is thrown from the try block the exception handlers are searched inorder for an appropriate match. The first handler that yields a match is execution. After executing the handler the control goes to the first statement after the last catch block for the try.

```
#include<stdio.h>
using namespace std;
void test(int x)
{   try
    {  if(x==1)
          throw x; //int
       else if(x==0)
          throw 'x'; //char
       else if(x==-1)
          throw 1.0;
       cout<<"end of try block\n";
    }
    catch(char c)
    {  cout<<"caught a character\n";
    }
    catch(int m)
    {  cout<<"caught an integer\n";
    }
    catch(double d)
    {  cout<<"caught a double\n";
    }
    cout<<"end of try catch system";
} //end of function test
int main()
{  cout<<"\n testing multiple catches\n";
   cout<<" x==1\n";
   test(1);
   cout<<" x==0\n";
   test(0);
   cout<<" x==-1\n";
   test(-1);
   cout<<" x==2\n";
   test(2);
   return 0;
}
```

output:

Testing multiple catches
x==1
Caught an integer

```
End of try block
end of try catch system
x == 0
Caught a character
end of the try block
end of try catch system
x == -1
Caught a double
end of try block
end of try catch system
x == 2
end of try block
end of try catch system
```

Description:

The above program invokes the function test with x = 1 and therefore throws 'x' as an int exception this matches the type of parameter m in catch 2 and therefore catch handler is executed immediately after the execution the function test is again invoked with x = 0 this time the function throws x a character type exception and therefore the catch 1 is executed. and for sending '1' as a parameter then '1.0' will be throwed and catch 3 will be executed and when we are sending '2' as a parameter there is no matching in the try block so nothing is thrown with the parameter of '2' so no appropriate catch is there for the parameter 2.

Rethrowing an exception:

A handler may decide to rethrow the exception got without processing it in such situations we may simply invoke throw without any arguments.

syntax throw;

This causes the current exception to be thrown to the next enclosing try catch sequence and is got by a catch statement listed after that enclosing try block.

```
#include<stdio.h>
using namespace std;
void divide(double x, double y)
{ cout<<"Inside function\n";
  try
  { if(y==0.0)    //Throwing double
      throw y;
    else
      cout<<"Division = "<<x/y<<endl;
  }
  catch(double)
  { cout<<"caught double inside function";
    throw; //rethrowing
  }
  cout<<"\n End of function\n";
}
int main()
{ cout<<"\n inside main\n";
  try
  { divide(10.5,2.0)
    divide(20.0,0.0)
  }
  catch(double)
  { cout<<"caught double inside main\n";
  }
  cout<<"End of main\n";
  return 0;
}
```

Output:

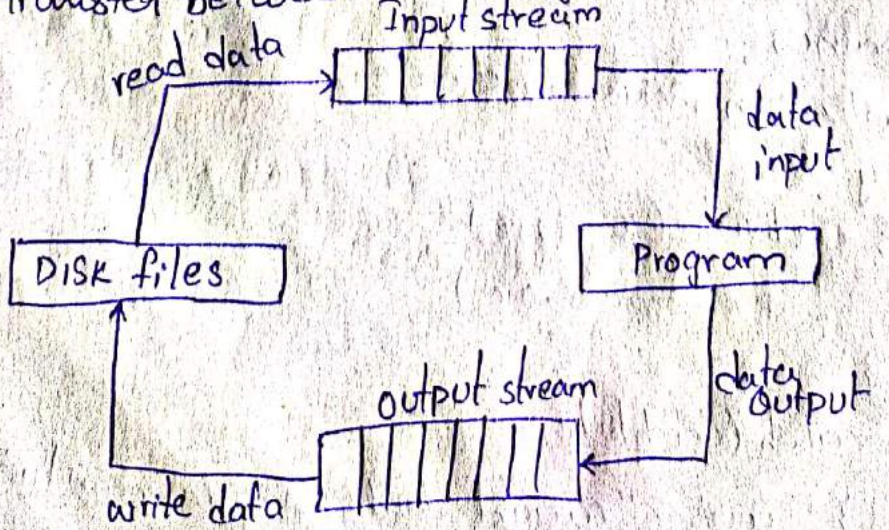
```
Inside main
Inside function
Division = 5.25
End of Function
Inside function
Caught double inside function
Caught double, inside main
End of main
```

Working with files

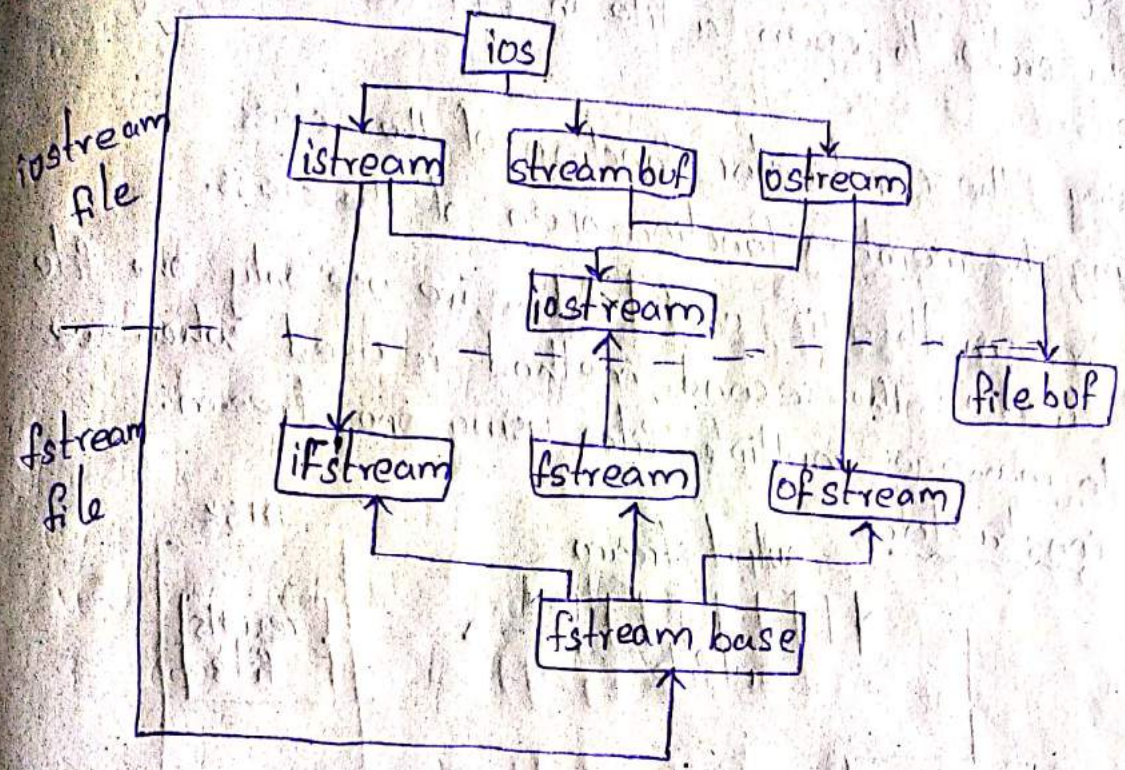
Many real time problems handle large volumes of data and in such situations we need to use some devices such as floppy disk, hard disk or pendrive. The data is stored in these auxilary storage devices using the concept called of files. A file is a collection of related data stored in a particular area on the disk. Programs can be designed to perform the read and write operations on this files.

Data communication involves two types.

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



classes for file stream operation



The io system

The I/o system of c++ handles file operations. which are very much similar to the input and output operations. It uses file stream as an interface between the program 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 reads data from the file and the output stream inserts data to the file.

classes for file stream operation:

The I/o system of c++ contains a set of classes that define the file handling methods. These include if stream, ofstream and fstream. These classes are derived from fstream base and from the corresponding iostream class as shown in the above diagram.

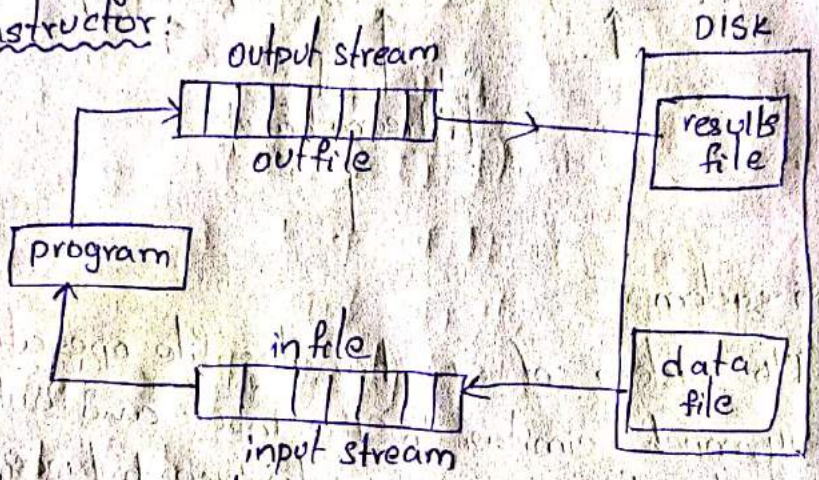
Opening and closing a file:

In order to open a file in C++ we are having two methods.

1. Using the constructor function of the class
2. Using the member function open of the class.

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

Using constructor:




```
ofstream outfile("results"); // output only
ifstream infile("Data"); // input only
```

creating file with constructor function:

```
#include<fstream.h>
int main()
{ ofstream outf("ITEM");
  cout<<"enter item name";
  char name[30];
  cin>>name;
  outf<<name<<endl; // write to file item
  cout<<"enter item cost";
  float cost;
  cin>>cost;
  outf<<cost<<endl; // write cost to item
  outf.close(); // Disconnect item from outf
  ifstream inf("ITEM"); // connect inf to file item
  inf>>name;
```

```
    inf>>cost;
    cout<<endl;
    cout<<" ITEM Name:"<<name<<endl;
    cout<<" ITEM cost:"<<cost<<endl;
    inf.close();
    return 0;
}
```

output:

```
enter item name Hard disk
enter item cost 2500
ITEM Name : Harddisk
ITEM cost: 2500
```

Creating files with Multiple files (using open() function):

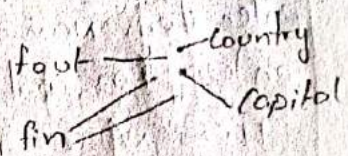
```
#include<iostream.h>
#include<fstream.h>
int main()
{
    ofstream fout; // create dp stream
    fout.open("country");
    fout<<"united states of America\n";
    fout<<"united kingdom\n";
    fout<<"south korea\n";
    fout<<"India\n";
    fout.close();
    fout.open("capital");
    fout<<"washington\n";
    fout<<"london\n";
    fout<<"seoul\n";
    fout<<"India New Delhi\n";
    fout.close();
    // Reading files
    ifstream fin;
    fin.open("country");
    cout<<"\n contents of country file\n";
    char line[80];
```

output

```
contents of the country file
united states of America
United kingdom
South korea
India
contents of capital
washington
london
seoul
new delhi
```

```
while(fin)                    // inbuilt function
{  fin.getline(line, 80);
   cout << line << endl;
}
fin.close();
fin.open("capital");
cout << "In contents of capital \n";
while(fin)
{  fin.getline(line, 80);
   cout << line << endl;
}
fin.close();
return 0;
}
```

to open a file separate use
c:\dir country
c:\typo country




```
// Reading 2 files simultaneously
#include<iostream.h>
#include<fstream.h>
int main()
{  char line[80];
   ifstream fin1, fin2;
   fin1.open("Country");
   fin2.open("Capital");
   for(int i=1;i<10;i++) (or) while(fin1, fin2)
   {  if(fin1.eof()!=0)
      {  cout << "exit from Country \n";
         exit(1);
      }
      fin1.getline(line, 80)
      cout << "capital of " << line << " is ";
      if(fin2.eof()!=0)
      {  cout << "exit from capital \n";
         exit(1);
      }
```

```
    fin2.getline(line,80);
    cout<<line<<endl;
  }
  return 0;
}
```

output:

```
capital of USA is washington
Capital of United kingdom is london
capital of South korea is seou)
capital of India is New Delhi
capital of is exit from country
```

*create file student having rno, name, address, pincode, 6 subject marks. calculate total and percentage put them in a file called studentf.dat. Display the contents of studentf

Detecting end of file using eof()

Detecting of the end of file condition is necessary for preventing any further attempt to read data from the file. In the above program a loop, while(fin) statement returns a value 'zero' if any error occurs in the file operation including the end of file condition. Thus the while loop terminates while fin returns the value of '0' on reaching the end of file condition.

There is another way to detect end of file condition i.e, in the above program we have given the statement

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

Unit-6

Templates examples:

Function Template:

```
//bubble sort
template<class/typename T>
void bubblesort(T a[],int n)
{
    T temp;
    for(int i=0; i<n; i++)
    {  for(j=n-1; i<j; j--)
       {  if(a[j] < a[j-1])
          {  temp=a[j];
             a[j]=a[j-1];
             a[j-1]=temp;
          }
       }
    }
}
int main()
{ int a[5]={10,50,30,40,20};
  bubblesort(a,5);
  char b[5]={'X','U','F','A','M'};
  bubblesort(b,5);
  double c[5]={7.8,4.5,12.9,3.4,5.6};
  bubblesort(c,5);
  Cout<<"In sorted int array:";
  for(int i=0;i<5;i++)
    Cout<<a[i]<<" ";
    Cout<<endl;
  Cout<<"In sorted char array:";
  for(int i=0;i<5;i++)
      Cout<<b[i]<<" ";
    Cout<<endl;
```

output

10 20 30 40 50

a f m u x

3.4 4.5 5.6 7.8 12.9

```
cout<<"\n sorted double array: ";
for(int i=0;i<5;i++)
  cout<<c[i]<<" ";
cout<<endl;
return 0;
}
```

class Template:

```
template<typename T>
// (or)
//template<class T>
class Array
{ private:
    T *ptr;
    int size;
  public:
    Array(T arr[], int s);
    void print();
};
template<typename T>
Array<T>::Array(T arr[], int s)
{ ptr=new T[s];
  size=s;
  for(int i=0; i<size;i++)
    ptr[i]=arr[i];
}//end of constructor array
template<typename T>
void Array<T>::print()
{ for(int i=0;i<size;i++)
    cout<<" "<<*(ptr+i);
  cout<<endl;
}
int main()
{ int arr[5]={1,2,3,4,5};
```

```
Array < int >a(arr,5);
a.print();
char arr1[6] = { 'A', 'B', 'C', 'D', 'E', 'F' };
Array<char>b(arr1, 6);
b.print();
double arr2[4] = { 1.2, 2.3, 3.3, 4.5 };
Array <double> c(arr2, 4);
c.print();
return 0;
}
```

output

```
1  2  3  4  5
A  B  C  D  E  F
1.2  2.3  3.3  4.5
```

File mode par.

Unit 5

File mode parameters: stream_object.open("Filename", mode);

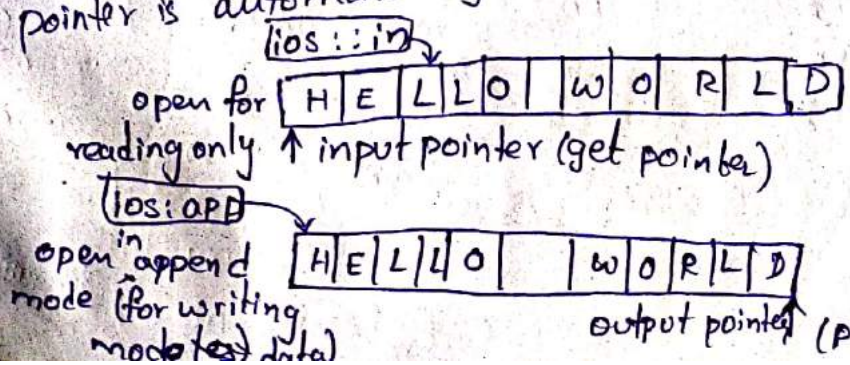
we already used ifstream and ofstream for reading data from the file and writing data to the file respectively. so far we have not mentioned file mode parameters the default mode for write mode is ios::app. The default mode for ios object is ios::in

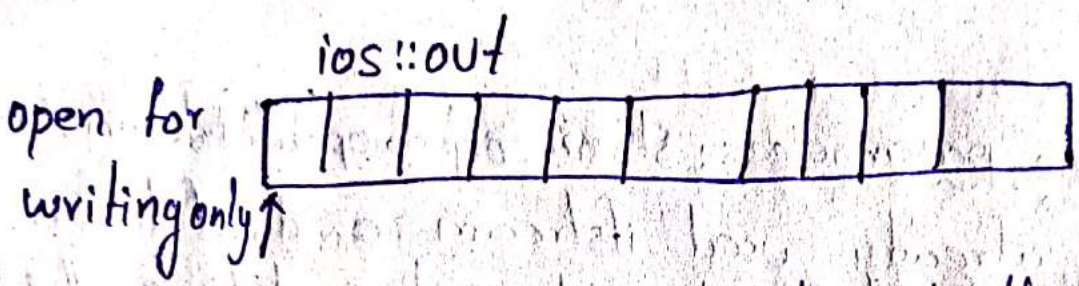
parameter	Meaning
ios::app	append to end of file
ios::ate	go to end of file on opening
ios::binary	open binary file
ios::in	Open file for reading only
ios::nocreate	open fails if the file does not exist.
ios::noreplace	open fails if the file already exist.
ios::out	open file for writing only
ios::trunc	delete the contents of the file if it exist.

I/o manipulators:

Every file has 2 associated pointers known as the file pointers one of them is called the input pointer (get pointer) and the other is called the output pointer (put pointer).

we can use these pointers to move to 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 write into a given file location. Each type an input or output operation takes place the appropriate pointer is automatically advanced.





In the above diagram according to the mode of operation the file pointer is set by the compiler automatically. There are some manipulators to control the moment of pointers by using some manipulator functions. Some below:

i) seekg() - moves get pointer to specified location
ii) seekp() - moves put pointer to specified location
iii) tellg() - give the current position of the get pointer
iv) tellp() - give the current position of put pointer

Formatted console I/O operation:

1. width()

To specify the required field size for displaying an o/p value.

2. precision()

To specify no. of digits to be displayed after the decimal value of float value.

3. fill()

To specify a character that is used to fill the unusual portion of a field.

4. setf()

To specify a format flags that can control the form of o/p display (left-justification or right-just)

5. unsetf()

To clear the flags specified.

Manipulators	Equivalent io's function
setw()	width()
setprecision()	precision()
setfill()	fill()
setiosflags()	setf()
resetiosflags()	unsetf()

width()

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

Ex:1

```
cout.width(5);
cout<<543<<12<<endl;
```

		5	4	3	1	2

Ex:2

```
Cout width(5)
Cout<<543;
Cout width(5)
Cout<<12<<endl
```

		5	4	3			1	2

Ex:

```
#include<iostream.h>
int main()
{ int items[4] = [10,8,12,15];
  int cost[4] = [75,100,60,99];
  cout.width(5);
  cout<<"ITEMS";
  cout.width(8);
  cout<<"COST";
  cout.width(15);
  cout<<"Total value"<<endl;
  int sum=0;
  for(int i=0; i<4; i++)
  {  cout.width(5);
     cout<<items[i];
     cout.width(8);
     cout<<cost[i];
     int value=items[i]*cost[i];
     cout.width(15);
     cout<<value<<endl;
     sum=sum+value;
  }
  cout<<"\n Grand total=";
  cout.width(2);
  cout<<sum<<endl;
}
```

O/P

ITEMS	COST	Totalvalue
10	75	750
8	100	800
12	60	720
15	99	1485

Grand total = 3755

Precision()

```
cout.precision(d);
```

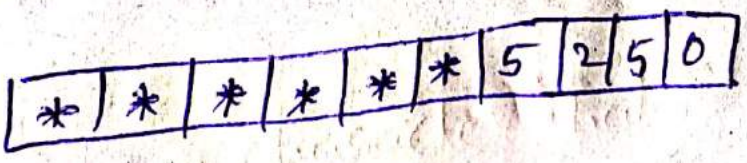
fill()

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

Ex:

```
cout.fill('*');
cout.width(10);
cout<<5250<<endl;
```

o/p:



4th unit

Standard Template library (STL):

Iterator

```
vector<int>::iterator x;
    for list
    list<int>::iterator y;
```

String

```
string s0;
string s1("Hello");
string s2(s1);
string s3(s1, 1, 2); // output → s3 = "el";
string s4(s1, 0, 3); //output → s4 = "Hell";
```

vector class

```
vector<int> a;
vector<int> b(5,10);
vector<int> c(b.begin(),b.end());
```

The meaning of vector is dynamic arrays. These are also known as sequence containers.

sequential containers:

* string
* iterator
* vector
* list.

vector

```
void traverse(vector<int>v)
{   vector<int>::iterator it;
    for(it=v.begin();it!=v.end();it++)
      cout<<*it<<' ';
      cout<<endl;
      for(int i=0;i<v.sizec);i++)
        cout<<v[i]<<' ';
        cout<<endl;
}
```

List:

```
list<int>l1;
list<char>l2;
list<double>l3;
list<int>L(5,100)
// Here L creates 5 elements and stores 100 in all 5 locations
```

Sets

```
set<int>s1; //empty set
int a[] = {1,2,3,4,5,5};
```

Stacks

```
stack<int>s;
push()
pop()
top()
empty()
size()
```

Ex:
```
s.push(10);
s.push(20);
```

Display
```
stack<int>::iterator it;
```