# Procedure Oriented Programming (POP) :-

POP mainly focus on action or procedure that will take place within the program. The procedure are also called function / routine / sub routine / method. Consist of a series of computational steps that they need to carry on when a problem needs to be fixed using POP it is required to start with the problem and then logically fragment the problem down into subproblems. This process will continue until a sub procedure is simple enough to be solved by itself. If a change is required to the program the developer has to change every line of code that links to the main or the original code It follows top down approach. Example- C Basic, fortan, COBOL

Modification of code is difficult in POP

# Object oriented Programming :-

OOP follows the concept of object. The object contain code in the form of method or function and data in the form of attribute. These object can communicate with each other by passing the messag. It follows Bottom up approach. Example, java, C++, python etc It is easier to write program and manipulate them OOP. The motivation for OOP is to remove flaws of POP. It treats data as

critical element and do not allow to follow around the system. It ties data to the function that operates on it and protects it from accidental modification from outside function. The main focus is object i.e the fundamental unit

# Characteristics of POP

- Emphasis on doing things
- Large problems are divided into smaller program known as function
- Most of the function share global data
- Employs top down approach in program design

# Characteristics of OOP

- Emphasis on data than procedure
- Programs are divided into objects
- Data is hidden and cannot be accessed by external function
- New data and function can be easily added
- follows Bottom up approach for program design

# Difference b/w POP and OOP

| POP | OOP |
|---|---|
| 1) This programming language makes use of step by step approach by breaking down a task into a collection of routine and variables following a sequence of instruction | 1) OOP uses object and classes for creating models based on the real world environment. This model makes it very easy for a user to modify as well as maintain the existing code |
| 2) It do not provide data hiding hence not secure | 2) It provides data hiding hence more secure than POP |
| 3) POP divides the program into small program and refer to them as function | 3) OOP divides the program into small parts and refer them as object |
| 4) It follow top down approach | 4) It follows Bottom up approach |
| 5) It is procedure oriented | 5) It is object oriented |
| 6) It does not give importance to data. It gives priority to the function and the sequence of action that needs to follow | 6) It gives importance to data rather than function. It is because it works on the basis of real world |
| 7) Not suitable for solving any big or complex problem | 7) suitable for solving any big or complex problem |
| 8) Not easy to add new function and data | 8) Very easy to add new function and data |

9) Lacks reusability
10) ex:- C, COBOL, BASIC

9) It offers features to reuse it
10) ex - CPP, PYTHON, JAVA

# Benefits of OOP

1. Through inheritance, we can eliminate redundant code and extend the use of existing classes
2. We can build programs from the standard working modules that communicates with one another, rather than having to start writing the code from scratch. This leads to saving of development time and higher productivity
3. The principle of data hiding helps the programmer to build secure programs that cannot be invaded by code in other parts of the program
4. It is possible to have multiple instances of an object to co-exist without any interference
5. It is possible map objects in the problem domain to those objects in the program.
6. It is easy to partition the work in a project based on objects
7. The data centered design approach enables us to capture more details of a model in implementable form.
8. Object-oriented systems can be easily upgraded from small to large systems
9. Message passing techniques for communication between objects makes the interface descriptions with external system much simpler
10. Software complexity can be easily managed

# Application of OOP

1. Real time system
2. Simulation and modelling
3. Object-oriented database
4. Hypertext - Hypermedia and expertext
5. AI and expert systems
6. Neural networks and parallel programming
7. Decision support and office automation systems
   CIM/CAM/CAD systems

# Basic concept of OOP

1) Objects :- objects are the basic runtime entities in an object oriented system. They may represent a person, a place, a bank account, a table of data or any item program has to handle. Each object contains data, and code to manipulate the data.

2) Classes :- classes are user defined datatype and behave like the built in types of a programming language. It works as blue print for the object. Hence objects are variable of type class.

A class is a collection of objects ~~are variable~~ of similar types. It is similar to structure but here we have member function that operate on those data

3. Data Abstraction:- Abstraction refers to the act of representing essential features without including the background details or explanation

4. Data Encapsulation:- The wrapping up of data and function into a single unit (called class) is known as encapsulation. This insulation of the data from direct access by the program is called data hiding or information hiding Encapsulation is performed when we make a class

5. Inheritance:- It is the process by which objects of one class acquire the properties of objects of another class. It supports the concept of hierarchical classification.

6. Polymorphism:- Ability to take more than one form. It is extensively used in implementing inheritance example:- Operator overloading (The process of making an operator to exhibit different behaviours in different instances is known as operator overloading function Overloading (using a single function name to perform distinct types of tasks is known as function overloading)

7. Dynamic Binding:- Binding refers to the linking of a procedure call to the code to be executed in response to the call. Dynamic Binding means that code associated with a given procedure is not known until the time of the call at run time. It is associated with polymorphism and inheritance

# C++
→ C++ is an object oriented programming language
→ C++ is super set of C
→ It was developed by Bjarne Stroustrup at AT & T bell laboratories in Murray Hill, New jersey, USA in 1979.
→ C++ has rich library
→ C++ does exception handling
→ C++ is hybrid langue (as POP is also involved)
→ C with classes was original name of C++
→ A language is called fully object oriented when
- Polymorphism
- Inheritance
- Encapsulation

Hence C++ is not pure object oriented
→ (Since main is not encapsulated in the class)
→ (C++ has C, and C support global variables) i.e any function can reach to any data, that is no security is there for the data)

⇒ (without object and class, program can be runned)

⇒ C++ Keyword = class, new, operator, public, private

⇒ C++ is developed by = BJARNE STROUSTRUP (1979) at Bell Laboratories in Murray New jersey, USA

⇒ Additional element = OOP, Exception handling, Rich libraries

⇒ C++ is not pure object oriented program because (main), data hiding, hello world program without class & object

⇒ Identifier :- It refers to the name variable, function, classes etc

⇒ Constant :-

⇒ Datatypes :-

| • User define | • Primitive | • Derived |
|---|---|---|
| structure, union, enum, class | int, float, char, double | array, function, pointer, function |

⇒ Enum week {Monday, Tue, wed, Thur, fri, sat, sun}

```
main ()
{ enum week day;
  day = friday;
  cout << day++;
```

⇒ Reference datatype :-

Syntax :- type & name;

```
int x; int &y = x;
x = 5;
```

```
int x; int &y = x
x = 5;
x++;
cout << x;
cout << y;
```

⇒ Operators in C++

- Scope resolution operation :- ::
- Memory dereferencing operator :- a) ::* b) .* c) ->*
- Memory management operator :- a) new b) delete
- Manipulator :- a) endl b) Set w()

a) ex = `cout << "the value is :" << endl << x;`

Output :- the value is 5

b) ex = `sum = 236.`

`cout << Set w(5) << sum;`

output :-

| | | | 2 | 3 | 6 |
|---|---|---|---|---|---|

- Type cast operator :- `sum = x/(float)i;` // In C

`sum = x/float(i);` // In C++

⇒ Control structure :- 1) if 2) if else 2) if else ladder 4) while 5) do while 6) switch

# Functions:-

```
void show (int x)    // function prototype
show (b)             // function call
Void show (int x)
{ printf ("%d", x);  } - function defination
```

here b is an argument and x is a parameter

```
Void show (int p, int q=2)   // default initialization in function
```

Q) Write a program to find the Area of rectangle

```
# include <iostream.h>
# include <conio.h>
  void area_of_rectangle (float x, float y);
  void main ()
{ float l, b;
  clrscr ();
  cout << "Enter the length :";
  cin >> l;
  cout << "Enter the breadth :";
  cin >> b;
  area_of_rectangle (l,b);
}
  void area_of_rectangle (float x, float y)
    { float ar;
      ar = x*y;
      cout <<ar;
    }
```

# Structure in C++

→ In C++ we can also add function in structure

→

| In C | In C++ |
|---|---|
| main () | main () |
| { struct ABC abc;} | { Abc abc;} |

→ In C++ while using structure we can use access modifier

```
Struct ABC
{ int x;     } These are public in C so we can use
  float y;   } it in any place but in C++ they are
}              private
```

⇒ Security and data hiding in C++ structures

- Access modifier
- No need to write Struct word
- public ⇒ default

# Classes in C++:- It is a user defined data type
It is a blueprint of object

syntax:-

```
class classname
  { access specifier:
      data member;
    access specifier:
      member function ();
  }
return type classname :: function name ()
```

⇒ access specifier :- In c++ these are used for determining or setting boundaries for the availability of class members beyond the class This is private, public and protected

- Private = member cannot be acessed from outside the class
- Public = members are acessible from outside the classe.
- Protected = Members cannot be acessed from outside the class however they can be acessed in inherited class

example :-

```
Class myclass
{ private :
    int x;
  public:
    void show (int y)
};
void myclass :: show (int y)
{ x=y
  cout << x;
}
void main ()
{ myclass obj;
  obj. show (5);
  getch ();
}
```

when we creat a class then encapsulation is done.

object = Variable of class is called object

⇒ Instantiation = When we create the object of class then the method is called Instantiation and object is said to be instance

⇒ Allocation of memory :- When we create the object then the memory is allocated to data members and when we create a class then the memory is allocated to function.

⇒ Instantiation = Instantiation is when a new instance of class is created (an object). In c++ when a class is instantiated, memory is allocated for the object and class constructor is run.

In c++ we can instantiate object in two ways :- on stack as variable declaration or in heap with 'new' operator

# Typecasting :- Typecasting refferred to the conversion of one data type to another

⇒ Types of Typecasting :-

1) Implicit (It is done by the compiler)
2) explicit (It is done by the programmer)

1. Implicit typecasting :- It is automated type casting. It automatically converts from

one data type to another without any intervention such as programmer or user. It means compiler automatically converts one data to another.

Char → short int → unsigned int → long int → float → double → long double etc.

Implicit type casting should be done from low to higher data type

2. Explicite typecasting :- It is also known as manual typecasting. It is manually cast by programmer or user to change from one data type to another in a program

⇒ Typecasting Operator :- ()

⇒ Example of Implicit typecasting

```
# include <iostream.h>
  void main ()
{ short int x = 200;
  int y;
  y = x;
  cout << "Implicit typecasting" << endl;
  cout << "value of x : " << x << endl;
  cout << "value of y : " << y << endl;
  int num = 20;
  int res = 20 + 9;
  cout << " type casting char to int data type
        ('9' + 20) : " << res << endl;
  float val = num + 'A';
  cout << " type casting from int to float type :"
       << val << endl;
```

⇒ Example of Explicit typecasting

```
# include <iostream.h>
  void main ()
{ int num;
  num = (int) 4.534;
  cout << num;
  float res;
  cout << " explicit type casting " << endl;
  res = (float) 21/5;
  cout << " value of float variable (res)"
       << res << endl;
```

# function Overloading

```
#include <iostream.h>
 int square (int x)
{ cout << " Inside integer square";
  return (x * x);
}
 void square (double d)
{ cout << " Inside double square";
  cout << (d * d);
}
```

```
Void Square (float f)
{ cout << "Inside float Squar ";
  cout << (f*f);
}
 Void main ()
{ int a=5, P;
  float b = 5.5f;
  double c = 6.5;
  P= Square (a); cout <<P;
     Square (b);
     Square (c);
     getch ();
}
```

Ques) Write a program to add two Numbers of integer float and double by using function overloading

Sol"
```
#include <io.stream>
  int add (int x)
  { cout << "Inside integer add";
    return (x+x);
  }
  void float add (float f)
  { cout << "Inside float add";
    cout << (f+f);
  }
  void add (double d)
  { cout << "Inside double add";
    cout << (d+d);
  }
```

```
Void main ()
{ int a= 5, P;
  float b= 5.5f;
  double c = 2.5;
  P= add (a);    cout << P;
     add (b);
     add (c);
  getch ();
}
```

# Inline function = When function is define under the class then we call it inline function. It is used when function is small.

ex:-
```
inline void Show ()
  { cout <<" Hellow world !";
  }
  Void main()
{ cout << "This is Inline function program";
  Show ();
  cout <<" The message hellow would is
          printed above";
  Show ();
  cout <<" The same message is printed again";
  Show ();
  cout <<" Last time this message is printed";
}
```

function call is replaced by statement of function during compile time

C++ provides an inline function to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call.

⇒ Second case:-

```
class A
{ private:
   int x, y;
  public:
   void show()
     { cout << "Hello";
     }
};
```

This will be treated as inline function

⇒ Third case:-

```
Class A
{ int x, y;
  public:
   void show();
};
inline void A :: show()
   { - - - -
     - - - -
     - - - -
   }
```

| ⇒ C Structure | C++ Structure |
|---|---|
| 1. Struct Keyword mandatory | 1. Struct keyword not mandatory |
| 2. Access modifier is not used | 2. Access modifier is used |
| 3. Static variable not used | 3. Static variable used |
| 4. Only variable | 4. Variable + function used |

⇒ C++ Structure & class Similarity

- Both have Access Specifier
- Both have static variable.
- Both have variable + function

| ⇒ C++ Structure | C++ class |
|---|---|
| • If access modifier is not given then by default treated as public | • If access modifier not given then by default treated as private |
| • It is concept of C | • It is concept of C++ |

⇒ Static :- static data members are also called as class variable.

If we have used static keyword then the initialization will be with '0' by default

ex:-

```
# include <iostream.h>
# include <conio.h>
  class A
  { static int Y;  // It is static & private then
    public:        initialize it into main.
```

```
static int x;          // If is static & public then
public:                   initialize into main
void show_x()
{ x++;
  cout << x << endl;
}
void show_Y()
{ Y++;
  cout << y << endl;
}
};
int A::x;
int A::y=2;
void main()
{ A::x = 10;           // Initialize x (if not then 0
  A a1, a2;               value again)
  clrscr();
  a1.show_x();
  a2.show_Y();
}
```

# Static function :- static function can only access static variable

```
static void show()

class Test
{ int non_var;
  static int stat_var;
public:
 void show_non_var()
  { non_var = 5;
    cout << non_var;    //5 //5 //5
  }
static void show_stat_var()
  { stat_var ++;
    cout<<stat_var;     //1  //2
  }
};
int test :: stat_var;  //Initialize static variable
void main()
 { Test t1, t2;
   t2.show_non_var();
   t2.show_non_var();
   Test::show_stat_var(); // calling static function
   Test t3;                  with class name &
   t3.show_non_var();        scope resolution
   Test::show_stat_var();    operator
 }
```

# Constructors :- It is a special member function whose task is to initialize the object of its class. It is special because its name is same as that of class because it construct the value of data member

1) It should be declared in public section.

2) They are invoked automatically when object is created
3) They do not have return type not even void
4) They can't be inherited
5) Constructor can't be virtual
6) We can not refer to their address

⇒ Default Constructor :-

```
class A
{int x;
public:
  A()
{ x= 5; }
  void show()
{ cout << "x = " <<x; }
  };
void main()
{ A a;
  a. show();
}
```

⇒ Parameterised Constructor :-

```
Class A
{int x;
public:
  A(int w)
{ x= w; }
  void show()
{ cout <<"x = " <<x; }
  };
  void main()
{ int k;
  cout <<" Enter k:";
  cin >> k;
  A a(k);
  a. show();
}
```

Other thing will remain same

```
void main()
{ A c= A(20); // explicit call
  A a(10), b(20); // Implicit call
  a. show();
  b. show();
}
```

⇒ Constructor Overloading

```
#include <iostream.h>
#include <conio.h>
  class P
{ float x;
  float y;
  Public:
  P(float a) {
    x= a; }
  P(float w, float v)
  { int c;  x=w; y=v;
```

```
    cout << " I am going to be a rectangle;
}
P()
{ X = 2.1;
  Y = 3.1;
}
void main()
{ A a (5.1, 5.2);
  A b();
}
```

⇒ Copy Constructor:-

```
# include <iostream.h>
  Class A
  { int x;
  public:
    A()
    {X = 7;}
    A (A &y)
    { X = y.X;
      cout << " Copy Constructor invoked & variables initialized
    }
    void Show()
    { cout << x;}
  };
  void main()
  { A a;
    A b = a; // Or A b(a);
    a.show();
    b.show();
  }
```

⇒ To allocate memory:-

```
void main()
{ my class a1; //stack
  a1.show(2);
  my class *a2; //Heap
    a2 = new A();
    a2 → show(5);
    getch();
}
```

- new operator is used to allocate the memory in heap

⇒ defination of Copy Constructor :- In the C++ programming language, a copy constructor is a special constructor for creating a new object as a copy of an existing object. Copy Constructor are the standard way of copying objects in C++, as opposed to cloning, and have C++ specific nuances.

# Destructor :- syntax :-

```
~Classname();
```

```
class A
{ int x;
public ;
  A ()
{x = 5;}
A (A &b)
{ x= b.x;
 cout <<x<< endl ;
 Cout << " Copy Constructor;
}
 ~ A()
 { cout << " Destructor called ";
 }
};
 void f( A a3)
{ Cout << " Inside function"; }
 void main ()
{  A a1
 A a2(a1) ; // or A a2=a1;
 f(a1);
}
```

Output :-
```
default Constructor
5
copy constructor
5
Copy Constructor
Inside function
destructor called
destructor called
destructor called
```

⇒ Conditions when copy constructor is called

- A a= b;
- when we pass Object argument in function
- while returning object from a function

```
#include <iostream.h>
 class A
{ int x;
 Public:
  A()
 { x=2; }
 A (A &c)
 A f1 (A d);
 { A t
  t.x= x+d.x;
  return t;
 }
};
 A::A(A &c)
 { x= c.x;
  cout<<" Copy Constructor called ";
 }
  void main ()
 {  A a1, a2;
```

A a3 = a2.f1(a1);
}

# Reference variable :- C++ has a new kind of a variable known as reference variable.
A reference variable provide an Alias (alternative name) for a previously defined variable
Syntax :-
data type &ref_variable = another variable
A reference variable must be initialised at the time of declaration.

| Reference Variable | Pointer |
|---|---|
| 1. It is an alias to another variable | 1. It holds the memory address of another variable |
| 2. A reference variable cannot have a null value assigned | 2. Null values can be assigned to pointer |
| 3. Syntax :- data type &ref_variable = another variable | 3. Syntax :- Data type *Pointer_variable |
| 4. Reference variable can never be created un-initialised | 4. Pointer can be created uninitialised. |
| 5. Reference variable can be initialised only once | 5. Pointer can be reinitialised |
| 6. NO pointer arithmetic can perform | 6. Pointer arithmetic can be performed |

ans) Write a program of pass by value, pass by reference and pass by address.

Sol^n
```
# include <iostream.h> void pass_by_value (int x, int y)
{ int z=x;
  x=y; y=z;
}
void pass_by_reference (int &x1, & int &y1)
{ int z=x1;
  x1=y1; y1=z;
}
void pass_by_address (int *x2, int *y2)
{ int z= *x2;
  *x2= *y2;
  *y2= z;
}
void main ()
{ int a= 5, b=6;
cout << " before swapping " <<a<<b;
  Pass_by_value (a, b);
Cout << " after pass by value" << a<<b;
  Pass_by_reference (a,b);
Cout << "after pass by reference" <<a<<b;
  pass_by_address (&a, &b)
cout << " after pass by address " <<a<<b;
}
```

# friend function :-

It is not in the scope of class.
They cant be used with the object of class
Mostly they take object of class as a paramater

⇒ Example 1:-

```
class Box
{ double width ();
  public :
     Box ()
     { width = 5; }
  friend void show width (Box b);
};
void show width (Box b)
{ cout << " width of Box " << b. width;
}
void main ()
{ Box b1;
  show width (b1);
}
```

⇒ Example 2:-

```
class B; // This is forward declaration
class A
{ Public :
   void show (B b);
};
class B
{ int x;
  Public :
   friend void A :: show (B b);
};
void A :: show (B b)
  { b.x = 2;
    cout << b.x;
  }
void main ()
{ cout << " friend function ";
   A a;
   B b;
   a. show (b);
}
```

Ques) Write a program to make void compare function as a friend to both the classes i.e A & B

```
class B;
class A
{ int x;
  Public :
   friend void compare (A a, B b);
};
class B
{ int y;
  Public :
   friend void compare (A a, B b);
};
```

```
void compare (A a, B b),
{ if (a.x > b.y)
  { cout << " a is greater than b";
  }
  else {
     cout << " b is greater than a";
    }
}
void main ()
{  A a;
   B b;
   void compare (a, b);
}
```

⇒ Pointer :- If you want to create a pointer for data member of class then syntax will be
return type classname :: n * P = & classname
:: data member of class

⇒ Example :-

```
class A
{ int m;
  public :
   void show ();
};
int A :: *P = &A :: m;
```

NOTE= '->*' it is a dereferencing operator

⇒ Example 3 :-

```
class A
{ int x, y;
  public :
    A ()
    { x=2; y=3;
    }
    friend void show (A a);
};
void show (A a)
  { int A :: *Px = &A :: x;
    int A :: *Py = &A :: y;
     A * Pm = &a;
    cout << (a *Px) + Pm ->*Py);
};
void main ()
{ A a1;
  show (a1),
  getch ();
}
```

# friend class = Like a friend function a class can also be friend of another class a friend class can acess all the private and protected member of other class. In order to acess the private and protected member of a class into friend class we must pass an object of a class to the member function of friend class

example :-

```
class Rectangle
{ int l, b;
  Public:
  friend class Square;
};
class Square
{ int side;
  public:
    void get_valued (Rectangle R1)
      { side = 3;
        R1.l = 2;
        R1.b = 4;
      }
    Void display (Rectangle R2)
      { cout << " length of rectangle = " << R1.l;
        cout << " breadth of rectangle = " << R1.b;
        cout << " side of Square = " << side;
      }
};
void main ()
{ Rectangle R;
  Square S;
  S. get_valued (R);
  S. ~~get_valued~~ display (R);
}
```

⇒ Delete Keyword = It is used to delete the memory of the object & data types

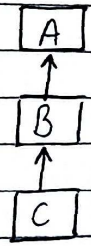
# Inheritance :- The mechanism of deriving a new class from an old one is called 'Inheritance' The old class is called base class and new class is called derived class. OOP provides reusability with its one of the important feature called Inheritance.

Reusability saves time, money and effort

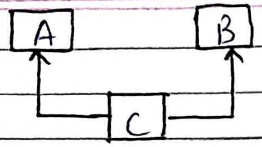
⇒ Types of Inheritance :- There are of 5 types

1) Single inheritance / Simple inheritance :-

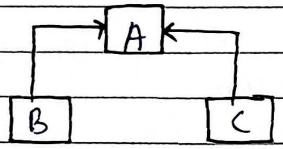

2) Multilevel Inheritance



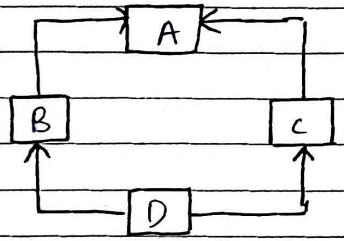

3) Multiple inheritance




4) Heirarchical Inheritance




5) Hybrid inheritance




⇒ Syntax for inheritance :-

```
class derived class : visibility mode
                      Base class
{
};
```

visibility mode can be public, private, protected

⇒ Example of "Single Inheritance"

```
class B
{ int a;
  Public:
  int b;
  void get a-b()
  { a=5; b=10; }
  int get.a
  { return a; }
  void show a()
  { cout << " a=" << a; }
  };
class c : public B
  { int c;
    public:
      void mul();
      void display();
  };
    void c:: mul()
      { c = b * get_a();
      }
    void c:: display()
    { cout << "a=" << get_a();
      cout << "b=" << b;
      cout << "c=" << c;
    }
    void main()
    { C c;
      c. get a-b();
```

c.mul();
c.show();
c.get a();
c.display();
}

⇒ Protected access specifier = What do we do if private data member need to be inherited by derived class. This can be accomplished by modifying the visibility limit of the private members by making it public. This would make it accessible to all the other functions of the program. Thus taking away the advantage of data hiding.

C++ provides 'protected' which serves limited purpose. A member declared as protected is accessible by member function within its class and any class immediately derived from it. It cannot be accessed by a function outside these two classes.

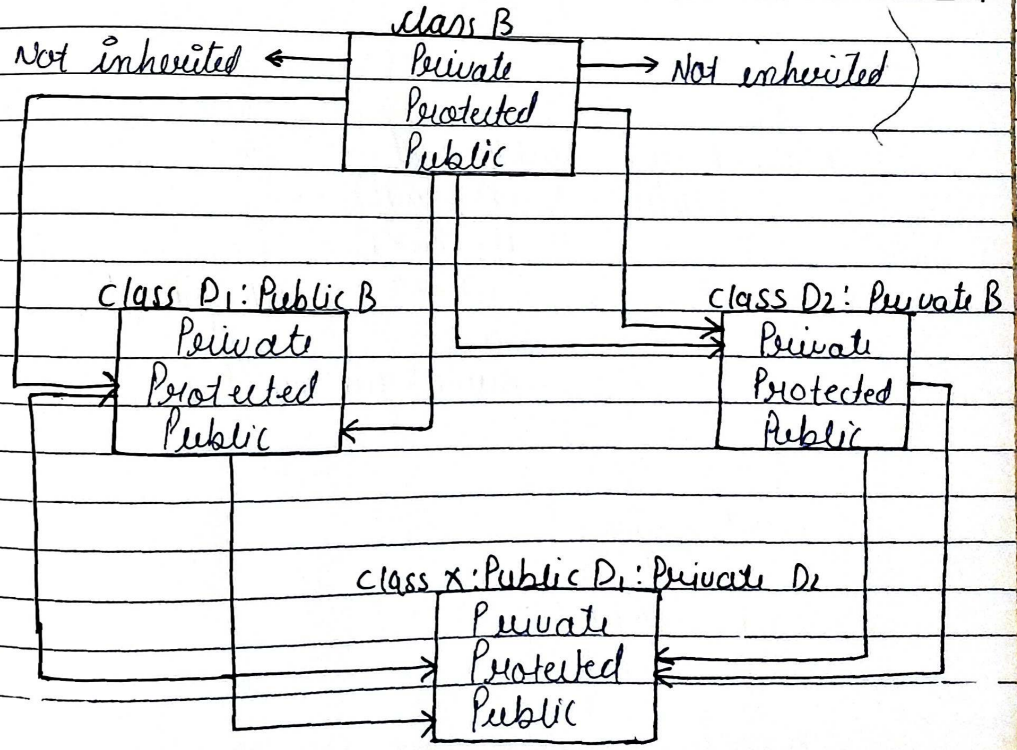
NOTE = Agar hum ye access specifier use karte hai to data member or member function dono ko base class or derived class mai access kar sakte hai.

class A
{

private : // Visible to member function of its class
protected : // Visible to member function of its own & derived class.
public : // Visible to all function in program.

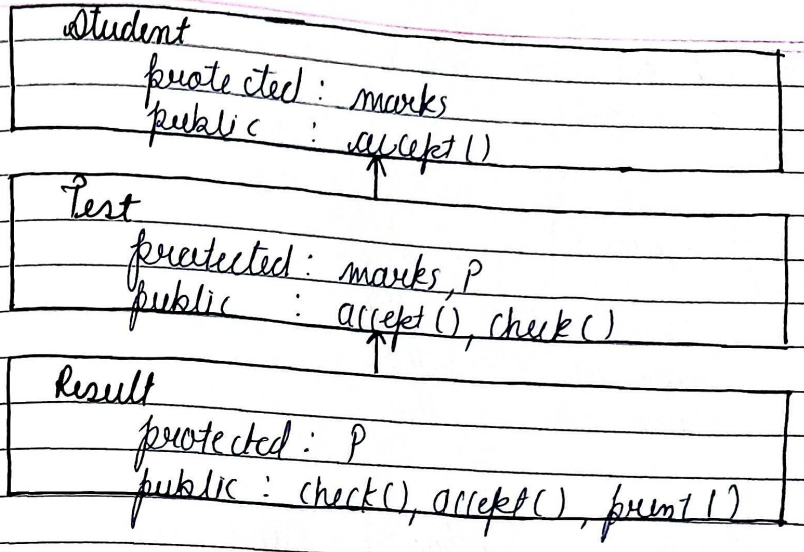
⇒ Derived class visibility

| Base class member | Public | Private | Protected |
|---|---|---|---|
| Private | Not Inherited | Not Inherited | Not Inherited |
| Protected | Protected | Private | Protected |
| Public | Public | Private | Protected |
| | | | |



⇒ Example of Multilevel Inheritance

```
class Student
 { protected : int marks;
   public   : void accept();
              { cout << "Enter marks : ";
                cin >> marks;
              }
};
class Test : public Student
   { protected : int p=0;
     public : void chek()
              { if (marks > 60)
                { p=1; }
              }
   };
class Result : public Test
   { public : void print()
              { if (p==1)
                cout << " you have passed ";
                else
                cout << " you are failed ;
              }
   };
void main()
{ Result R;
  R. accept ();
  R. chek ();
  R. print ();
}
```




⇒ Example of multiple inheritance

```
class A
{ int x = 1;
  public : void show_x()
           { cout << x; }
};
class B
{ int y;
  public : void show_y()
           { y = 2;
             cout << y;
           }
};
class C : public A : public B
  { int z;
    public : void show_z()
```

```
    { z=6;
      cout << z;
    }
};
void main()
{ C c;
  c.show_x();   //1
  c.show_y();   //2
  c.show_z();   //6
}
```

Note :- If the name of function is same then

```
C.A :: show();
C.B :: show();
C.C :: show();
```

⇒ Example of Hybrid Inheritance

```
class A
{ public:
    int b;
};
class B: public A
  { public: int d1;
  }
class C: public virtual A
  { public: int d2;
  };
class D: public B, public C
  { public: int d3;
  };
void main()
{ D d;
  d.b = 10;
  d.b = 30;
  d.d1 = 50;
  d.d2 = 25;
  d.d3 = 20;
  cout << d.b;
  cout << d.d1 << d.d2 << d.d3;
}
```

⇒ Example of Hierarchical inheritance.

```
class A
{ protected:
    int x;
    int y;
  public:
    void get_data()
    { x=1;
      y=1;
    }
};
class B: public A
  { public:
```

```
void show ()
 { cout <<x<< endl;
   cout << y<< endl;
 }
};
class C : public A
 { public:
    void sum ()
    { int sum;
      sum = x+y;
      cout << sum << endl;
    }
};
void main ()
{  C a;
   a. get_data ();
   a. sum ();
   B b;
   b. get_data ();
   b. sum ();
   getch ();
}
```

# Constructors with Inheritance

1) If any class contains constructor with one or more arguments then it is mandatory for derived class to have a constructor and pass the argument to the base case constructor

2) When both derived and base class contains constructor the base class constructor is executed first and then the constructor in derived class is executed

ex :- A {
    int x, y; }
  B {
    float a, b; }
  C { int d;
  C(int a1, int a2, float f1, float f2, int d1) : A(a1, a2),
                                                  B(f1, f2)
    { d = d1; }
main ()
{ C c(1, 2, 4.2f, 3.5f, 7);
}

| Inheritance | Order of execution |
|---|---|
| 1) class B : Public A | 1) firstly A() will execute then B() |
| 2) class A : Public B, Public C | 2) firstly B() will execute then C() and then A() |
| 3) class A : Public B, Virtual Public C | 3) firstly C() will execute then B() and then A() |

# Method for initialisation of Constructor

⇒ Constructor (Argument : initialization)

```
{ Assignment section;
  }
```

```
class XYZ
  { int a,b;
    public : XYZ (int i, int j) : a(i), b(2*j)
      {
      }
```

⇒

```
XYZ( int i, int j)
 { a = i ; b = j; }
```

⇒

```
XYZ (int i, int j): b(i) ,a (i+j)
  {
  }
```

# Operator Overloading :-

Operators that we can't overload are as follows-

1) Member selection (.)
2) Member selection through pointer to function (.*)
3) Scope resolution (::)
4) Size of
5) Conditional operator (?:)

⇒ Overloading '+' operator :- 1) Without friend function

```
class Box
 { int l, b, h;
   public:
     Box (int l1, int b1, int h1)
     { l = l1 ; b = b1 ; h = h1 ; }
     Box () { }
     Box operator + (Box B);
       void display ();
 };
 void Box :: display ()
  { cout << "The length is =" << l ;
    cout << " The breadth is =" << b;
    cout << " The height is =" << h;
  }
 Box  Box :: operator + (Box B)
   { Box t;
     t.l = l + B.l;
     t.b = b + B.b;
     t.h = h + B.h;
     return t;
   }
   void main ()
   { Box b1(1,2,3) , b2 (4,5,6) , Big_box;
     Big_box = b1 + b2   // or b1. operator (b2)
     cout << "dimension of Big box are :";
     Big_box. display ();
   }
```

Invoking object — Passed object

2) With friend function

```
class Box
{ int l, b, h;
  Box () {}
  Box (int l1, int b1, int h1)
  { l = l1; b = b1; h = h1;
  }
  friend Box operator + (Box b1, Box b2);
  void display ()
  { cout << "The length is =" << l;
    cout << "The breadth is =" << b;
    cout << "The height is =" << h;
  }
};
  Box operator + (Box b1, Box b2)
  { Box t;
    t.l = b1.l + b2.l ;
    t.b = b1.b + b2.b ;
    t.h = b1.h + b2.h ;
    return t;
  }
  void main ()
  { Box b3 (4,5,6), b4 (7,8,9), b5;
    b5 = b3 + b4;
    b5.display ();
  }
```

⇒ overloading '-' operator :-

1) With friend function

```
class Box
{ int l, b, h;
  Box () {}
  Box (int l1, int b1, int h1)
  { l = l1; b = b1; h = h1; }
  friend Box operator - (Box b1, Box b2);
  void display ()
  { cout << "The length is =" << l;
    cout << "The breadth is =" << b;
    cout << "The height is =" << h;
  }
};
  Box operator - (Box b1, Box b2)
  { Box t;
    t.l = b1.l - b2.l;
    t.b = b1.b - b2.b;
    t.h = b1.h - b2.h;
    return t;
  }
  void main ()
  { Box b3 (7,8,9), b4 (4,5,6), b5;
    b5 = b3 - b4;
    b5.display ();
  }
```

2) Without friend function

```
class Box
{ int l,b,h;
  Box (){};
  Box (int l1, int b1, int h1)
  { l=l1; b=b1; h=h1;}
  Box operator-(Box B);
  void display ();
};
Void Box :: display ()
{ cout << "The length is =" << l;
  cout << "The breadth is =" << b;
  cout << "The height is =" << h;
}
Box Box :: operator-(Box B)
{ Box t;
  t.l = l - B.l;
  t.b = l - B.b;
  t.h = l - B.h;
  return t;
}
void main ()
{ Box b1(7,8,9), b2(4,5,6), b3;
  b3 = b1 - b2; // or b1.operator(b2)
  b3.display ();
}
```

⇒ Overloading '=' operator

1 with friend function :-

```
class Box
{ int l,b,h;
  Box (){};
  Box (int l1, int b1, int h1)
  { l=l1; b=b1; h=h1;}
  friend Box operator = (Box b1, Box b2);
};
Box operator = (Box b1, Box b2)
{ b1.l = b2.l;
  b1.b = b2.b;
  b1.h = b2.h;
  cout<<" objects are assigned;
}
void main ()
{ Box b3(7,8,9), b2;
  b2 = b3;
}
```

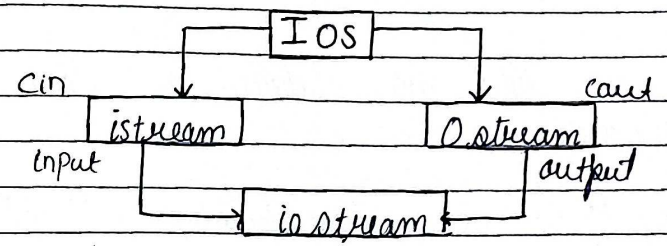
⇒ Overloading increment operator

```
class Incr
{ int x,y;
  public:
    Incr ()
    { x=2; y=2;}
  Incr operator ++()
    { Incr t;
```

```
t.x = ++x;
t.y = ++y;
return t;
}
Incr operator++(int)
{ Incr t;
  t.x = x++;
  t.y = y++;
  return t;
}
void display()
{ cout << x << y; }
};
void main()
{ Incr P,q;
  Cout << " Before preincrement ";
  P.display(); ++P;
  Cout << " After preincrement ";
  P.display();
  Cout << " Befor post increment ";
  q.display();
  Cout << " After post increment ";
  q++;
  q.display();
}
```

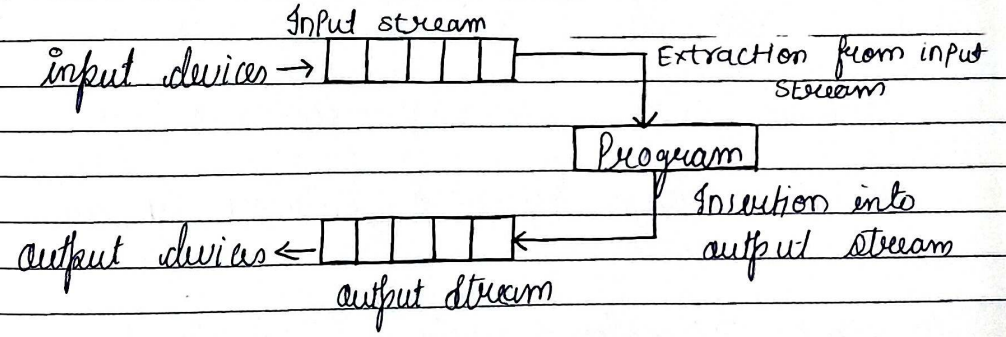
Compiler wants ki jaha aap pre increment function bana rahe ho to use koi parameter de or agar post increment k liye function define kar rahe ho to usme ek parameter dedo taki Compiler compiler dono function mai distinguish kar sake.

⇒



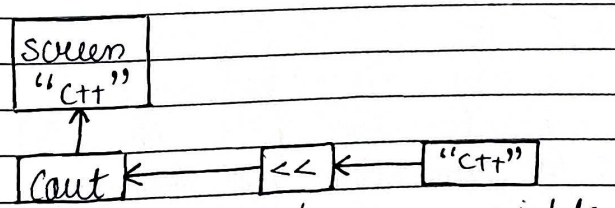

Cin is the objut of istream
Cout is the objut of ostream

⇒

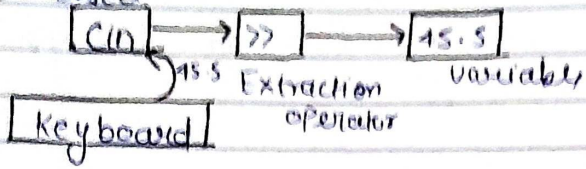



NOTE:- Stream is a sequence of bytes

⇒



v) → object




NOTE :- cin and cout are declared as extern objects in iostream

# Overloading ">>" (extraction operator) & "<<" (Insertion operator)

```
class fraction
{int num, deno;
 public:
 friend ostream &operator << (ostream &out, fraction &F2);
 friend istream &operator >> (istream &in, fraction &F2);
};
ostream &operator << (ostream &out, fraction &F2)
{
   out << F2.a << "/" << F2.b;
   return out;
}
istream &operator >> (istream &in, fraction &F2)
{ cout << "Enter numerator & denomenator :";
   in >> F2.a;
   in >> F2.b;
   return in;
}
void main ()
{ fraction F1, F2;
   cin >> F1 >> F2;
   cout << F1 << F2;
   getch ();
}
```

# Stream= A stream is a sequence of byte it acts as a source from which the input data can be obtained or as a destination to which output data can be sent.

The source stream that provides data to the program is called input stream, and the stream that recieves output from the program is called output stream

The data in the input stream can come from keyboard or any other input data similarly the data in the output stream can go to the screen and any other storage devices

# function overriding :- function overriding is changing the definition of base class function by derived class having relationship of inheritance b/w them.

2)

```
class Base
{ public :
    void show()
    { cout << " I'm Base"; }
};
class Derived : public Base
{ public :
    void show()
    { cout << " Im derived "; }
};
void main()
{ Derived d;
  d.show();
  d.Base :: show();
}
```

Output :- I'm Base
I'm Derived

⇒ Difference b/w function overriding & function overloading

| function overloading | function overriding |
|---|---|
| 1) function overloading is performed in same class | 1) It is performed in diff. classes |
| 2) No need of Inheritance | 2) Inheritance is must |
| 3) function name and parameter list are important for function overloading | 3) Return type, function name & parameter list all are important |

# Virtual function :-

```
Class Base
{ public :
    void show();
  { cout << " function of Base"; }
};
class Derive : public Base
{ public :
    void show()
    { cout << " function of Derive"; }
};
void main()
{ Base * P;
  Base b;
  Derive d;
  P = &b;
  P->show();   // Output :- function of Base
  P = &d;
  P-> show();  // output :- function of Base
}
```

agar Base class k pointer ko use karte hue derived k function ko call karna ho to virtual keyword use karenge function k aage

```
  Class Base
  { public :
virtual void show()
  { cout << " function of Base"; }
};
```

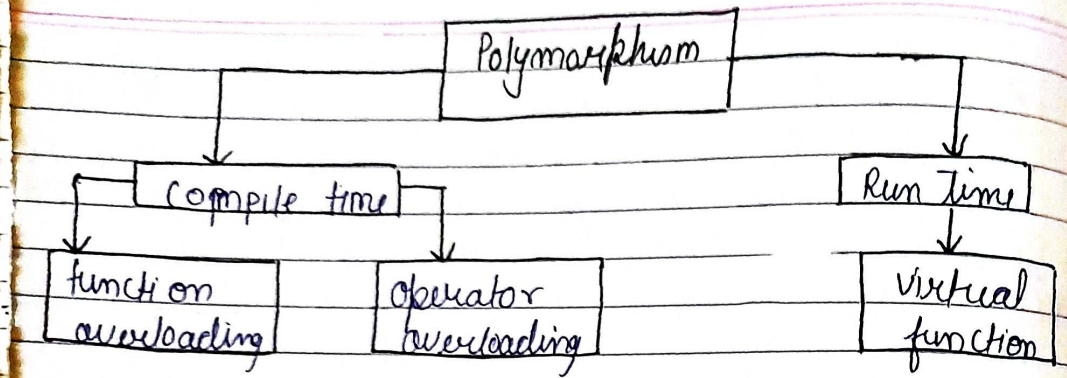
class Derive : public base
{public
void show()
{cout << "function of Derived;}
};

⇒ Run time polymorphism
- function must be overrude
- Base class must have pointers
- ex:- function overriding

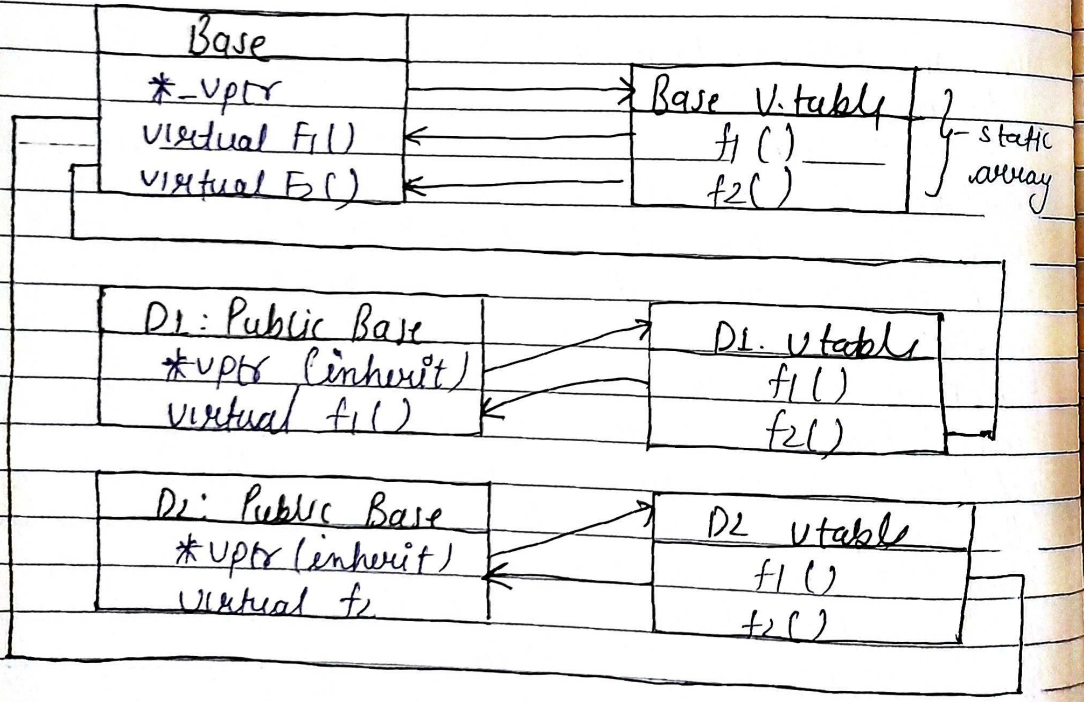
⇒ Virtual function :- When we use some function name in the both, the base & the derived class the function in the base class is declared as virtual using the keyword "Virtual" preceeding its normal declaration. When a function is made virtual C++ determines which function to use at run time based on the type of object pointed by the base pointer rather than the type of pointer. Thus by making base pointer to point different objects we can execute diff. version of virtual function

⇒ Compile time polymorphism / early binding / static binding / static linking
The concept of polymorphism is implemented using overloaded function & operator overloading. In overloaded function, member function are selected for invoking by matching argument both type and number. This information is known to the compiler at the compile time and therefore compiler is able to select the appropriate function for a particular call at compile time itself. This is called early binding / static binding / static linking it is also known a compile time polymorphism Early binding simply means that an object is bound to function call at compile time

⇒ Run time polymorphism / late binding / dynamic binding / dynamic linking
C++ supports a mechanism known as virtual function to achieve runtime polymorphism. At run time when it is known what class object are under consideration, the appropriate version of function is invoked. Since the function is linked with a particular class much later after the compilation this process is termed as late binding it is also known as dynamic binding because the selection of appropriate function is done dynamically at run time. Dynamic binding is one of the powerful feature of C++ that requires the use of pointer to object




⇒ Mechanism behind the virtual function




```
Class A
{ virtual f1();
  virtual f2();
};
```

```
Class D1 : A
{ virtual f1()
  {
  }
};
Class D2 : A
{ virtual f2()
  {
  }
};
void main()
{ Base *P
  D1 d1;
  D2 d2;
  P= &d1;
  P-> f1;
  P-> f2;
  P= &d2;
  P-> f1;
  P-> F2;
}
```

⇒ 'THIS' Operator

```
class S
{ int x
  Public :
    sum (S s3)
    { x = x + s3.x
      cout << x;
    }
```

```
}; void main ()
{
    S S1, S2;
    S1. sum(S2);      //S1 is current object
}
```

- this pointer = C++ uses a unique keyword called 'this' to represent an object that invokes the member function. 'This' is a pointer that points to the object for which the function is called. 'This' is a unique pointer that is automatically passed to a member function when it is called. The pointer 'This' acts as an implicit argument to all the member function

```
Class A
{ int x, y;
  Public: A (int x, int y)
      { this → x = x;
        this → y = y;
      } void show (); greater (A a); };
A*A:: greater (A *a)
    { if (((this → x) > (a → x)) && (this → y) > (a → y))
        return this;
      else
        return a;
    }
```

```
voide A:: show ()
    { cout << x << y; }
void main ()
{ A a1(3,5), a2(2,4);
  a1. greater (&a2) → show ();
  a1. greater (&a2) → show ();
}
```

⇒ Pure Virtual function

```
Class Rectangle
{ Public:
    void show () { ==- }
    void area () { ==- }
    virtual void volume () = 0; // pure virtual functa
};
```

A pure virtual function is a function declared in a base class that has no definition related to the Base class. In such cases the compiler requires each derived class to either define the function or redeclare it as pure virtual function. A class containing pure virtual function can't be used to declare any object of its own. Pure virtual function only serves as place holder

# Abstract class

```
class Davvuniversity
{ virtual void admission ()=0;
  virtual void result()=0;
  void migration ()
    { ---- ;}
};
class Affiliated college: Davvuniversity
{ void admission ()
   { must ; }
   void result () { ---- ;}
};
class UTD: Davvuniversity
{ CET ; }
  void result(){ CGPA ;}
};
```

A class containing pure virtual function can not be used to declare any object of its own and such classes are called Abstract class. the main objective of an Abstract class is to provide some traits to the derived class and to create a base pointer required for achieving run time polymorphism. We can create pointer and reference variable of abstract Base class that may point to instance of its child class. Abstract class can have normal / concrete function and variables. Abstracts class are mainly used for upcasting (Base class K object mai derived class ka address assign karna is upcasting)

# Exception handling

There are some problem which are other than logic or syntax errors they are known as exception. Exceptions are run time or unusual condition that a program may encounter while executing. These anomalies might include conditions such as divide by 0, acess to an outside of its bound etc. When a program encounters an exceptional condition it is imp. that it is identify and dealt with effectively. Exceptional Handling was not a part of original C++ but today almost all compiler supports this feature except Turbo C++.

Exceptions are of two types Synchronous and Asynchronous. Errors such as out of range index and overflow belong to Synchronous exception.

And the errors that are caused by events beyond the control of program (such as keyboard Interrupts) are called Asynchronous exception.

Three Keywords by which we can handle exception are
1) try { ---- }
2) Throw ( Ye try K andar hi likha jata hai )
3) Catch

```
try
{
  ---
  ---
  throw exceptional object
}
catch (  )
{
  ---
  ---
}
```

C++ exception handling mechanism is build upon three keywords namely try throw and catch.

⇒ Try is used to prefall a block of statement which may generate exception. The block of statement is known as 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 appropiately. The catch block that catches an exception must immidiately follow the try block that throws the exception

The general form of these two blocks are as follows:-

```
try
{
  ---
  throw exception;
  ---
}
catch (type arg)
{
  ---
  ---
}
```

NOTE:- Agar exception aa gya to throw k niche wale ~~exception~~ statement nahi execute hongy.

Example:-

```
{ int a,b;
  cout << "Enter value of a and b";
  cin >> a;
  cin >> b;
  int x = a-b;
  try {
    if (x!=0)
      { cout << "Result (a/x) << a/x; }
    else {
      throw (x); }
  { catch (int i)
    { cout << "Exception caught denominator is zero"; }
    cout << endl;
    return 0;
  }
```

02/05/2022

```
class A
{ public:
  void show (int );
};
void A : show (int x)
{ int a[5] = {1,2,3,4,5};
  try
  { if (i<5)
      cout << a[i] << endl;
      else
      throw i;
  }
    catch (int b)
    { cout << " Invalid index";
    }
int main ()
{ A a;
  a. show (3);
  a. show (7);
  return (a);
}

class divide by zero
{ public: void msg1()
  { cout << "\n div by 0";
  }
};
class Array_invalid_index
{ public: void msg2()
  { cout << "\n invalid index";
  }
};
int main ()
{ int a[5], int b;
  cout << "enter b";
  cin >> b;
  for (int i=0; i<5; i++)
  {
    cin >> a[i];
  }
  try
  { if (a[0] == 0)
      throw new_divide_by_zero;
      b = 5/a[0];
    if (b>A)
      throw new Array_invalid_index;
      cout << "value of index = " << b << " is " << a[b];
  }
  Catch (divid_by_zero *e1)
  { e -> msg1 ();
  }
  Catch (Array_invalid_index *e2)
  { e2 -> msg2();
  }
}
```

⇒ it allows the programmer to define generic classes & generic functions. A template is a pattern that the compiler uses to generate a family of classes or functions. Templates are expanded at compile time like macros. The difference is compiler does type checking before template expansion. The source code contain only function/class but compiler may contain multiple copies of same function/class.

Template can be of two types in c++
① function Template 2) class template

⇒ ① Function Template :- It defines the family of functions.

(typename) return type

```
template <class T> T max (T a, T b)
{
    return (a>b)? a:b;
}
void main ()
{
cout <<"max (10,15) =" << max (10,15);
cout << "max ('k', 's') =" << max ('k', 's');
cout << "max (10.1, 15.2) =" << max (10.1, 15.2);
```

NOTE :- ① T is a dependent datatype
② use max <int, int (10,15) in other compiler

⇒ Only at compile time : family of function
⇒ T max has specific definition when it is called
i.e int max (int a, int b)
{
return (a>b)? a:b;
}
similarly with char max (char a, char b)
similarly with double max (double a, double b)

⇒ Example 2 :- More than one dependent variable

```
template <class T1, class T2> void show (T1 a, T2 b)
{
  cout << " a =" << a << endl;
  cout << "b =" << b << endl;
}
int main ()
{
  show (2020, "pandemic");   // a = 2020, b = Pandemic
  show (12.84, 1700);        // a = 12.84, b = 1700
}
```

↳ void show (int a, char b)
void show (double a, int b)

| function overloading | Template |
|---|---|
| 1. Perform similar work | 1. Perform exactly same thing |

09/05/2022

## ② Class Template :-

⇒ Example 1 :-

```
template <class T> class myclass
{ T n1, n2;
  public:
    myclass (T n1, T n2)
  { this.n1 = n1 ; this.n2 = n2; }
  void show result ()
  { cout << " values are = " << n1 << endl << n2;
    cout << " Addition is = " << n1 + n2 << endl;
    cout << " Subtraction is = " << n1 - n2 << endl;
    cout << " Multiplication is = " << n1 * n2 << endl;
    cout << " Div is = " << n1 / n2 << endl;
  }
};
void main ()
{
  myclass <int> ob1 (10, 5);
  myclass <float> ob2 (10.5f, 5.5f);
  cout << " performing computation on ob1 " << endl;
  ob1. show result ();
  cout << " performing computation on ob2 " << endl;
  ob2. show result ();
}
```

⇒ Example 2 :-

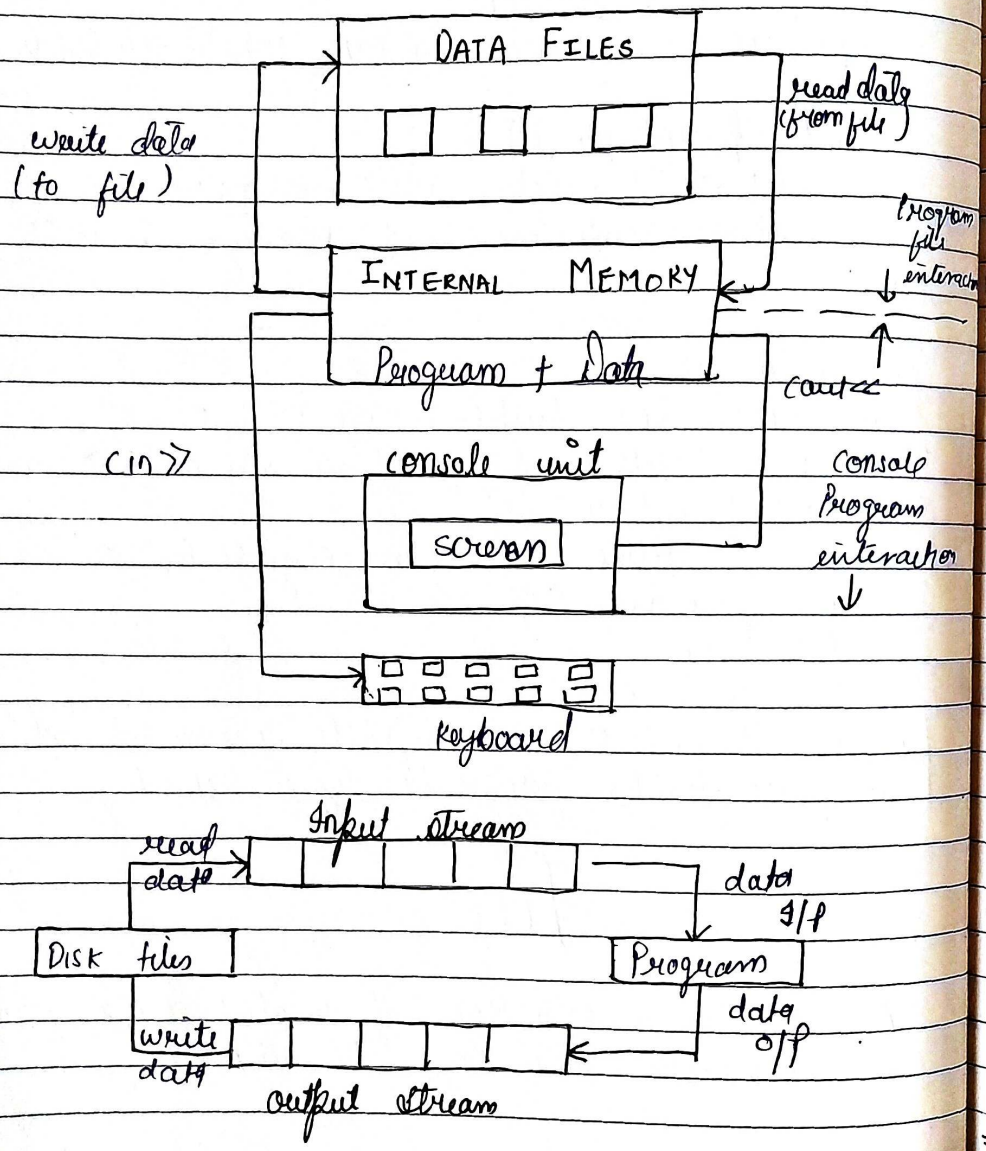
Template Parameter

```
template <class T, class U> class myclass
{ T var1; U var2;
  public: myclass (T v1, U v2) : var1(v1), var2
                                 (v2) { }
  void print var ()
  { cout << " Variable 1 = " << var1 << endl;
    cout << " variable 2 = " << var2 << endl;
  }
};
int main ()
{ myclass <int, double> ob1 ( 7, 13.6);
  cout << " value of first object is " << endl;
  ob1. print var ();
  cout << " value of second object is " << endl;
  ob2. print var ();
  return 0;
}
```

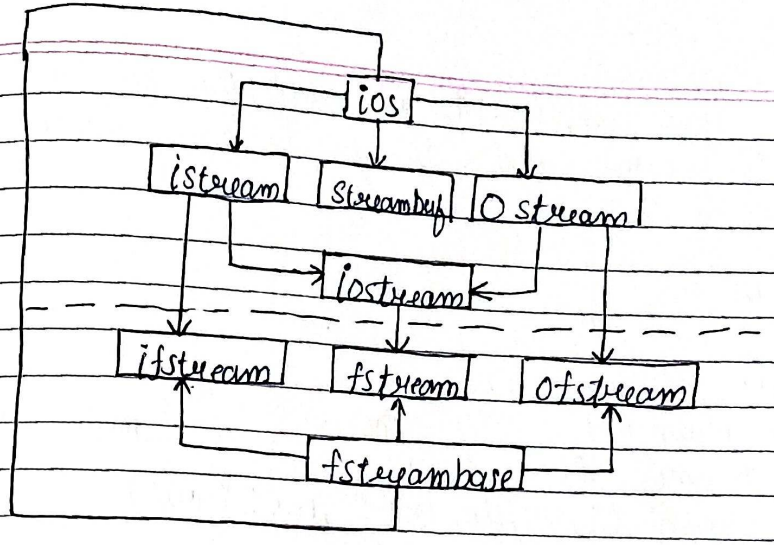
NOTE = T and U are template argument which is a place holder for data type used.

# File

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




NOTE:- <Fstream.h> is a header file for "file"




⇒ fstream base = It provides operation common to file stream, It serves as base for ifstream, fstream & ofstream class contains open() and close() function

⇒ ifstream = Provides input operation. contains open() function with default input mode. It inherits the function get(), getline(), read(), seekg(), tellg() function.

⇒ Ofstream = Provides output operation. contain open() function with default mode as output. It inherits put(), seekp(), tellp() and write() function from ostream class

⇒ fstream = Provide supports of simultaneous input & output operation contains open() function with

default input mode. Inherits all the function from istream and ostream classes through iostream

⇒ Example :-

```
# include <fstream.h>
  void main()
  { fstream fs;                          (file name, mode (out/in))
    fs.open("myfile.txt", ios::out);
    cout<<"New file created;
    fs<<"learning files in C++";
    fs.close();
    fs.open("myfile.txt", ios::in);

      const int N = 80;
      char line[N];
    while(!fs.eof())
      {
        fs.getline(line, N);
        cout<<line;
      }
      fs.close();
  }
```

out = file mai jaa kar likhna, Program --data--> file
in = content of file will come into the program
file data --> program

```
# including object in file
# include <fstream.h>
  class student
  { char name[30];
    int age;
    public: void get_data()
    {
      cout<<"Enter name :";
      cin.getline(name, 30);
      cout<<"Enter age :";
      cin>>age;
    void show_data()
    {
      cout<<"Name = "<<name<<"Age "<<age<<endl;
    }
  };
  void main()
  { student s;
    fstream f;
    f.open("student.txt", ios::out);
    cout<<"file created"<<endl;
    s.get_data();
    f.write((char *)&s, sizeof(s))
    f.close();
  cout<<"file closed successfully";

  fstream f2;
  f2.open("student.txt", ios::in)
```

```
    f2.read((char*)&s, sizeof(s));
    s.showdata();
    f2.close();
}
```

⇒ ios::app = Append to end of file
⇒ ios::ate = Goto end of file on opening
⇒ ios::in = open file for reading only
⇒ ios::out = open file for write only
⇒ ios::trunc = delete the content of file if it exist

# Constant :-

⇒ Const :- USES

1) Const is used for creating symbolic constants in C++
   for ex :- const int N=10;
   N is a symbolic constant whose value can't be modified in the program

2) The qualifier const tell the compiler that the function should not modify the argument
   ex :-

```
int manipulate(const int *p)
{
    ---
    ---
}
```

the compiler will generate an error when this condition is violated. This type of declaration is significant only when we pass argument by reference or pointer

3) Constant Member function :- If a member function doesnot alter any data in the class then we may declare it as 'Const Member function'. The Qualifier const is appended in both function definition as well as function declaration. The compiler will generate an error message if such function try to alter the data value

ex :-

```
class A
{ int x;
  Public:
    void add() const;
};
void A::add() const
{ ---
  ---
}
```

4) Const object :- We can create and use constant object using 'Const' keyword before object declaration. If any object is declared

class A as constant it means that the values of its data member should not be modify & any attempt to modify will generate compile time error.

The constant object can call only const member function. Whenever a const object try to invoke Non const member function the compiler generates error.

ex:-

```
class A
{int x;
 A()
  { x=7;}
manipulate ()
{ x++;}
add ()
{ x=x+10;}
show () const
{ cout << x;}
void main
{
const A a;

a.manipulate (); // wrong statement
a.add () // wrong statement
a.show () // correct statement
}
```

NOTE:-

any non const object can call non const function as well as const function

# SCOPE RESOLUTION

```
# include <---->
  int m;
  void main ()
{
 int m=10;
 cout << m;
 cout << :: m;
}
```

It also allows acess to global version of variable

NOTE:- New and delete are two memory management operators, malloc and calloc belongs to c

In malloc and calloc uses size of operator to determine the size but while using new we didn't have to use size of.

# Precision = we can specify the no. of digets to be displayed. After the decimal point while printing the floating point number

ex:- cout.precision(3);
cout << 3.1415;
output 3.141

# fill = we use fill function to fill the unused position by any desired character

ex.- cout.fill('?');
cout.width(5);
cout << 362
output - ??362

# Get line = Syntax :- cin.getline(line, size)

takes the value that we entered in variable

It will take input until :-
1) The user press enter key
2) if the size reach size - 1

The getline function reads a whole line of text that ends with a new line character. The function can be invoked by using the object "cin", the reading is terminated as soon as either the new line character is encountered or size - 1 characters are reached

# STL & Container

⇒ STL = In order to help the c++ users in generic programming a set of general purpose Templatised classes and functions that could be used as a standard approach for storing and processing of data is developed by c++. The collection of these generic classes and function is called standard Template library.
STL is large and complex too.

→ Components of STL

1. Container = Container is an object that actually stores data it is a way data is organised in memory. The STL containers are implemented by Template classes and therefore can be easily customised to hold diff. types of data
ex:- vector, list, stack, queue, map etc.

2. Algorithms = An algorithm is a procedure that is used to process the data contained

in the container. The STL includes many diff kinds of algorithm to provide support for task such as initializing, searching, sorting, merging and copying

3. Iterators :- An iterator is an object (like a pointer) that points to an element in a container. We can use iterator to move through the contents of the container.