Syntax

```
class xyz
{  int a;
 Public:
    void in ()
    {  cin>>a;
    {
     void out ()
     {  cout<<a;
      }
};
int main ()
{   xyz ob;     // Creating object of class
                   xyz as ob;
    ob.in();    // Calling func" or member
    ob.out();      func" of a class using
    return 0;      object
}
```

for Area

a

$A = a^2$

```
Class xyz
{  int z;
   int a;
   int b;
Public:
   void in()
   {  cin>>a;
      cin>>b;
   {
   void Area()
   {
      z = a*b;
      cout<<z;
   {
}
```

```
int main()
{
   xyz ob;
   ob.in();
   ob.Area();
   return 0;
}
```

```
Class xyz
{  int b,h,Ar;
public:
   void in()
   {
      cin>>b;
      cin>>h;
   }
   Void area()
   {
      Ar = 0.5xbxh;
   }
   void out()
   {
      cout<<Ar;
   {
}
```

```
int main()
{
   xyz ob;
   ob.in();
   ob.area();
   ob.out();
   return 0;
{
```

```
Class X
{
    Access specifier :  ]
    Variable
    Func^n ( )
};
int main( )
{   X ob;
    Ob. fun ( )
    return 0;
```

There is a default access specifier

] → Public, Private, protected

Procedural and OOP

```
void main( ){
    int a, b, sum;
    Statement 1
    Statement 2
      :
      :
    sum = a + b;
    printf ("%d", sum);
}
```

Class → Virtual Entity

Object → Physical Entity

Encapsulation : (Writing / Defining) Wrapping of variable & function in a single unit.

Abstraction
- Data & Func^n
- Data Abstraction

} Showing only useful info to user. and hiding implementation Details.

#Define Class :

The collection of objects having similar properties, common behaviours and shared relationship.

Class is a user-defined datatype.
Using class we can create object.

Syntax of Class

```
class Class_Name
{
    Access Specifier :
        variable ;
        function ( ) ;
};   // End of class
```

Access Specifier

Public : The data member & member func^n declared under public access specifier can be accessed outside the class using object of same class.

private: The data member & member func^n declared under private section cannot be accessed outside the class.

Protected: Same as private access specifier in a single class. But in case of inheritence, the protected access specifier must be used instead of private access specifier.

Data → Data member

2) Object: (creation of object happens during the execution of prog.)
It is a runtime entity in OOPs
It is a variable of class datatype
It is physical entity in OOP.
It is a user define datatype.

Syntax:

```
class_Name object;
ex: xyz ob;
```

3) Encapsulation
Wrapping of data & func^n into a single unit.

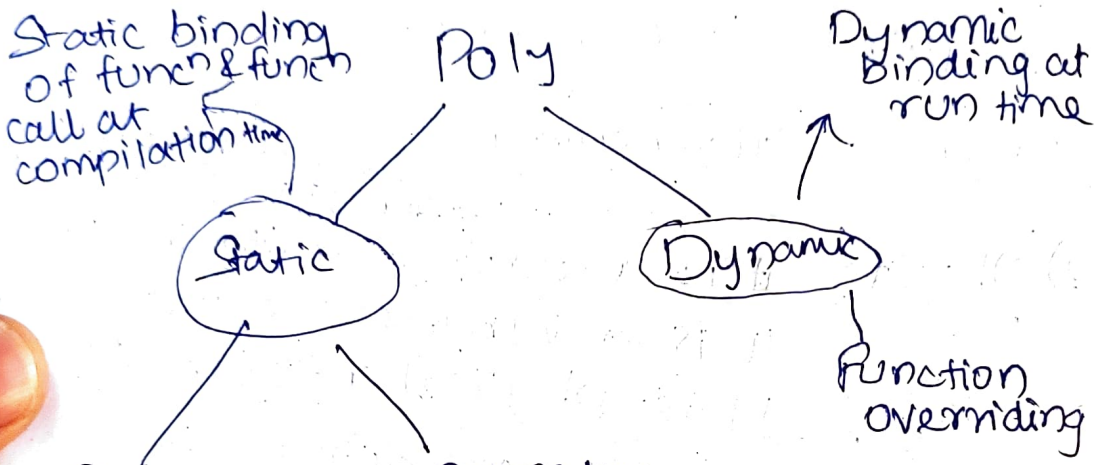
4) Abstraction
Process of representing essential features without providing background details.

```
class ABC
{   int balance;
    public:
        ==
        ==
};
```

Polymorphism

Many form

When a func^n or operator act differently at different instance of time. The process is known as polymorphism.



* Data hiding

The process of hinding the member of class to be accessed by external user.

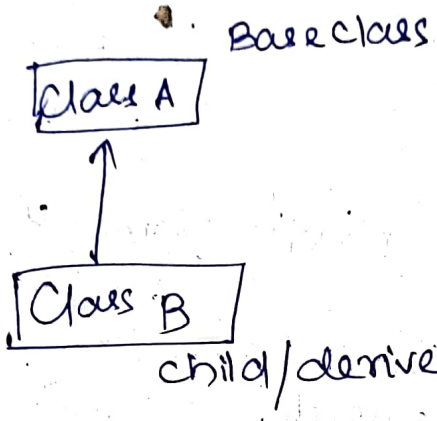
This can be achieved using private access specifier.

Inheritence

inheriting/copying/deriving

The process of aquiring the features of base class into a derived class is known as inheritence.

Also known as 'Kind of Relationship'.



Based on no. of child

```
Class A
{

};
Class B: public A
{

};
```

Deligation:

```
Class A
{ int a;

};
```

```
Class B
{ A obj;

};
```

When object of one class used as data member of another class.

It is also called as "has a relationship".

Message Passing: In OOPs, one object can communicate with other object using message passing.

```
obj.text(obj1);
obj.text(obj1.getInfo());
```

Datatype: type of value
- Range: 0-255 byte
- type of operⁿ it support

Functions

Syntax:

```
Return Type function_Name (with/without arguments)
{
    ——————
    statements...
}
```

Recursion

Entry point

```
void test()
{
    ——————
    test();
}
```

Exit Point

Constructor

It is a member function used to initialise the member variable of a class.

- Name of func^n is same as class Name
- It don't contain any return type.
- It is automatically invoked when object of class is created

Ex:

```
class ABC{
    int x;
    int y;
public:
    ABC()
    { x=10, y=20;
      cout<<x<<y;
    }
};
int main(){
    ABC ob;
    Ob.ABC();

    return 0;
}
```

(search for constructor)

Types of constructor

① Dummy constructor

Job is to provided by compiler to initialise value to zero.

② Default Constructor

In default constructor there is no argument (No formal argument) is required.

Ex:

```
Public:
  ABC()
  { x=10, y=20;
    cout<<x<<y;
  }
```

③ Parameterized Constructor

In this constructor a parameter of an argument required to be passed in the constructor.

Ex:

```
class ABC {
  int x;
  int y;
public
  ABC (int a, int b)
  { x=a, y=b;
    cout<<x<<y;
  }
};
int main(){
  int p=1, q=9;
  ABC ob(5,6);
  ABC ob(p,q);
  return 0;
}
```

④ Copy Constructor

It is used to copying an existing object to a new object.

> It is a variation of parametrized constructor.

```
Class ABC
{  int x, y;
   public
      ABC (int a, b)
      {   x=a; y=b;
          cout<<x<<y;
      }
      ABC (ABC &obj)
      {   x=obj.x;
          y=obj.y;
      }
};
int main ( )
{
 ABC ob (5, 6)
 ABC ob1 (ob);
 // calling constructor
}
```

Swarup ob3;
ob2 = ob3;

shallow copy

swarup ob3;
Sbcarup ob4 = ob3;

Deep copy

(*) Destructor

```
class ABC
{
    int x,y;
    public:
        ABC() {cin>>x>>y;}      // Default constructor
       ~ABC() {cout<<x<<y;}

};
int main()
{
    ABC ob;
    return 0;
}
```

> Destructor is used to deallocate the value and memory initialized by the constructor.

> It looks like constructor with a negation symbol prior to its definition. (It don't contain any argument, for the reason destructor cannot be overloaded)

> Yousing destructor we can also deallocate the memory of "New" using "Delete"

> Destructor automatically invoke called when the execution of the program ends, scope of the code ends and whenever delete program is executed.

```
class ABC
{ int x;
  public:
    ABC()
    { cout<< "Inside cons";}
  ~ ABC()
    {cout<< "in destructor";} };
```

```
int main()
{ ABC ob;
  ABC ob1;
  return 0;
```

O/P:
Inside cons
Inside cons
in destructor - ob1
in destructor - ob

Inline function

Syntax:

```
inline void text()
{
}
```

```
class A
{
  public:
    inline void text()
    {
      cout << "Hii";
    }
};
```

```
int main()
{
  A obj;
  obj.text();
```

Inline is a keyword which is used to make a member function inline. In the Inline, the func^n def^n of the member function will replace the function call

in the main function. This helps to reduce the execution time by reducing the pointer junk between function call and function definition.

```
#define Add(x) x+x

class A
{
  public:
  inline void Add1(int a)
        {
          cout << a+a;
        }
};
int main()
{
  A obj;
  obj.Add1(5)<<endl; cout<<a+a;
  cout << Add(5); 5+5
  return 0;
}
```

++x ++x

o/p

10
10

(*) Rules for Inline function

1) Inline function must be used as and when it is required
2) Inline function should not contain large no. of statements.
3) Inline func^n must not be recursive func^n.
4) It should not contain any kind of loop.

Static Variable

```
class X
{   static int a;
    public:
        void test()
        {   cin>>a;
            cout<<"Static"<<a;
        }
};
int x::a=0;          <- Initialisation (outside the class before the main)
int main()
{   x ob;
    ob.test();
    return 0;
}
```

- Static variable in C++ is used to create a global variable.
- Static variable can be accessed by all the objects.

> It's memory allocation is same as the member function. Where a different memory is allocated.

Count the number of objects created in the program using static variable

```
class X
{ static int a;
  public:
    X( )
  {
     a=a+1;
     cout<< "object" <<a<< "created";
  }

};
int x::a=0;
int main( )
{   X ob;
    X ob1;
    return 0;
}
```

O/P:

```
Object 1 created
Object 2 created
```

Static function

> Static functions are used to access static variable

> Static function can be called using ~~object~~ class name.

```
class xyz
{ static int a;
  public:
  static void show()
  { cout <<a;

  }
};
int xyz::a=0;
int main()
{ xyz::show();
  return 0;
}
```

Static func^n can only access to static variable

BUT

Normal func^n can access both static & normal variable

Note

Static Object

To initialize all the data member to zero.

```
static class_name object;
```

⊙ Creating Function Outside class

```
class xyz
{ public:
    void show();  //Declaration of func show()
};
```

```
void xyz :: show()  // Def^n of func show()
{
    cout << "I am outside the class";
}
```

#screenshot

Scope used operator & use

```
class xyz;  // Forward Declaration of class
int main()
{

}
class xyz
{

};
```

Friend function

- It is non-member function which can be made friend to a class using friend keyword.
- Friend func^n can access private data member of a class within which it is declared as friend.
- It can access the private data member using object of the class.
- It violates the rules of encapsulation

```
Class XYZ
{   int a;
    public :
      xyz ()
      {  a=5;
      }
    friend void outside(xyz &ob);  // Declaration

};
void outside(xyz &ob)
{   cout << ob.a;
 }
```

✳ Function Overloading

Definition

Whenever multiple func^n is defined with same name, different number of argument & diff. types of argument. The process is known as function overloading

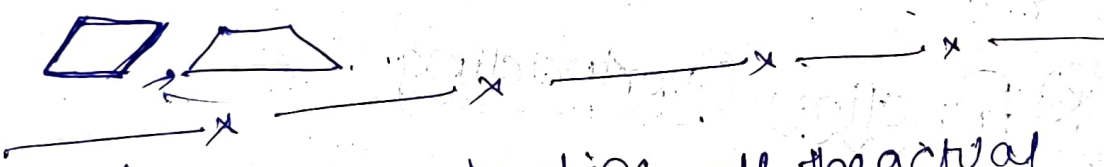
```
void Add(int a)
{   sum=a;
};
void Add (int a , int b)
{  sum=a+b;
};
void Add (int a , int b, int c)
{  sum = a+b+c;

}
```

```
int main() {
    Abc ob
    OB.Add(5);
    ob.Add(6,7);
    ob.Add(8,9,10);
    return 0;
}
```

Command line Argument

Q) WAP to overload Area function to find the area of △, □, □, ○, trapeza



> In function overloading, all the actual argument must be passed using a variable the variable must match the datatype of the formal argument.

> A constructor can also be overloaded as similar to member functions by defining multiple arguments and different data types.

> Destructor cannot be overloaded because it donot contain any argument.

(*) Operator Overloading

Definition:
In operator overloading, a single operator act differently at different instance of time

Syntax

Return Type operator operator symbol (with/without argument)

{

//Body of operator func^n

}

- List down the operators which cannot be overloaded using operator overloading.
  - :: Scope resolution operator
  - . Dot operator
  - ?: Conditional operator
  - sizeof()

- List down the operators which cannot be overloaded using operator overloading & friend function.

⊛ Types of operator overloading

1) Unary :→ Single operator and single operand
   ++a, a--;

2) Binary :→ Single operator with 2 operand.
   Ex c = a+b

⊙ Unary

(++) prefix

```
++ ob;
Void operator ++ (A &ob) {
        ++x;
        ++y;
}
int main()
{
    ABC ob;
    ++ ob.
    OB.display;
}
```

++(postfix)

```
void operator ++(int){
        x++;
        y++;
 }
int main()
{  ABC ob;

   ++ob;
  ob.display;
}
```

Assig
WAP to overload '--' prefix & postfix operator using operator overloading

Binary Operator Overloading

ob1+ob2

```
class x{
   int a,b;
   public:
      x(){
         cout << "Enter 2 Nos: ";
         cin >>a>>b;
      }
      void display(){
         cout<<"The 2 Nos are:" <<a <<b;
      }
      operator +(x &ob){
         x temp;
         temp.a = a+ob.a;
         temp.b = b+ob.b;
         return temp.};
```

```
int main(){
    x ob1, ob2;
    ob1.display();
    ob2 display();
    cout << ob1+ob2;
     ob3= ob1 +ob2;
     ob3.display();

}
```

```
Class x {
    int a,b;
    public:
        x() { cout<< "Enter 2 nos:";
              cin>>a>>b; }

        void display() {
            cout << "The 2 Nos. are:";
            cout << a << b;
        }
        x operator + (int p) {
            x temp;
            temp.a = a+p;
            temp.b = b+p;
            return temp;
        }
};
```

```
3+ob1                    (int p, y &ob)
y operator+(y &ob)
{ y temp;
  temp.a = p + ob.a;
  temp.b = p + ob.b;
  return temp;
}

int main(){
    y ob1;ob2;
    cout << 3+ob1;
    ob2 = 3+ob1;
}
```

```
class y{
  int a,b;
public:
  y(){a=1; b=2;}
  void show(){
    cout<<a<<b;
  }
friend y operator+(
        int, y&);
};
```

Questions asked

ob3 = ob1 + ob2 - 3

ob3 = ob1 * ob2 / ob3

ob3 = 3 - ob1 / ob2

⋮

⊛ Type Conversion

```
int a=9;
float b=(float)a;
```

Basic to Class →

Ex
```
xyz ob; int a;
ob= a;
```
class ← basic

> Class to basic

a = ob;

> Class to Class

ob1 = ob2 → ABC

XYZ

explicit call to constructor

① Basic to Class → Ex: XYZ ob; int a;

ob = a; ② → ob = ABC(a); (using constructor)*

① → void operator = (int a){

{

```
Class ABC
{   int X;
    public:
        //helper func
        ABC (int P){
            x = P;
        }
};
```

② WAP to convert the given time in minutes into an hour

```
float X;
public:
    ABC (int P){
        X = float(P) / 60;
```

3.0

3.3

}

H.W.

To understand how to convert class to basic.

Assignment - 2

① Class to Class Conversion.

② Convert from hour class to minute class

```
class H{
    int hour;
    public:
     H(){
        cout << "Enter value in hour:";
        cin >> hour;
     }
     void show(){
        cout << "Hour is:" << hour;
     }
};
class M{
    int min;
    public:
        void show(){
            cout << "Minute is:" << min;
        }
};
int main(){
    H ob1;
    M ob2;
    ob2 = ob1;   // or  ob2 = M(ob1);
    ob1.show();
    ob2.show
}
M(H ob){
    min = ob.hour * 60;
}
```

using operator in M Class

Add

```
operator M()
{
    return (hour*60);
}
```

ob2 = M(ob1);

(Theory → 7 marks)

(new & delete for array)

complex no.
↳ operator overlo

Module-4

Inheritance

Defn

Inheritance is the process of creating a new class from an existing class (or Old class) in order to acquire (get) property from an existing class

Syntax

```
class A {
    ===
};
class B : public A
{
    ===
};
int main() {
    B obj;
    ===
}
```

(public ⇈ Access Specifier, A ⇈ Old class)

In the above example class B is created with the help of class A

Class B ⇒ Child class / Derived class
class A ⇒ Parent class / Base class

> Code reusability.
> Extension of the code

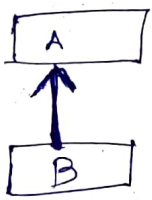
⊛ Types of Inheritance Derivation

*→ Next
✓ → Lab

Derivation ↓ / Data member →	Private	Public	Protected
Private	X	Private	Private
Public	X	Public	Protected
Protected	X	Protected	Protected

⊛ Types of Inheritance

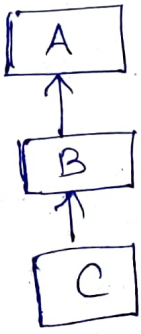
1) Single Inheritance




```
class A {
//code
};
class B : public A {
};
```

In single inheritance there exist a single base class and a single derived class.

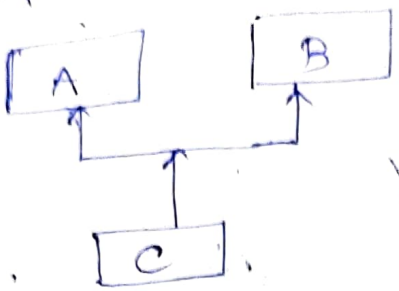
2) Multilevel Inheritance




```
class A
{ ---- };
class B : public A {
};
class C : public B
{
};
```

In multilevel inheritance a class is derived from a base class which is again derived from another base class.

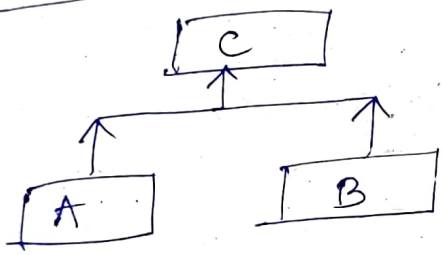
3) Multiple Inheritance



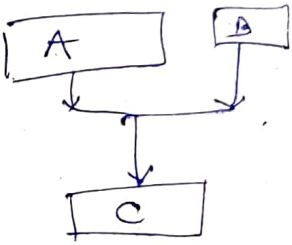

```
class A
{
};
class B {
};
class C: public A, public B
{
};
```

In multiple inheritance a class is derived from multiple base class.

4) Hierarichal Inheritance (Tree)



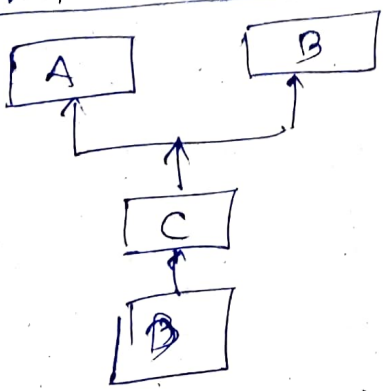
or




```
class C{
};
class A: public C{
};
class B: public C{
};
```

When multiple classes are derived from a single base class, it is known as hierarichal class.

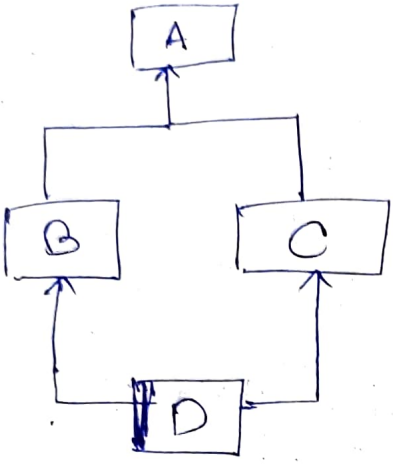
5) Hybrid Inheritance

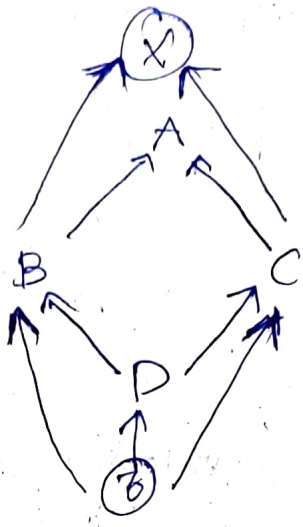



```
class A {
};
class B
{
};
class C: public A, public B
{
};
class D: public C{
};
```

The hybrid inheritance is a comb^n of more than one type of inheritance

6) Multipath / Diamond





Multipath inheritance is a special type of hybrid inheritance in which more than one path exist from the top base class to the bottom derived class.

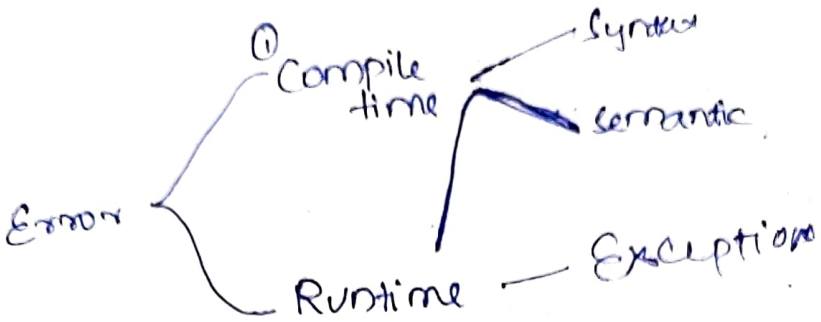
```
Class A {
    public:
        int x;
};
Class B: virtual public A {

};
class C: virtual public A {

};
Class D: public B, public C {

};
```

Exception Handling



Definition:

→ Exceptions are the runtime error which leads to abnormal termination of the program.

→ This runtime error can be handle at the runtime and the process is known as exception handling.

The process of handling exception (Run time error) is known as exception handling.

try: → Within code which may leads to exception.

throw: → Inside try at the point where exception can be generated. And pass the exception to catch block for exception handling into catch block.

catch: → Recieve exception from throw statement and handle the exception.

```
int main(){
  try{ int a;
       cin>>a;
       if (a==0)
          throw 0;
       else
          cout<<"a:"<<a;
  }catch(int x){
       cout<<"Value of input is:"<<x;
  }
  cout<<"After exception handling";
} // End of program
```

```
Class X{
    int a;
    public:
        X(){ cin>>a; }
        void disp(){
            if(a==0){
                throw 0;
            }else{
                cout<<"a:"<<a;
            }
        }
};
int main(){
    try{
        X ob;
        ob.disp();
    }
    catch(int x){
        cout<<"Error:"<<x;
    }
}
#include<stdexcept.h>
```

⇒ display

⇒ Try

⇒ Catch

```
try {
    if (i==0){
        throw 0;

    if (i>0){
        throw 2.5;

    if (i<0){
        throw 'A';
    }

}
catch (int a)
{
   cout<<"zero";
}
catch(char c){
    cout << "less than zero";
}
catch(float b){
    cout<<"More than zero";
}
```

```
try {
    if (i==0)
        throw 0;

    if (i>0)
        throw 1;

    if (i<0)
        throw 2;

}
catch (int a) {
    switch(a){
        case 0: cout << "zero";
                break;

        case 1: "less than zero";
                break;

        case 2: cout<<"less than 0";
                break;
    }
}
```

Catch all

```
Catch(...)
{
    cout << "Exception generated";

}
```

⊛ Template

Def^n: Template is used for generic programming.

Generic programming means a program which can support all kinds of datatype (all primitive types as mentioned by the programmer.) → (float, int, char)

To write a generic programming following syntax is required:

Syntax: template <class T>

In the above syntax, template and class are the keywords and T is referred as template class

Using template class variables are declared or used within a function or a class to make the program generic.

→ There are 2 types of template;

(i) Function template
(ii) Class template

⊛ Function Template

```
#include <iostream>
using namespace std;
template <class T>
void fun (T a, T b){                    // T is convert to int
    cout<<"Value of a & b are";
    cout<<a<<b;
}
int main(){
    int x, y;
    x=10, y=30.5;
    fun(x, y);

    return 0;
}
```

When a function uses template variable as its argument then the function is known as function template.

In the above example there is a single template class as T. Here fun() is a function template having two argument with the datatype as T. Withing the fun() T datatype variable a, b are printed using cout statement.

In the main program during the call of function template either direct value or a primitive datatype variable can be passed to execute fun().

It is better to used different class for diff data type
template <class T1, class T2>

→We can use T as return type

```
int fun(T1 a, T2 b){
    T1 c; T2 d;
}
```

⊛ Class Template

```
template <class T>
class xyz{
    T a;
    T b;
    xyz (Tx, Ty){
        a=x; b=y;
    }
    void disp(){
        cout<<a<<b;
    }
};
```

```
int main(){
    xyz <int> ob(1,2);
    ob.disp();

    return 0;
}
```

When a template class variable used as datam
of the class, the class is known as class
template.

In the above example T variables such as a,b
are used as datamember of class xyz. In the
main function during the creation of object
the primitive datatype must be specified

```
template <class T1, class T2>
xyz <int, float> ob (9, 10.5);
ob.disp();
```

Function overloading using template
operator also

```
template <class T1, int size>
class xyz {
    T1 a[size];
    T2 &b;
};
int main () {
    xyz <int, 20> ob1;
    xyz <float, 100> ob2;
}
```