What do you mean by OOP's? Write the basic features of OOP's.

C++ is a superset of C. Every valid statement of C is also valid in C++.But vice–versa is not correct. C++ is based on the concept of object oriented programming. Object oriented means resource must be accessed by the object. A programming language can be considered as OOP’s if they have the following property-

1. Data hiding 2. Encapsulation 3. Inheritance 4. Polymorphism 5. Abstraction

1.**Data Hiding**->Data hiding means to control the access of data. It means data must be accessed by the right person and right place. C++ implement the features using access specifier (Private, Public, & Protected) for the member of class.

2.Encapsulation->Encapsulation is the process of binding variables and functions together in a logical frame to make a single existence. C++ implement this function using class.

3.Inharitance->It is ability of a class to make several genetic class at lower level so that lower level class will get the features of upper class automatically. The major advantage of inheritance is code reusability. C++ implement this concept using base class and derived class.

4.Polymorphism->Polymorphism means one thing having many forms. It means it is the ability to take multiple forms for different operation. C++ implement this concept Using Operator Overloading function, overloading and virtual function.

5.Abstraction->Abstraction means to hide the complexity of system from the user. C++ implement this concept using class and object.

Points related to class and object

1.Class is an important concept of C++ which implement the features of encapsulation.

2.Class is collection of variables and functions. Variables are known as data member and functions are known as member function.

3.To declare a class we use a keyword "class". The general format is

```
class classname
{
     data member
      --------------
access specifier:
       member function
        ---------------
 };
```

4.The declaration of class is treated as a single statement hence it must be terminated by the semi colon.

5.The declaration of class is only a blueprint or template. There is no any memory allocation for the member of class.

6. Class creates an user defined data types of its own type.

7.To use the features of class we have to create an object. To create an object we use class name as a data type. The general format is

classname obj1,obj2,_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ objn;

8. As soon as object is created there is sufficient memory will be allocated for the object. The amount of memory for an object will be depend on the data member of class.

9. Every object gets a separate memory allocation according to its data member which is not overlapped by each other.

10. To access the member of class, we use the object with the help of dot operator.

11.Class is a logical construct where as object is physical entity.

12. In C++ structure is similar to class.

13. In C++ we get three access specifier for the member of class. a) Private b)Public c)Protected

14. The member of class is private by default. If the member of class is private, it can be accessed only inside the class. Private member can't be accessed outside of the class.

15. If the member of class is public, it can be accessed inside the class or outside the class.

16 .We should define the data member in the private section of class and member function in the public section of class.

Early binding Vs late binding / Static Binding Vs Dynamic Binding

When all information’s are available at compile time for association it is called early binding. Early binding is also called compile time binding or static binding or compile time polymorphism. The major advantage of early binding is its efficiency because all associations are completed at compile time. The function overloading and operator overloading are example of early binding.

When information’s are not decided at compile time and association is completed by the run time system, it is called late binding. Late binding is also known as run time binding and dynamic binding or run time polymorphism. The major advantage of late binding is its flexibility. Virtual function is known as example of late binding.

Abstract class:→A class which contains at least one pure virtual function it is called abstract class. We can't create the object of abstract class.

All derived class inherited through the abstract class must define the pure virtual function otherwise derived class also becomes an abstract class. Although we can't create the object of abstract class but we can create pointer which will hold the address of derived class object.

Write at least 5 uses of scope resolution operators.

1. C++ allows flexibility of accessing both the variables through a scope resolution operator. **2.** Uses of scope resolution operator to write function definition outside class definition. **3.** To access the over ridden member we use scope resolution operator. **4.** Scope resolution operator is used to identify and disambiguate identifiers used in different scope. **5.** Scope resolution operator is used to with namespaces and classes. **6.** Scope resolution operator is used to call static members of classes. **7.** The scope resolution operator is also used with the values of a scoped enumeration Declaration.

What is inheritance? Write different types of inheritance with example.

Inheritance->Inheritance is the important concept in C++ which provides the features of code reusability. Inheritance is the ability of a class to make a several generic class at lower level so that lower level class will get the features of upper level class automatically.

The upper level class which contains some common features of program and defined for inheritance is called base class. Lower level classes which inherits common features of base class is called derived class. Derived class can access any member of base class (Except private member). It can also defined some separate features in its own class but these self-defines function can't accessed by the base class.

```
class first
{
    int a,b;
    protected:
      int c;
public:
void get(int p, int q, int r)
{
    a=p;
    b=q;
    c=r;
}
void show()
{
cout<<a<<b<<c;
}
};
class second:public first
{
int d,e;
public:
void getdata(int p,int q)
{    d=p;
     e=q;
}
void disp()
{
      cout<<d<<e;
      cout<<c;
      cout<<a<<b;        ---------------------We can't write
}
};
void main()
{
    first F1;
    second S1;
    F1.get(10,15,20);
    F1.show();-------------------10,15,20
    S1.getdata(50,60);
    S1.get(70,80,90);
    S1.disp();-------------------50,60,90
   S1.show();------------------70,80,90
   F1.getdata(100,200);--------------Error
   F1.disp();------------------------Error
}
```

Types of inheritance & Examples→C++ provides following types of inheritance -

1.Single inheritance/Simple inheritance
2.Multilevel inheritance
3.Multiple inheritance
4.Hierchical inheritance
5.Hybrid inheritance

1. Single inheritance-When a single base class having single derived class is called single inheritance.

```
class first
{
    int a;
     int b;
public:
void get(int p,int q)
{
    a=p;
    b=q;
}
void show()
{
     cout<<a<<b;
}
};
class second:public first
{
     int c;
     int d;
public:
void getdata(int p,int q)
{
    c=p;
    d=q;
}
void disp()
{
    cout<<c<,d;
}
};
void main()
{
    first F1;
    second S1;
    F1.get(10,15);
    F1.show();
    S1.getdata(20,25);
    S1.disp();
    S1.get(30,35);
    S1.show();
}
```

2. Multi level inheritance-When a single base class having single derived class in different level it is called multilevel inheritance

```
class first
{
     int a,int b;
public:
void get(int p,int q)
{
    a=p;
    b=q;
}
void show()
{
    cout<<a<<b;
}
};
class second:public first
{
    int c;
    int d;
public:
void getdata(int p,intq)
{
    c=p;
    d=q;
}
void disp()
{
    cout<<c<<d;
}       };
class third:public second
{
    int x;
    int y;
public:
void input(int p,int q)
{      x=p;
       y=q;
}
void output()
{
    cout<<x<<y;
}       };
void main()
{
     first F1;
     second S1;
     third T1;
     F1.get(10,15);
     F1.show();
     S1.getdata(20,25);
     S1.get(30,35);
     S1.disp();
     S1.show();
     T1.input(40,45);
     T1.getdata(50,55);
     T1.get(60,65);
     T1.output();
     T1.disp();
     T1.show();
}
```

3. Multiple inheritance->When a single derived class inherited with multiple base class, it is called multiple inheritance.

```
class first
{
   int a, b;
public:
void get(int p,int q)
{
    a=p;
    b=q;
}
void show()
{
     cout<<a<<b;
}            };
class second
{
    int c, d;
public:
void getdata(int p, int q)
{
    c=p;
    d=q;
}
void disp()
{
     cout<<c<<d;
}
 };
class third:public first,public second
{
      int x, y;
public:
void input(int p,int q)
{
    x=p;
   y=q;
}
void output()
{
    cout<<x<<y;
}
};
void main()
{
    first F1;
    second S1;
    third T1;
    F1.get(10,15);
    F1.show();
    S1.getdata(20,25);
    S1.get(30,35);                   ---------------Error
    S1.disp();
    S1.show();                       ----------------Error
    T1.input(40,45);
    T1.getdata(50,55);
    T1.get(60,65);
    T1.output();
    T1.disp();
     T1.show();
}
```

4. Hierarchal inheritance->When a single base class having multiple derived class in a hierarchical manner it is called hierarchal inheritance.

```
class first
{
    int a, b;
public:
void get(int p,int q)
{
      a=p;
     b=q;
}
void show()
{
     cout<<a<<b;
}                };
class second:public first
{
     int c, d;
public:
void getdata(int p,int q)
{
    c=p;
   d=q;
}
void disp()
{
     cout<<c<<d;
}
};
class third:public first
{
   int x y;
public:
void input(int p,int q)
{
    x=p;
    y=q;
}
void output()
{
     cout<<x<<y;
}          };
void main()
{
     first F1;
     second S1;
      third T1;
      F1.get(10,15);
      F1.show();
      S1.getdata(20,25);
      S1.get(30,35);
      S1.disp();
      S1.show();
      T1.input(40,45);
      T1.getdata(50,55);                ----------------Error
      T1.get(60,65);
      T1.output();
      T1.disp();                        ----------------Error
       T1.show();
}
```

5. Hybrid inheritance-The combination of above two more inheritances is called hybrid inheritance.

```
class first
{
     int a,b;
public:
void get(int p,int q)
{
     a=p;
     b=q;
}
void show()
{
      cout<<a<<b;
}
};
class second:public first
{
        int c,d;
public:
{
void get(int p,int q)
{
     c=p;
     d=q;
}
void disp()
{
      cout<<c<<d;
}
};
class third:  public first
{
     int x, y;
public:
void input(int p, int q)
{
      x=p;
      y=q;
}
void output()
{
       cout<<x<<y;
 }
};
class fourth: public third
{
      ----
};
void main()
{
      fourth F1;
     F1.get(10,15);
     F1.getdata(20,25);             -----------------Error
     F1.input(30,35);
     F1.show();
     F1.disp();.
     F1.output();                  --------------------Error
     F1.show();
}
```

Characteristics of constructor:

1) Constructor is also a member function of the class which has the same name as class name.
2)Constructer is executed automatically when the object is created. Actually constructer allocates the memory for the object.
3)If there is no any constructer defined in the class, System will provide a constructor called system default constructer or dummy constructer.
4)constructer has no any return type even we can't write void.
5)Although a constructer has no any return type but it can accept parameter. It means we can overload the constructer. It means we can define more than one constructer in a single class.
6)If any constructer defined in the class, system default constructer will not work.
7)constructer should define in the public section of class.
8)Basically constructer is useful to initialize the object.

```
class first
{
        int a b;
public:
void get(int p, int q)
{
        a=p;
       b=q;
}
void show()
{
         cout<<s<<b;
}
first()
{
        cout<<"In constructer";
        a=0;
        b=0;
 }              };
void main()
{
       first F1,F2;
       F1.show();
       F2.show();
       F1.get(10,15);
       F2.get(20,25);
       F1.show();
       F2.show();
 }
```

Characteristics of destructor

1. Destructor is also a member function of the class which has the same name as class name but it must follow by tilled symbol (~).
2. Like constructer, Destructor is executed automatically but it is executed when the object is not needed in the program. Actually constructer allocates the memory for the object where as Destructor de-allocates memory of objects.
3. Like constructer Destructor has no any return type but it also not accepts parameter. It means we can't overload Destructor. It means we can't define more than Destructor in a single class.
4. If there is no any Destructor defined in the class object will release the memory after the termination of program.
5. If the program is terminated forcelly, Destructor will be called for every alive object.
6.Distructer should be also define in the public section of class.

```
class first
{
    int a, b;
public:
void get(int p,int q)
{
       a=p;
       b=q;
}
void show()
{               cout<<a<<b;                  }
first()
{
       cout<<"In constructer";
       a=0;
       b=0;
}
~first()
{
          cout<<"Bye";
}
};
void main()
{
    first F1,F2;
    F1.get(10,15);
    F2.get(20,25);
    F1.show();                       --------------10,15
     F2.show();                      ---------------20,25
}
```

This pointer:→ It is special pointer which will hold the reference of calling object.

```
class first
{
        int a;
        int b;
public:
void get(int p,int q)
{
          a=p;
          b=q;
}
void show()
{
        cout<<this;
        cout<<this->a;
        cout<<this->b;
        cout<<a<<b;
}
};
void main()
{
      first F1,F2;
      F1.get(10,15);
      F2.get(20,25);
      F1.show();
      F2.show();
}
```

Dynamic memory allocation-->When we create an array, we must give the size of array at compile time it means we can say that user has no any roll to decide the size of array. This generates two types of problem-

1) insufficient memory

2)Wastage of memory

To remove the above problem, we can use run time memory management. C++ provides two special operators for run time memory management-

1)new 2) delete

new-->This operator allocates memory at run time. The general format is

```
  ptr=new datatype;
```

e.g.

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

delete-->This operator deallocates the memory at run time

```
int *p;
p=new int;
delete p;
```

e.g.

Write a program to input n numbers in the array and print the sum.

```
void main()
{
     int n,i,sum=0;
     cout<<"Enter array sie";
     cin>>n;
     int *A;
     A=new int[n];
    for(i=0;i<n;i++)
    {
         cout<"Enter a number";
        cin>>A[i];
    }
      cout<<"Elements of array";
      for(i=0;i<n;i++)
     {
        cout<<A[i];
        sum=sum+A[i];
     }
   cout<<sum;
   delete A;
}
```

Overridden member-->if a derived class member has same name as the base class member, member is known as over ridden member. We can override data member as well as member function. In this case if the object of derived class wants to access the over ridden member of base class, it has to take help the base class name and scope resolution operator.

```
class first
{
    int a;
protected:
   int b;
public:
void get(in p, int q)
{
     a=p;
     b=q;
}
void show()
{
    cout<<a<<b;
}
};
class second:public first
{
int b;
int c;
public:
void getdata(int p, int q)
{
   b=p;    c=q;
}
void show()
{
   cout<<b<<c;
   cout<<first::b;
}
};
void main()
{
     first F1;
     second S1;
     F1.get(1015);
     F1.show();                          ----------10,15
     S1.getdata(20,25);
     S1.get(30,35);
     S1.show();                          ---------20,25,35
     S1.first::show();                   ---------30,35
}
```

What is friend function? How friend function can be used in operator overloading.

Characteristics of friend function:

(1) Friend function is non-member function of the class. Although it is non-member function but it can access the private member of class. (**2**) Although it can access the private member of class but it has to take help the object of that class. (**3**) Although it is non-member function but it has to declare inside the class with the help of friend keyword. Although it can be declared in section of class (private, protected and public). (**4**) Friend function should be defined outside of the class. To define a friend function, we can't use scope resolution operator because scope resolution operator is used with only the member function. (**5**) Friend function can't be called by the object because object can call the only the member function. (**6**) A function can be also friend of more than one classes.

```
class first
{
     int a;
     int b;
public:
void get(int p,int q)
{
    a=p;
    b=q;
}
void show()
{
    cout<<a<<b;
}
friend int sum(first);
};
int sum(first F1)
{
    int S;
    S=F1.a+F1.b;
    return S;
}
void main()
{
     first F1,F2;
     F1.get(10,15);
     F2.get(20,25);
     F1.show();                    -----------------10,15
    F2.show();                     ------------------20,25
    cout<<sum(F1);                 ---------------25
     cout<<sum(F2);                  ---------------45
}
```

What is manipulator? How we can create our own manipulator.

Manipulators are used to define a specified format which can used with output statement. To define our own manipulator, we use following general format-

```
  ostream& manipulatorname(ostream &output)
  {
             -----------
            ----------
          return output;
  }
```

e.g.--

```
ostream& form(ostream &output)
{
      output.precision(2);
      output.width(10);
      output.fill('*');
      output.setf(ios::right);
      output.ios::showpos);
      return output;
}
void main()
{
       float a=5.2143;
       cout<<form<<a;
}
```

Write the various restrictions of operator overloading.

Various restriction of operator overloading

1. Non overloadable C++ operator

operator category	Operators
Member access	(dot operator)
scope resolution	::(global access)
conditional	?:(conditional statement)
pointer to member	*
size of data type	sizeof(..)

Similarly, any of the new casting operators:static_cast<>, dynamic_cast<>, reinterpret_cast<> and const_cast<> as well as the # and ## preprocessor tokkens may not be overload.

2. Neither the precedence nor the number of arguments of an operator may be altered. An overload && for example, must have exactly two arguments -- just like the built in && operator.

3. Invention of new operators is not allowed. For example

```
          void operator@(int);
```

Illegal, @ is not a built in operator or a type name.

4. After overloaded operator there is no any changes in basic meaning of operator.

Friend function in operator overloading→ When we overload unary operator, we have no need to pass any argument because unary operator function. Only one operand and that operand will be the reference of calling object itself. When we overload binary operator, we have to pass only one operand because binary operator needs two function of operation and one operand is reference of calling object itself. But if we use the concept of friend function in operator overloading, the complete story will be change. Friend function is non-member function hence there is no any concept of calling object it means we pass all the arguments externally.

Static member-->In C++ we can declare a variable as well as function as static.

1. Static data member

```
class first
{
    int a;
    int b;
public:
void get(int p,int q, int r)
{
   a=p;
   b=q;
   c=r;
}
void show()
{
          cout<<a<<b<<c;
}
};
int first::c;
void main()
{
     first F1,F2;
     F1.get(10,15,25);
     F1.show();                  ------------------10,15,25
     F2.get(50,60,70);
     F2.show();                  -------------------50,60,70
     F1.show();                 --------------------10,15,70
}
```

When we declare normal data member, separate memory will be allocated for each member but when we declare static data member, a common memory will be allocated which is share by all objects. When we declare normal data member, the memory will be allocated at the time of object creation because normal data member depends on the number of objects but in the case of static data member, memory will be allocated inside the class because it is impossible to allocate the memory for the member of class inside the class.

To remove the above problem, we have to declare the static data member outside the class. Static data member can be accessed directly without creating object.

```
class first
{
int a;
static int b;
public:
static int c;
void get(int p, int q, int r)
{          a=p;
            b=q;
            c=r;
}
int first::c=200;
void show()
{
     cout<<a<<b<<C;
 }
};
int first::b=100;
void main()
{
    cout<<first::b;                          Error(private)
    cout<<first::c;                          -------200
    first F1;
    F1.get(10,15,25);
    F1.show();                              --------10,15,25
    cout<<F1.a;                             -----Error(private)
    cout<<F1.b;                            -----Error(private)
    cout<<F1.c;                            ---------25
 }
```

2. Static member function: -->A function can be also declare as static. A static member function can access other static member. Static member function can be also access without the help of object.

```
class first
{
int a;
static int b;
public:
void get(int p, int q, int r)
{
a=p;
b=q;
c=r;
}
void show()
{          cout<<a<<b<<c;
 }
static void disp()
{
       cout<<a;        //(non static)
      cout<<b<<c;
}
};
int first::b=100;
int first::c=200;
void main()
{
          first::disp();               -----------100,200
          first F1;
          F1.get(10,15,25);
          F1.show();                   -------10,15,25
          F1.disp();                   --------15,25
}
```

Virtual Function: →C++ provides two types of polymorphism

1)Compile time polymorphism 2) Run time polymorphism

When all information are available at compile time for association it is called compile time polymorphism. Compile type polymorphism is also called compile time binding or static binding or early binding. The major advantage of early binding is its efficiency because all associations are completed at compile time. The function overloading and operator overloading are example of compile time polymorphism.

When information are not decided at compile time and association is completed by the run time system, it is called run time polymorphism. Run time polymorphism is also known as run time binding and dynamic binding or late binding. The major advantage of late binding is its flexibility. Virtual function is known as example of run time polymorphism.

```
class first
{
public:
  void show()
{
          cout<<"In first";
}          };
class second: public first
{
public:
void show()
{
       cout<<"In second";
}                };
class third:public first
{
public:
void show()
{
          cout<<"In third";
}
};
void main()
{
        first *P1,*P2;
        second S1;
        third T1;
        P1=&S1;
        P2=&T1;
        P1-->show();              -----------in second *          in first
        P2-->show();              -----------in third *           in first
}
```

Base class pointer can hold the address of derived class object. Although it will hold the address of derived class object but it always access the member defined in its own class.

Actually when we create the pointer of base class, it will automatically have associated with base class at compile time hence although it will store the address of derived class object but can't able to access derived class member.

--------------------------- Continue from ֍֍֍

--------------------------- Remaining part ֍֍֍

To remove the above problem, we have to declare base class function as virtual. in this case compiler only check the syntax and leave the association for run time system. In this case run time system perform the association according to the address of base class pointer.

```
class first
{
public:
 virtual void show()
{
     cout<<"In first";
}            };
class second:ublic first
{
public:
void show()
{
     cout<<"In second";
}               };
void main()
{
    first *P1,*P2;
    second S1;
    third T1;
    P1=&S1;
    P2=&T1;
    P1-->show();                    ------------In second
    P2-->show();                     ------------In third
}
```

Write a program to create a simple text file using constructer.

```
#include<iostream.h>
#include<fstream.h>
void main()
{
     ofstream fout("Item");
     char name[20];
     int cost;
     cout<<"Enter item,name and cost";
     cin>>item>>name>>cost;
     fout<<item<<cost;
     fout<<item<<endl;
     fout<<cost<<endl;
     fout.close();
     ifstream fin("Item");
     cout<<"------Content of item file-----";
     fin>>name;
    fin>>cost;
    cout<<name<<cost;
    fin.close();
}
```

Pure Virtual Function with example→pure virtual function has only the declaration on the base class. There is no anybody of pure virtual function in the base class. Actually pure virtual function is not a real implementation in the base class. It is over ridden by the derived class. When a base class want to restrict all derived class to override some member function, that function will be declare as pure virtual function.

```
class first
{
public:
void get(int p, int q)
{
     a=p;
     b=q;
}
virtual void show()=0;
};
class second : public first
{
    int c, d;
public:
void getdata(int p,int q)
{
     c=p;
     d=q;
}
void show()
{
      cout<<c<<d;
}
};
class third : public first
{
     int x, y;
public:
void input(int p,int q)
{
    x=p;
    y=q;
}
void output()
{
       cout<<x<<y;
}
};
void main()
{
          first F1;                              ------Error
          F1.get(10,15);                   ------error(can't write)
          F1.show();                          --------error(can't write)
          second S1;
          S1.getdata(20,25);
          S1.show();                          ------------20,25
          third T1;
          T1.input(30,35);
          T1.output();                        ----------30,35
          first*p1;
          p1=&S1;
         P1-->show();                      ----------20,25
}
```

What is type casting? Explain different types of typecasting with example.

Type Casting->Type casting is the process to change the data type of variable during the execution of expression.

There are two types of type casting-

1.Implicit type casting (Automatic type conversion)

2.Explicit type casting

(i) Implicit type casting→It is the process to change the data type of variable automatically. When we use more than one variable in a single expression, the lower types of variable will be change into the higher types automatically. This process is called implicit type cast. Following are the memory size priority of variables given order to highest to lowest priority.

Long double, Double, float, long, int, short, char

```
          void main ( )
          {
                   int a;
                   float b,c;
                   a =12;
                   b = 13;
                   c = (a+b)/2;
                   cout<<c;           25.00/2 = 12.50
            }
```

2. Explicit type casting :->It is the process to change the data types of variable forcibly.

```
                    Ex. main ( )
               {
                              Inta,b;
                              Float c;
                              a = 12;
                              b = 13;
                              c = (a+b)/2;
                              cout<<c;            12.00
                    }
```

The above program generates wrong output to solve this problem we have to need explicit type casting.

```
          void main ( )
          {
                   int a,b;
                   float c;
                   a = 12;
                   b = 13;
                   c = (float)(a+b)/2;
                   cout<<c;
}
```

Exception handling→Sometimes we don't sure that certain part of programming code is going to work right or not due to unavailability of resources, out of range etc.. These types of critical situation are known as exception. C++ provides three special keywords for exception handling-

1)try 2)throw 3)catch

try block contains suspicious code that to be tried. If exception takes places, throw the exception which is received by the catch block according to exception type. The general format is

```
try
{
   Suspicious  code
   if(condition)
     throw exception_type
 }
catch(exception_type)
{
 User Defined  Message
}
```

WAP to throw and exception "out of index" in the case of array if user break the boundary.

```
void main()
{
int A[10];
for(i=0;i<10;i++)
{
     cout<<"Enter a number";
     cin>>A[i];
}
cout<<"Elements of array";
try
{
  for(i=0;i<=10;i++)
     If(i>=10)
         throw"Out of index";
        cout<<i;
}
catch(char* str)
{
      cout<<str;
}
}
```

WAP to overload == operator to compare two strings.

```
enum  Boolean {false, true} ;
class string
{
     char str[50];
public:
void get()
{
    cout<<"Enter one string";
    cin>>str;
}
void show()
{
     cout<<str;
}
boolean operator==(string p1)
{
       return(strcmp(str,p1.str)==0)?true:false;
}
};
void main()
{
      string S1,S2;
      S1.get();
      S2.get();
   if(S1==S2)
      cout<<"Both strings are equal";
  else
       cout<<"Both strings are not equal";
}
```

Inline function→When we call a function control will be transfer from calling function to called function and after the execution of called function control will be again transfer from called to calling. If we call a function many times, program execution becomes slow due to control transfer. If we want, we can save this time if we define a function directly at the point of calling but it is impossible to define a function inside the other function. To remove the above problem, we can define frequently used function as inline. In this case compiler creates a logical copy of inline function and paste it directly at the point of calling. We must define inline function before the calling function.

```
inline void show()
{
       ---
}
void main()
{
for(int i=1;i<=10000;i++)
      show();
}
```

Note-->If we want to define a large size of function as inline, compiler may ignore the request due to memory issue.

Copy constructor→ A constructor can also accept the reference of an existing object as an argument, constructor is called copy constructor. In this case the member of existing object will be coping on the member of new object. Copy constructor is basically useful to initialize a new object with the existing object sometimes also called clone of object.

```
class first
{
    int a,b;
public:
void get(int p,int q)
{
    a=p;
    b=q;
void show()
{
     cout<<a<<b;
}
first()
{
    a=0;
    b=0;
}
first(first &p1)
{
     a=p1.a
     b=p1.b;
}
};
void main()
{
    first F1;
    F1.get(10,15);
    F1.show();                    --------------------10,15
    first F2(F1);
     F2.show();                   --------------------10,15
}
```

Parameterized constructer→If we pass a parameter as constructer argument, it is called parameterized constructer. In this case constructer will be call according to parameter passed at the time of object creation.

```
class first
{
     int a, b;
public:
void get(int p,int q)
{
   a=p;
   b=q;
}
void show()
{
   cout<<a<<b;
}
first()
{
    a=0;
    b=0;
}
first(int p)
{
   a=b=p;
}
first(int p,int q)
{
   a=p;
   b=q;
}          };
void main()
{
     first F1;
     first F2(50);
     first F3(100,200);
     F1.show();                 --------------0,0
     F2.show();                 --------------50,50
     F3.show();                 --------------100,200
     F1.get(10,15);
     F2.get(20,25);
     F3.show(30,35);
     F1.show();                 ----------------10,15
     F2.show();                 ----------------20,25
     F3.show();                -----------------30,35
}
```

What is template? How we can overload the function template. Explain with example.

When the logics are same but datatypes are different, we can use the concept of function overloading to share the same name but we must define function body separately it means we can say that function overloading does not provides actual features of reusability. If the logics are same only datatypes are different we can use the concept of template. Here we have to define a function template which can work on generic data type. The actual data type will be known at the time of function calling.

e.g.

```
template<class T>
void swap(T&a,T&b)
{
    T c;
    c=a;
    a=b;
    b=c;
}
void main()
{
     int a,b;
     float c, d;
     a=10;
     b=15;
     c=10.50;
     d=15.50;
     cout<<a<<b;
     swap(a,b);
     cout<<a<<b;
     cout<<c<<d;
     swap(c,d);
    cout<<c<<d;
}
```

Overloading Function Template→ As any normal function we can also overload function template.

e.g.

```
template<class T>
void show(T data)
{
     cout<<data;
}
void show(T data, int n)
{
for(int i=1;i<=n;i++)
      cout<<data;
}
void main()
{
  show(25);                          ----------------25
  show(25.50);                      -----------------25.50
  show("CodersHelpline");          ---------------- CodersHelpline
  show(25,3);                      -----------------25,25,25
  show("CodersHelpline ",3);       ------------CodersHelpline,CodersHelpline,CodersHelpline
}
```

Function overloading:→ If we define more than one function with same name, but different type signature, functions are known as overloaded function. Type signature means number of argument, data type of argument and sequence of argument.

```
void sum(int,int);
void sum(int,int,int);
void sum(int,float);
void sum(float,int);
void main()
{
     sum(5,10);
     sum(5,10,15);
     sum(5,10.50);
     sum(10.50,5);
}
void sum(int a,int b)
{
     int c;
     c=a+b;
     cout<<c;
}
void sum(int a,int b, int c)
{
        cout<<(a+b+c);
}
void sum(int a,float b)
{
        cout<<(a+b);
}
void sum(float a,int b)
{
        cout<<(a+b);
 }
```

WAP to define a class template for stack and use as a header file.

```
template<class T,int n>;
class stack
{
   T stk[n];
    int top;
public:
  stack()
   {
        top=-1;
    }
void push(T item)
{
if(top==n-1)
{
    cout<<"Stack overflow";
    return;
}
   top=top-1;
   stk[top]=item;
}
T top()
{
    T delitem;
if(top==-1)
{
    cout<<"Stack empty";
    return;
}
    delitem=stk[top];
    top=top-1;
}
void traverse();
};
void stack::traverse()
{
    int i;
if(top==-1)
{
     cout<<"Stack empty!";
     return;
}
for(i=top;i>=0;i--)
      cout<<stk[i];
}
```

Save the above file with stack.h don't compile the file. Now include the above header file in any program to perform stack operation. In this case we must pass two parameters for template class first may be any data type but second must be an integer value to specify the stack size.

```
#include"stack.h"
void main()
{
    stack<int,5>S1;
    stack<char,5>S2;
    S1.push(25);
    S1.push(35);
    S1.traverse();
    cout<<S1.pop();
    S2.push('A');
    S2.push('B');
    S2.traverse();
    cout<<S2.pop();
}
```