1. **OOPS**: Go through the concepts of C++,  or any other OOP language that you had written deeply. For C++ OOPS, you can refer "*Balagurusamy "* or can go through Saurabh Shukla Videos (in Hindi) or this short course by *freecodeacademy* (in English) on Youtube. **Always have a real-life example ready for each concept of OOP.** For revision purposes, you can refer to short notes from balagurusamy or short notes from GFG.
   For interviews, apart from direct questions, you may also be asked problems like “Design Snake and ladder game using OOP”, “Design a car parking lot using object-oriented principles” etc. to judge your object-oriented skills.
2. **Complete the C quizzes and C++ quiz from GFG**, as some companies ask language MCQs, debugging questions, etc. in the coding test. [And keep on revising the concepts learned from these]

**Only diff bw Structures & Class**

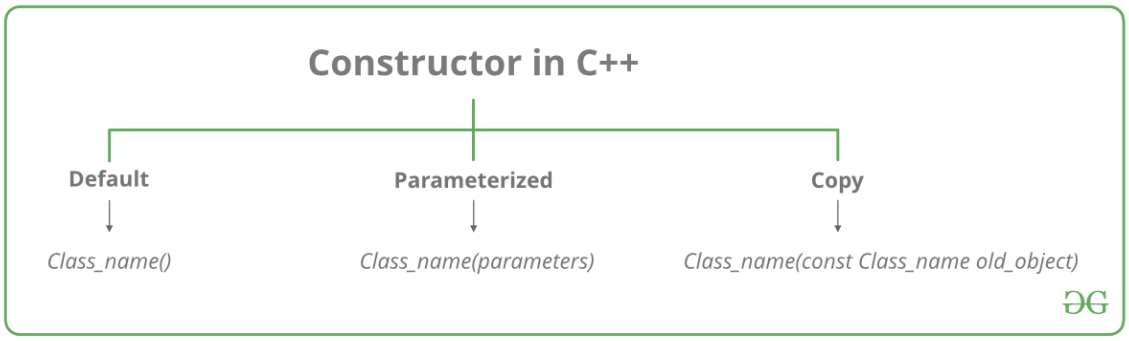
Members of a class are private by default and members of struct are public by default. When deriving a struct from a class/struct, default access-specifier for a struct is public and when deriving a class, default access specifier is private.

# Constructor:

A default constructor will always be called if parameterised constructor is not called.

A constructor is a function that is automatically called when an object is created.

- Constructor has same name as the class itself
- Constructors don’t have return type
- Constructors can be private.We make constructors private when we don't want to create copyable objects. The reason for not making copyable object could be to avoid shallow copy.
- It can’t be static
- It can’t be virtual
- A class can have multiple constructor thus constructor overloading possible




Three ways for object declaration:

1. Complex c1=Complex (3,4);
2. Complex c1(3,4);
3. Complex c1=3;  //can be used only when 1 argument. //3=c1 is operator overloading

## Default Constructor:

No parameters but may have non-empty body

## Parameterized constructor:

Used to initialize objects

## Copy constructor:

A copy constructor is a member function that initializes an object using another object of the same class.
A copy constructor has the following general function prototype:
***ClassName (const ClassName &old_obj);***

```
class Point
{
private:
    int x, y;
public:

     //parameterized constructor
    Point(int x1, int y1) { x = x1; y = y1; }
```

```
    // Copy constructor
    Point(const Point &p1) {x = p1.x; y = p1.y; }
//writing const and & is must .If we don't write '&' above,it will lead to infinite
recursion of complex calling

};

int main()
{
    Point p1(10, 15); // parameterized constructor is called here
    Point p2 = p1; // Copy constructor is called here

    return 0;
}
```

Default copy constr→ shallow copy,parameterised cc→ deep copy(i.e. Different memory)
**Three cases:**

**1.**We have neither declared default/parameterised constructor nor copy constructor→Compiler automatically generates both a default constructor (with empty body) and a copy constructor for the object.

2. We have declared default/parameterised constructor but not copy constructor→Compiler will not make a default constructor but will make a copy constructor for the object.

3.We have declared only copy constructor→Compiler will neither make a default constructor nor a copy constructor for the object.

*Whenever we define one or more non-default constructors( with parameters ) for a class, a default constructor( without parameters ) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it's considered to be the best practice to always define a default constructor.*

Either you call default constr class or copy constr.,the default constr. Of its parent classes will be called always.

# Destructor :

Last function called before objects get destroyed.
Same name as that of class preceded by ~.
Can't be static but can be virtual ,no return type,no arguments (thus no overloading)

Can be private

**Types:**

1. Compiler made:Empty body
2. User made:We can do something inside the body

**Application:.**

To do that thing(inside the body) which can't be done after an object gets destroyed e.g. releasing resources allocated to an object if the pointer to the object is pointing to that resource.

```
int i;

class A
{
public:
    ~A()
    {
        i=10;
    }
};

int foo()
{
    i=3;
    A ob;
    return i;
}

int main()
{
    cout << foo() << endl;
    return 0;
}
```

Run o

A 0

✓ 3

✗ 10

D None of the above

C++ Destructors

Discuss it

**Question 2 Explanation:**

While returning from a function, destructor is the last method to be executed. The destructor for the object "ob" is called after the value of i is copied to the return value of the function. So, before destructor could change the value of i to 10, the current value of i gets copied & hence the output is i = 3. See this for more details.

```cpp
#include <iostream>
using namespace std;
class A
{
    int id;
    static int count;
public:
    A() {
        count++;
        id = count;
        cout << "constructor for id " << id << endl;
    }
    ~A() {
        cout << "destructor for id " << id << endl;
    }
};

int A::count = 0;

int main() {
    A a[3];
    return 0;
}
```

Run on IDE

✔
```
constructor for id 1
constructor for id 2
constructor for id 3
destructor for id 3
destructor for id 2
destructor for id 1
```

B
```
constructor for id 1
constructor for id 2
constructor for id 3
destructor for id 1
destructor for id 2
destructor for id 3
```

C
```
Compiler Dependent.
```

D
```
constructor for id 1
destructor for id 1
```

C++ Destructors
Discuss it

**Question 4 Explanation:**
In the above program, id is a static variable and it is incremented with every object creation. Object a[0] is created first, but the object a[2] is destroyed first. Objects are always destroyed in reverse order of their creation. The reason for reverse order is, an object created later may use the previously created object. For example, consider the following code snippet.

```
A a;
B b(a);
```

In the above code, the object 'b' (which is created after 'a'), may use some members of 'a' internally. So destruction of 'a' before 'b' may create problems. Therefore, object 'b' must be destroyed before 'a'.

# Operator Overloading:

In C++, we can make operators to work for user defined classes.

To overload an operator, we use a special operator function.Operator function returns an object.

Operator function must be member function or **friend function.**

| NO OF ARGUMENT TO BE PASSED | Unary | Binary |
|---|---|---|
| Member function | 0 | 1 |
| Friend function | 1 | 2 |

## Why are friend functions used for operator overloading?

Let c1, c2, and c3 be objects of some class c,then the invocation c3 = c1 + c2 is equivalent to c3 = c1.operator +(c2).
The left one (c1) invokes the function, and the 2nd one (c2) is passed as an argument.
So if we have to overload '+' to do c3 = 100 + c1, here, the left one is not a user-defined data type. It cannot invoke the overloading function. So we have to use the friend function here.
So by using the friend function, that call is equivalent to c3 = operator+(100,c1).

# Operator overloading using member function:

```
class className {
    ... .. ...
    public
        returnType operator symbol (arguments) {
            ... .. ...
        }
    ... .. ...
};
```

Here,

- Operator symbols can be used as a function name if we write 'operator' keyword before the operator symbol.
- symbol is the operator we want to overload. Like: +, <, -, ++, etc.

# Unary operator:

**Prefix operator:** Must have no argument in operator function.
**Postfix operator:** Must have at least one argument in operator function.

#Unary operators

```
class Integer{

private:
int a;

public:
void set_data(int x)
{a=x;}


Integer operator ++()
{
Integer temp;
temp.a=++a;
return temp;
}

Integer operator ++(int)
{
Integer temp;
temp.a=a++;
return temp;
}

};

int main()
{

Integer i1,i2,i3;
i1.set_data(3);

i2=++i1; //i2=i1.operator ++();
i3=i1++; //i3=i1.operator ++(100); //pass any integer as argument

return 0;
}
```

# Binary operator:

```
class Complex{

private:
int a,b;

public:
void set_data(int x,int y)
{a=x,b=y;}

Complex operator +(Complex c1)
{
Complex temp;
temp.a=a+c1.a;
temp.b=b+c1.b;
return temp;
}

};

int main()
{

Complex c1,c2,c3;
c1.set_data(3,4);
c2.set_data(5,6);

//Two ways to write code to call a operator function
//1.c3=c1.operator +(c2);
c3=c1+c2;  //object c1 is calling operator + function and passing argument as c2.

return 0;
}
```

→ In the same way ,operator overloading using the friend function can be performed.

- Two operators = and '&' are already overloaded by default in C++. For example, to copy objects of the same class, we can directly use the = operator. We do not need to create an operator function.
- Operator overloading cannot change the precedence and associativity of operators. However, if we want to change the order of evaluation, parentheses should be used.
- There are 4 operators that cannot be overloaded in C++. They are:
    1. :: (scope resolution)

2. . (member selection)
3. .* (member selection through pointer to function)
4. ?: (ternary operator)

# Friend function and friend class:

**Friend function:** A function is said to be a friend of a class if it can access all private , protected and public members of a class.

A friend function can be:

*//Here foo is friend of class B.:*

**a) A member of another class :**
```
Class B{
friend void A::foo();
}
```

**b) A global function**
```
Class B{
friend foo();
}
```

The function should be declared in class (class whose friend will be that function) as friend using friend keyword.

**Friend class:** A class A is said to be a friend of another class B if all functions of class A can access all private,protected and public members of B.

```
Class B{
friend class A;  //A is friend of B
}
```

- Suppose both manager and worker want to calculate income_tax from their salary, they share their private details with another person(two classes wish to share a particular function). This person: Common function - Friend Function ( not belong to any class )

- Friend function is used for operator overloading
- A function can be a friend of multiple classes.
- Inside class,you can declare friend function/class anywhere i.e. in public, private or protected→ no difference.
- It is defined like a normal function and called like a regular function.No object required to call global friend function
- Usually, friend function's arguments are class objects. They access private members not directly but using the dot operator.

# Function overriding

- When dc has function with the same signature as that of bc,then if object of dc calls the function,function of dc will be called not of bc.

```
class A {
  public:
  void display() {
     cout<<"Base class";
  }
};

class B:public A {
  public:
  void display() {
     cout<<"Derived Class";
  }
};

int main() {
  B obj;
  obj.display();  // Output : "Derived Class"
  return 0;
}
```

It is an example of run time polymorphism.

# Method Hiding

Derived class creates its own method that has the same name as the Base class method. Yet, its signature is different! .
Trying to call the Base version of the method using dc's object without explicitly stating it is the Base version (by using Base:: ) will end up with a compilation error.In this way dc method have hidden bc method.

```
class A {
  public:
  void display() {
     cout<<"Base class";
  }
};

class B:public A {
  public:
  void display(int x) {
     cout<<"Derived Class  "<<x;
  }
};

int main() {
  B obj;
  obj.display(5);  // Output : "Derived Class   5"
  obj.display();    // Error
  A::obj.display();   //  Output : “Base Class"

  return 0;
}
```

---

# Virtual functions:

- Runtime polymorphism

A class's pointer (A pointer having type as any class) can also point to objects of all its descendent classes.

Early binding:Done at compile time according to type of pointer.
Late binding:Done at run time according to the content of the pointer.

A virtual function is a member function which is declared within a base class and is **overridden by a derived class**.Called at run time.(Overriding means exactly same signature).
It is not mandatory for the derived class to override ;**in that case, the base class version of the function is used.**
We don't need virtual keywords in the derived class, **functions of derived class** are automatically considered as virtual functions if they are overriding bc virtual fns.
A pointer of base class type can't call those functions of derived class that are not virtual. (`p->fun_4(5);` in below example).

- Virtual functions cannot be static.
- A virtual function can be a friend function of another class.
- A class may have a virtual destructor but it cannot have a virtual constructor.//already studied

```
class base {
public:
        void fun_1() { coût << "base-1\n"; }
        virtual void fun_2() { cout << "base-2\n"; }
        virtual void fun_3() { cout << "base-3\n"; }
        virtual void fun_4() { cout << "base-4\n"; }
};

class derived : public base {
public:
        void fun_1() { cout << "derived-1\n"; }
        void fun_2() { cout << "derived-2\n"; }
        void fun_4(int x) { cout << "derived-4\n"; }
};

int main()
{
        base *p;
        derived obj1;
```

```
        p = &obj1;

        // Early binding because fun1() is non-virtual
        // in base
        p->fun_1();

        // Late binding (RTP)
        p->fun_2();

        // Late binding (RTP)
        p->fun_3();  //we have no fun_3() in dc,bc fn will be called

        // Late binding (RTP)
        p->fun_4();   //we have no fun_4() in dc,bc fn will be called

        // Late binding (RTP)
         p->Base::fun_2();

        // Early binding but this function call is
        // illegal (produces error) because pointer
        // is of base type and function is of
        // derived class,p can only call virtual fn
        // p->fun_4(5);  //we have

        return 0;
}
```

**Output:**

```
base-1
derived-2
base-3
Base-4

base-2
```

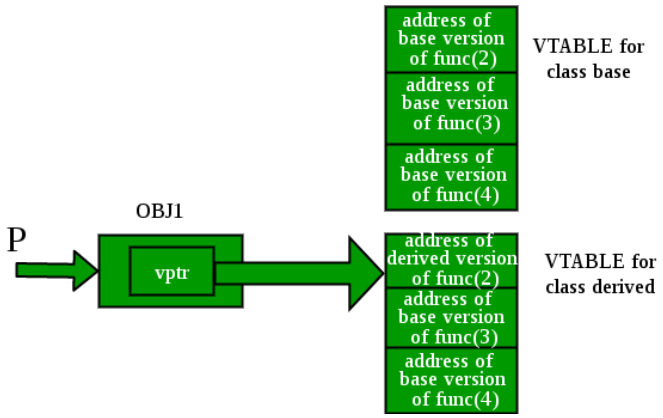
# VTABLE:

Each class that contains at least one virtual function has its own unique VTABLE. VTABLE is a static member array of type - function pointers. Cells of this table store the address of each

virtual function contained in that class. This VTABLE is created by the compiler irrespective of object creation.
Needless to say,if we don't override the function in dc, VTABLE of dc contains the address of function of the base class version.

## VPTR:

If an object of that class is created then the compiler creates a virtual pointer (VPTR) as a data member of the class to point to the starting address of the VTABLE of that class.Each object has its unique vptr.
Below,since the object is of derived class,vptr will point to VTABLE of dc.




## How late-binding is achieved through vptr and vtable.

When a function call is made through a base class pointer, the compiler inserts the code to fetch the VPTR of that object which is content of bc pointer and through this VPTR lookups function address in the VTABLE, thus calling the right function and causing Late Binding

### Application of virtual functions:

1.**Declaration in bc,definition in dc:**Sometimes implementations of some function cannot be provided in a base class because we don't know the implementation. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). So declaring draw() in Shape class is better because it is a property of any Shape.

2.To achieve run-time polymorphism.
https://stackoverflow.com/questions/2391679/why-do-we-need-virtual-functions-in-c?rq=1

# Virtual destructor:

Make base class destructor virtual (late binding) to ensure the object of derived class is destructed properly (if not ,only base class destructor will be deleted, i.e., both base class and derived class destructors are called.
Always make destructor virtual when there is a virtual function.

```
class base {
public:
        base()
        { cout<<"Constructing base \n"; }
        virtual ~base()
        { cout<<"Destructing base \n"; }
};
class derived: public base {
public:
        derived()
        { cout<<"Constructing derived \n"; }
        ~derived()
        { cout<<"Destructing derived \n"; }
};

int main(void)
{
base *b = new derived();  //b is a b type pointer which points object of derived type
delete b; //If early binding object of b will get deleted and destructor of derived will not called
getchar();
return 0;
}
```

**Output when not virtual destructor:**

```
Constructing base
Constructing derived
Destructing base
```

**Output when virtual destructor:**

```
Constructing base
Constructing derived
Destructing derived
Destructing base
```

# Abstract class:

## Pure virtual function:

It is a do nothing virtual function, written in base class as : `virtual void show() = 0;`

A class having at least one pure virtual function is an abstract class.We can't make objects of abstract class but We can have pointers or references of abstract classes. We have to override all pure virtual functions of bc in all its dc otherwise dc will also become an abstract class.
Every class containing pure vrt fn must be declared abstract.

### Applications of abstract class:

1.Suppose there are two class student and faculty and they have many common features .We can store these common features in another class Person but we are not interested to store information (i.e. to create object) of Person class.

2. and 3. advantage same as of virtual functions.

# Inheritance:

The capability of a class to derive properties and characteristics from another class is called Inheritance.
DC objects can't access BC private members but whenever a DC object is created ,memory of even BC private members is created.

**Syntax:**

```
class subclass_name : access_mode base_class_name
{
  //body of subclass
};
```

| Base class member access specifier | Type of Inheritence | | |
|---|---|---|---|
| | Public | Protected | Private |
| Public | Public | Protected | Private |
| Protected | Protected | Protected | Private |
| Private | Not accessible (Hidden) | Not accessible (Hidden) | Not accessible (Hidden) |

- Multiple Inheritance:Ek child ke 2 papa.
- Hierarchical Inheritance: Ek papa ke 2 bachhe
- Hybrid Inheritance:Combination of 2 or more types of inheritance.

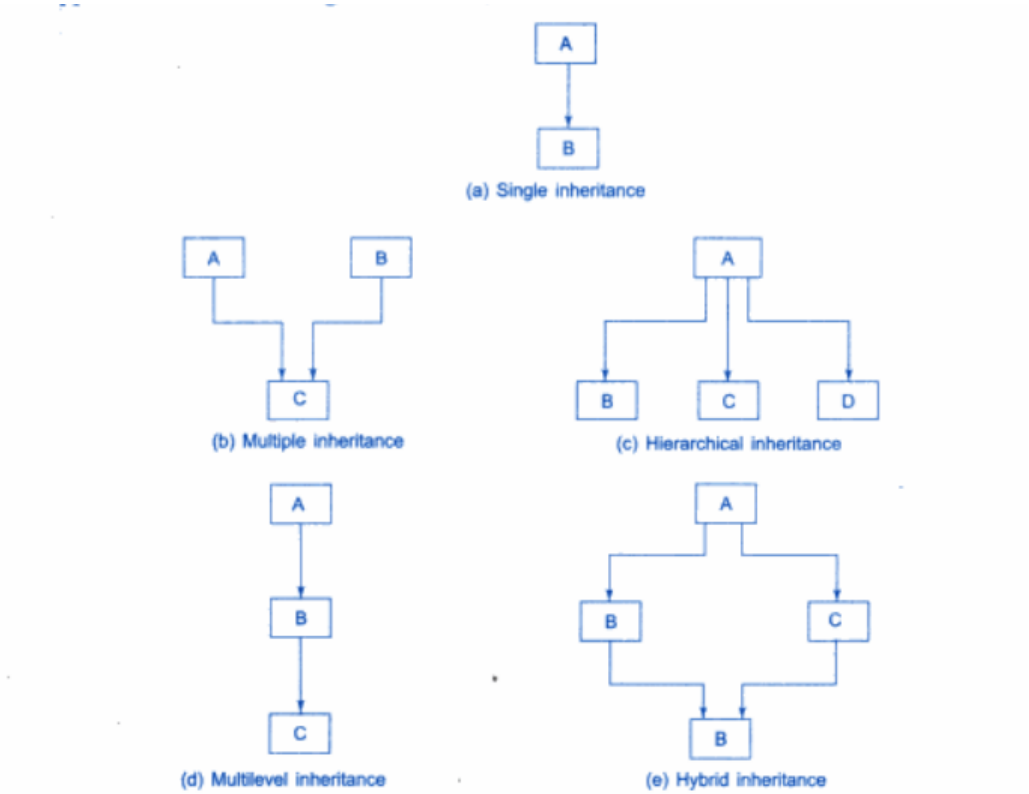



Fig. 8.1 ⇔ *Forms of inheritance*

# Dreaded Diamond problem:

The diamond problem occurs when two superclasses of a class have a common base class.Here B1 and B2 superclasses of C have same base class A.So C will have multiple copies of members of A.This will result in i.ambiguity and ii.space wastage.

This can be resolved by

It can be resolved using any of the 2 methods :

1. Only Ambiguity resolved:Use scope resolution operator to avoid ambiguity, but still there are 2 copies **e.g.** obj.ClassB1::a = 10;

2. Deriving the common base class a virtual base class while declaring it in immediate derived class.When a class is made virtual BC, only one copy of that class is inherited (ambiguity solved, and space saved), regardless of how many inheritance paths exist between the virtual BC and target DC.

```
class A                          // grandparent
{
      .....
      .....
};
class B1 : virtual public A      // parent1
{
      .....
      .....
};
class B2 : public virtual A      // parent2
{
      .....
      .....
};
class C : public B1, public B2   // child
{
      .....                 // only one copy of A
      .....                 // will be inherited
};
```

Order of execution of constructors & destructors in Inheritance:

| *Method of inheritance* | *Order of execution* |
|---|---|
| Class B: public A<br>{<br>}; | A( ) ; base constructor<br>B( ) ; derived constructor |
| class A : public B, public C<br>{<br>}; | B( ) ; base(first)<br>C( ) ; base(second)<br>A( ) ; derived |
| class A : public B, virtual public C<br>{<br>}; | C( ) ; virtual base<br>B( ) ; ordinary base<br>A( ) ; derived |

**Order of Inheritance**

**Class C** (Base Class 2)
↓
**Class B** (Base Class 1)
↓
**Class A** (Derived Class)

**Order of Constructor Call**

1. **C()** (Class C's Constructor)
2. **B()** (Class B's Constructor)
3. **A()** (Class A's Constructor)

**Order of Destructor Call**

1. **~A()** (Class A's Destructor)
2. **~B()** (Class B's Destructor)
3. **~C()** (Class C's Destructor)

**Destructor calling is always exactly opposite to constructor calling**

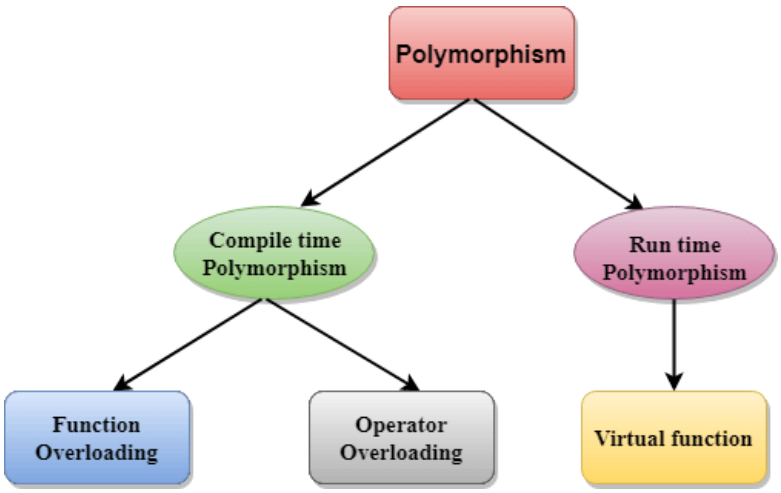
Calling bc parameterized constructor in dc:

- Whenever the derived class's default constructor is called, the base class's default constructor is called automatically.
- The parameterized constructor of base class cannot be called in the default constructor of dc, it should be called in the parameterized constructor of dc (because we put arguments of bc cons. in constructor of dc ).To call the parameterized constructor of base class inside the parameterized constructor of dc, we have to mention it explicitly using colon

```
D(int a1, int a2, float b1, float b2, int d1):
A(a1, a2),          /* call to constructor A */
B(b1, b2)          /* call to constructor B */
{
     d = d1;     // executes its own body
}
```

# Polymorphism:

In OOP, Polymorphism is the ability of a message to be interpreted in more than one form.
**Real life example:** Like a man at the same time is a father, a husband, an employee




## Binding

For function calls, binding means matching the call with the right function definition by the compiler. It takes place either at compile time or at runtime.
There are 2 types of binding :
i. Early Binding or static binding (Compile-time time polymorphism)
ii. Late Binding or dynamic binding (Runtime polymorphism).

# Data encapsulation

**Real life example:**

Consider a company with various divisions like financial, technical, marketing, etc. A division consists of the related records and the employees that manipulate the records. Here, the employees are functions that manipulate the data and are wrapped into a single division.

- Wrapping up data and functions into a single unit  called class.
- Encapsulation also leads to data abstraction or hiding.

# Data hiding:

**Real life example:**The manager wants to access the records of the financial team. They cannot access the records ( data) directly and have to ask the employees ( functions ) for it.

Data members can't be directly accessed  but only by functions.This is called data hiding.

## Diff b/w Data encapsulation & Data hiding:

1. The main difference between data hiding and encapsulation is that the former focuses on enhancing data security in the program, while the latter deals with hiding the program's complexity.
2. In data hiding, the data has to be defined as private only. In data encapsulation, the data can be public or private.

# Data Abstraction:

Displaying essential information without including irrelevant background details.Intention is to reduce complexity.
**Example :**

1.A light bulb glows on, turning a switch on. The internal implementation remains unknown.
2.When using any function like the pow() function, its implementation remains hidden.
3.Using Classes: Access specifiers are the main pillar of implementing abstraction in C++. Internal implementation can be marked as private and interaction with the outside world public.

**Advantages :**

1. helps the user to avoid writing low-level code
2. internal implementation can be changed w/o affecting the user.
3. Increases security.

# Namespace:

Header files contain declaration of functions/variables/classes and macros.
Library file contains function  definitions and namespaces.
Namespace is a container for identifiers.Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes, giving them namespace scope.
Using namespaces, we can create two variables or member functions having the same name.
There cannot be two variables with the same name in the same scope.

```
namespace ns1
{
int value()    { return 5; }
}
namespace ns2
{
   const double x = 100;
   double value() {  return 2*x; }
}
int main()
{    cout << ns1::value() << '\n';
      cout << ns2::value() << '\n';
      coût << ns2::x << '\n';
   return 0;
}
```

- We can use different name for our namespace name for ease of use

```
namespace ms=MySpace;
```

- Namespace declarations appear only at global scope, not inside any function.
- Namespace declarations can be nested within another namespace.*void MySpace::A::fun1() {  }//fun1() is declared in Class A and class A is declared in MySpace namespace.*
- **using** keyword imports an entire namespace into your program with a global scope i.e. after writing this we don't have to write scope resolution operator for using members of this namespace.For e.g.,members like cin,cout are in namespace 'std' which is in iostream file i.e.when you are using cout, it is actually std::cout
- Namespace declarations don’t have access specifiers. (Public or private)
- Class/Functions can be declared inside namespace and defined outside namespace
- Namespace is not a class so you can't create objects of namespace.
- There can be unnamed namespaces too.// namespace {  //declarations )

Namespaces can be extended i.e. they can be continued and extended over multiple files.They are not redefined or overridden.

# Dynamic Constructor:

When allocation of memory is done dynamically (using pointer) using dynamic memory allocator new in a constructor, it is known as dynamic constructor.

```
class geeks {
	int* p;
public:
	geeks(int x)
	{
		p = new int;
		*p = x;
	}
	void display()
	{
		cout << *p << endl;
	}
};

int main()
{
	geeks obj1 = geeks();
	obj1.display();
	geeks obj2 = geeks(7);
	obj2.display();
}
```

# Nested Class:

- We make a class inside a class when the object of the nested class has no real significance.For example, address class (having variables as city,state and pincode) is nested inside student class.Address class has no real significance.
- Treat the nested class as a member variable of the enclosing class like other member variables. Thus it has the same access rights as the other members.
- The members of an enclosing class have no special access to members of a nested class; the usual access rules shall be obeyed i.e. private members of nested class can't be accessed by enclosing class.
- We can create an object of a nested class in the enclosing class.
- We can also make an object of the nested class in main(),if the nested class is a public member of the enclosing class. // Student::Address a1;

# Initialiser List:

Initializer List is used in initializing the data members of a class.

Using initialiser list ,we initialize:

1. non-static const data members:
2. Reference members

**Why?**
In a class we can't initialize variables while declaration so we initialize them in the constructor.But constant variables and reference variables need to initialize while declaration so this problem is solved by initialisers.

```
class Dummy
{
private:
int a,b; //b=10; -> will give error
const int x;  // initializing non-static const data members
int &y;        //reference members:
public:
Dummy (int &n):a(10),x(5),y(n)
{
b=10;
}
};
void main(){
int m=6;
Dummy d1(m);
}
```

# Type Conversion:

4 Types of conversion

**1.implicit to implicit: Automatic**

**2.Primitive to class type: Using constructor**

```
class Complex{
```

```
private:
int a,b;
public:
Complex() {} //since we have made a parameterised constructor,compiler will not make one,so we have to make a default constructor
Complex(int k)
{a=k,b=0;}
void set_data(int x,int y)
{a=x,b=y;}
};
void main() {
int x=5;
Complex c1;
c1=x; // c1.Complex(x)
}
```

**3.Class type to Primitive: Using operator overloading concept**

**Syntax**:operator return_type() {return data;} //for overloading '=' operator

```
class Complex{
private:
int a,b;
public:
void set_data(int x,int y)
{a=x,b=y;}
operator int()
{
return (a);
}
};
void main() {
Complex c1;
c1.setData(3,4);
int x=c1; // c1.operator int(); //right side mei hai vo call kar raha aur left side waale ko return
}
```

**4.Class type to class type: Using constructor OR casting operator**

```
class Product{
private:
int a,b;
public:
```

```
void set_data(int x,int y)
{a=x,b=y;}
int get_a() {return a;} // since a and b are private members we can't access them directly so we
//made a public function to access them from another class
int get_b() {return b;}
operator Item() //using casting operator
{
Item temp;
temp.setData(a,b);
return (temp);
}
};

class Item{
private:
int a,b;
public:
void set_data(int x,int y)
{a=x,b=y;}
Item() {}
Item(Product p) //using constructor
{
a=p.get_a(),b=p.get_b();
}
};

void main()
{
Product p;
p.setData(3,4);
Item i=p; //p will call operator and return Item datatype
}
```

# Reference variable

Four types of variable
1.Ordinary variable : int x=5;
2.pointer variable int* p=&x;
3.const variable :const int x=5; //must be initialized while declaration

**4.reference variable:**
It is an **internal pointer** which stores address of a already initiated variable
It must be initialized while declaration

It can't be updated
int x=6;
int &y=x;
y++;
cout<<x<<<" "<<y; // 7 7
x=30;
cout<<x<<<" "<<y; // 30 30

# object pointer and this keyword

# object pointer

The ‘this’ pointer is passed as a hidden argument to all non-static member function calls and is available as a local variable within the body of all non-static functions. **‘this’ pointer is a constant pointer** that holds the memory address of the current object. ‘this’ pointer is not available in static member functions as static member functions can be called without any object .

```
class Box{
private:
int l,b,h;
public:
void setDimensions(int l,int b,int h)
{
//this is a local object pointer in every instance member function containing address of caller object
//it is used to refer caller object in member function
//this pointer can't be modified
this->l=l,this->b=b,this->h=h;
}
};

void main()
{
Box *p,b1; //p is a object pointer of datatype box
p=&b1;     //p is storing address of b1 object
p->setDimensions(12,10,5); //eqv. to p.setDimensions(12,10,5);
}
```

```
class Test
{
private:
  int x;
public:
  Test(int x = 0) { this->x = x; }
  void change(Test *t) { this = t; }
  void print() { cout << "x = " << x << endl; }
};

int main()
{
  Test obj(5);
  Test *ptr = new Test (10);
  obj.change(ptr);
  obj.print();
  return 0;
}
```

A x = 5

✗ x = 10

✓ Compiler Error

D Runtime Error

C++ this pointer
Discuss it

**Question 3 Explanation:**
this is a const pointer, so there is an error in line "this = t;"

# new and delete

For dynamic memory allocation(DMA) of variables(i.e. at runtime),we use a new keyword. These variables don't have names and can only be accessed through their pointer variable.

//datatype *pointer_variable_name=new data type;
int *p1=new int; //(*p1)=20;
int *p2=new int[5]; //array
Complex *p3=new Complex

//DMA variables are not automatically deleted like static memory allocated(SMA) variables.
delete p1,[]p2,p3;
Dynamic array of pointer having size 10 using new is created as,

```
int **arr = new int *[10];
```

```
 We can create a non-dynamic array using int *arr[10]
```

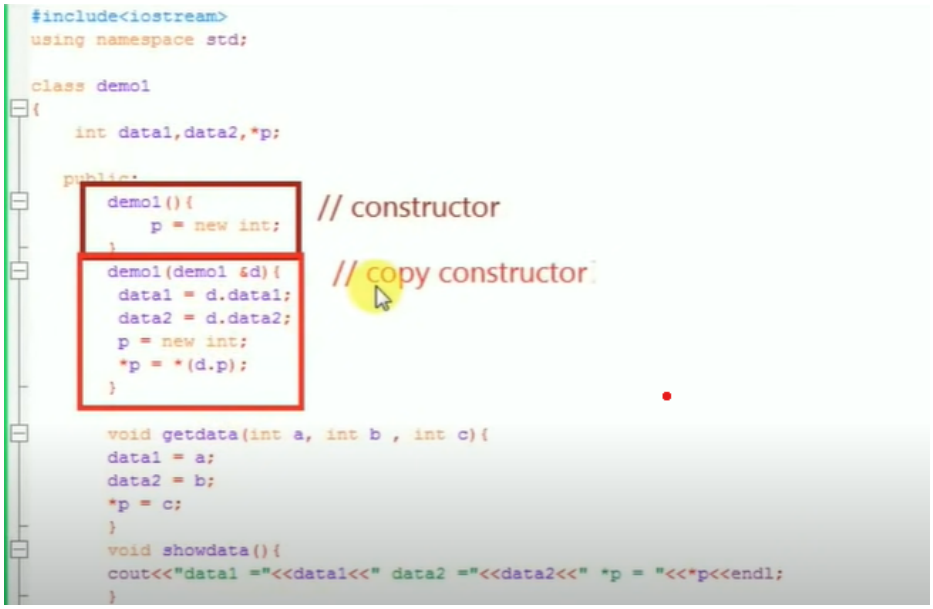
delete operator works only for objects allocated using operator new (See this post). If the object is created using new, then we can do delete this, otherwise behavior is undefined.

Deleting a null pointer has no effect, so it is not necessary to check for a null pointer before calling delete.
It is undefined behavior to call delete twice on a pointer. Anything can happen, the program may crash or produce nothing.

# Shallow copy vs deep copy

Dummy d2=d1; //when we write like this ,where compiler writes code itself ,it is shallow copy.

When we copy object's pointer variable to another ,it will not be copied and will share same address so we have to write manual code if there is pointer type variable like below.




# Exception handling:

- Exceptions are run-time abnormal conditions that a program encounters during its execution.

try: represents a block of code that can throw an exception.
catch: represents a block of code that is executed when a particular exception is thrown.
throw: The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

When an exception is thrown and not caught, the program terminates abnormally.
The catch all block must be written after all catch blocks otherwise it is a compile error.
*After the try block, the destructors are called only for* ***completely constructed objects***

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

int main()
{
    int a=-1;
    try  {
       cout<<”try1”;
       if(a>0)
        throw 10;
    }

     try{
     cout<<”try2”;
     if(a<0)
     throw ‘c’
     }
    catch (char *excp)  {
        cout << "Caught " << excp;
    }
    catch (...)  {
        cout << "Default Exception\n";
    }
    return 0;
}
```

Code after throw inside try is not executed.

***Output:***
*try1 try2 Caught*

```
class Base {};
class Derived: public Base {};
int main()
{
   Derived d;
   try {
       throw d;
   }
   catch(Base b) {
        cout<<"Caught Base Exception";
   }
   catch(Derived d) {
        cout<<"Caught Derived Exception";
   }
   return 0;
}
```

Run on ID

A Caught Derived Exception

✓ Caught Base Exception

C Compiler Error

**C++ Exception Handling**
**Discuss it**

**Question 4 Explanation:**
If both base and derived classes are caught as exceptions then catch block of derived class must appear before the base class. If we put base class first then the derived class catch block will never be reached. In Java, catching a base class exception before derived is not allowed by the compiler itself. In C++, compiler might give warning about it, but compiles the code.

```
int main()
{
    try
    {
        try
        {
            throw 20;
        }
        catch (int n)
        {
            cout << "Inner Catchn";
            throw;
        }
    }
    catch (int x)
    {
        cout << "Outer Catchn";
    }
    return 0;
}
```

Run on IDE

A
```
Outer Catch
```

B
```
Inner Catch
```

✓
```
Inner Catch
Outer Catch
```

✗ Compiler Error

**C++ Exception Handling**
**Discuss it**

**Question 7 Explanation:**
The statement 'throw;' is used to re-throw an exception. This is useful when a function can handles some part of the exception handling and then delegates the remaining part to the caller. A catch block cleans up resources of its function, and then rethrows the exception for handling elsewhere.

```
class Test {
public:
   Test() { cout << "Constructing an object of Test " << endl; }
  ~Test() { cout << "Destructing an object of Test "  << endl; }
};

int main() {
  try {
    Test t1;
    throw 10;
  } catch(int i) {
    cout << "Caught " << i << endl;
  }
}
```

Run on IDE

A. Caught 10

B.
```
Constructing an object of Test
Caught 10
```

✓
```
Constructing an object of Test
Destructing an object of Test
Caught 10
```

D. Compiler Errror

C++ Exception Handling
Discuss it

**Question 8 Explanation:**
When an object is created inside a try block, destructor for the object is called before control is transferred to catch block.

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

class Test {
  static int count;
  int id;
public:
  Test() {
    count++;
    id = count;
    cout << "Constructing object number " << id << endl;
    if(id == 4)
       throw 4;
  }
  ~Test() { cout << "Destructing object number " << id << endl;
}
};

int Test::count = 0;

int main() {
  try {
    Test array[5];
  } catch(int i) {
    cout << "Caught " << i << endl;
  }
}
```

***Output:***

*Constructing object number 1*
*Constructing object number 2*
*Constructing object number 3*
*Constructing object number 4*
*Destructing object number 3*
*Destructing object number 2*
*Destructing object number 1*
*Caught 4*

*→ After the try block, the destructors are called only for completely constructed objects.*

# Static Members:

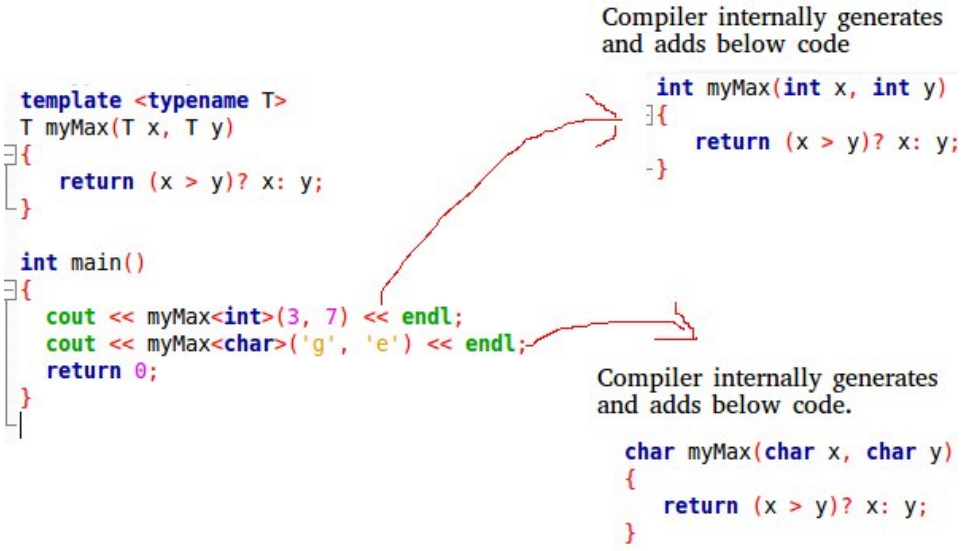
Static methods can only access static data members but static data members can be accessed by both static / non static methods.
If the static function is to be defined outside the class then static keyword must be present in function declaration only not in the definition outside the class.

## Templates in C++

Templates are used to pass data type( even class ) as a parameter so that we don’t need to write the same code for different data types.
For example, a software company may need to sort() for different data types. Rather than writing and maintaining multiple codes, we can write one sort() and pass data type as a parameter.

Templates are example of compile time polymorphism. Templates are expanded at compile-time. Since it is more difficult to debug macros,templates are more efficient than macros.

There is no difference between the **typename** and **class** keywords. Both of the keywords are interchangeably used by the C++ developers




There are 2 types of templates:

**1.Function templates:**
When write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray(). //above image

**2.Class templates:**
Like function templates, class templates are useful when a class defines something that is independent of the data type. Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

```
template <typename T> class Array {
private:
    T* ptr;
    int size;

public:
    Array(T arr[], int s);
    void print();
};
```

## Important points

- There can be more than one argument to templates
- There can be a default value for arguments to templates

*//code for both points*

```
template <class T, class U = char> class A {
public:
    T x;
    U y;
    A() { cout << "Constructor Called" << endl; }
};

int main()
{
    A<char> a; // This will call A<char, char>
    return 0;
}
```

- We can pass non-type arguments(i.e. No typename) to templates. Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of a template. The important thing to note about non-type parameters is, that they must be const. The compiler must know the value of non-type parameters at compile time. Because the compiler needs to create functions/classes for a specified non-type value at compile time.

```
template <class T, int max> int arrMin(T arr[], int n)
{
    int m = max;
    for (int i = 0; i < n; i++)
        if (arr[i] < m)
            m = arr[i];

    return m;
}

int main()
{
    int arr1[] = { 10, 20, 15, 12 };
    int n1 = sizeof(arr1) / sizeof(arr1[0]);

    char arr2[] = { 1, 2, 3 };
    int n2 = sizeof(arr2) / sizeof(arr2[0]);

    // Second template parameter to arrMin must be a
    // constant
    cout << arrMin<int, 10000>(arr1, n1) << endl;
    cout << arrMin<char, 256>(arr2, n2);
    return 0;
}
```

- *When there is a static member in a template class/function,each instance of a template contains its own static variable.*
- *We can have different codes for a particular data type using **template specialisation***
- *(only topics not studied in templates:not studied template specialisation and template metaprogramming)*

---

```
Point *t1, *t2;   // No constructor call,since  t2 is just a
pointer variable, not an object
t1 = new Point(10, 15);  // Normal constructor call
t2 = new Point(*t1);   // Copy constructor call
Point t3 = *t1;  // Copy Constructor call
Point t4;   // Normal Constructor call
t4 = t3;   // Assignment operator call,not copy constructor
```

```
X a = {10}; // a object can be initialised like this like
structures
```

Unlike new, malloc() doesn't call constructor .

Size of a empty class is 1 byte.

Header file can not be passed to a function in C++. While array, constant and structure can be passed into a function

Question 9 Explanation:
In C++ class template and function template are similar in the way the are initiated. Class template are not used for storage class. Class templates and function templates are instantiated in the same way and Class template is not initiated by defining an object using the template.

It is possible to define a class within a class termed as nested class. There are two types of nested classes. 1 - Outer class will use argument of inner class. 2 - Inner and outer class are independent to each other.

A const object can only call const functions.

Static fns can’t be virt. bcz static functions are class specific and may not be called on objects. Virtual functions are called according to the pointed or referred object.

New vs malloc
1) new is an operator, malloc is a function 2) new calls constructor, malloc doesn't 3) new returns appropriate pointer, malloc returns void * and pointer needs to typecast to appropriate type.

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

class Test
{
    static int x;
public:
    Test() { x++; }
    static int getX() {return x;}
};

int Test::x = 0;

int main()
{
    cout << Test::getX() << " ";
    Test t[5];
    cout << Test::getX();
}
```

Run on I

A 0 0

B 5 5

✓ 0 5

D Compiler Error

C++ Static Keyword
Discuss it

**Question 1 Explanation:**
Static functions can be called without any object. So the call "Test::getX()" is fine. Since x is initialized as 0, the first call to getX() returns 0. Note the statement x++ in constructor. When an array of 5 objects is created, the constructor is called 5 times. So x is incremented to 5 before the next call to getX().

```
class A
{
private:
    int x;
public:
    A(int _x)  {  x = _x; }
    int get()  { return x; }
};

class B
{
    static A a;
public:
   static int get()
   {  return a.get(); }
};

int main(void)
{
    B b;
    cout << b.get();
    return 0;
}
```

Run on

A 0

✔ Linker Error: Undefined reference B::a

C Linker Error: Cannot access static a

D Linker Error: multiple functions with same name get()

C++ Static Keyword
Discuss it

**Question 4 Explanation:**
There is a compiler error because static member a is not defined in B. To fix the error, we need to explicitly define a. The following program works fine.

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

class A
{
private:
    int x;
public:
    A(int _x)  {  x = _x; }
    int get()  { return x; }
};

class B
{
    static A a;
public:
   static int get()
   {  return a.get(); }
};

A B::a(0);

int main(void)
{
    B b;
    cout << b.get();
    return 0;
}
```

```
<br>
#include < iostream ><br>
using namespace std;<br>
class Point {<br>
Point() { cout << "Constructor called"; }<br>
};<br><br>
int main()<br>
{<br>
Point t1;<br>
return 0;<br>
}<br>
```

Run on IDE

A Runtime Error

B None of these

C Constructor called

✓ Compiler Error

C++ Constructors C Operators
Discuss it

**Question 4 Explanation:**

By default all members of a class are private. Since no access specifier is there for Point(), it becomes private and it is called outside the class when t1 is constructed in main.

Question 8 CORRECT

What is the output of following program?

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

class Point
{
    int x, y;
public:
   Point(const Point &p) { x = p.x; y = p.y; }
   int getX() { return x; }
   int getY() { return y; }
};

int main()
{
    Point p1;
    Point p2 = p1;
    cout << "x = " << p2.getX() << " y = " << p2.get'
    return 0;
}
```

Run on IDE

A x = garbage value y = garbage value

B x = 0 y = 0

✓ Compiler Error

C++ Constructors
Discuss It

**Question 8 Explanation:**
There is compiler error in line "Point p1;". The class Point doesn't have a constructor without any parameter. If we write any constructor, then compiler doesn't create the default constructor. It is not true other way, i.e., if we write a default or parameterized constructor, then compiler creates a copy constructor. See the next question.

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

class Point
{
    int x, y;
public:
   Point(int i = 0, int j = 0) { x = i; y = j; }
   int getX() { return x; }
   int getY() { return y; }
};

int main()
{
    Point p1;
    Point p2 = p1;
    cout << "x = " << p2.getX() << " y = " << p2.get`
    return 0;
}
```

Run on IDE

A Compiler Error

✓ x = 0 y = 0

C x = garbage value y = garbage value

C++ Constructors
Discuss it

**Question 9 Explanation:**
Compiler creates a copy constructor if we don't write our own. Compiler writes it even if we have written other constructors in class. So the above program works fine. Since we have default arguments, the values assigned to x and y are 0 and 0.

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

class Test<br>
{<br>
public:<br>
Test() { cout << "Hello from Test() "; }<br>
} a;<br><br>

int main()<br>
{<br>
cout << "Main Started ";<br>
return 0;<br>
}
```

Run on

A Main Started

B Main Started Hello from Test()

✓ Hello from Test() Main Started

D Compiler Error: Global objects are not allowed

C++ Constructors
Discuss it

**Question 11 Explanation:**
Output is

```
Hello from Test() Main Started
```

There is a global object 'a' which is constructed before the main functions starts, so the constructor for a is called first, then main()' execution begins.

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

class Test
{
public:
  Test();
};

Test::Test()  {
    cout << " Constructor Called. ";
}

void fun() {
  static Test t1;
}

int main() {
    cout << " Before fun() called. ";
    fun();
    fun();
    cout << " After fun() called. ";
    return 0;
}
```

Run on IDE

A Constructor Called. Before fun() called. After fun() called.

✗ Before fun() called. Constructor Called. Constructor Called. After fun() called.

✓ Before fun() called. Constructor Called. After fun() called.

D Constructor Called. Constructor Called. After fun() called.Before fun() called.

C++ Constructors
Discuss it

**Question 16 Explanation:**
Note that t is static in fun(), so constructor is called only once.

Implicit return type of a class constructor is:

- ✗ not of class type itself
- ✓ class type itself
- C a destructor of class type
- D a destructor not of class type

**UGC NET CS 2016 July – III** **C++ Constructors**
**Discuss it**

**Question 21 Explanation:**
Truly, constructor doesn't have any return type not even void. However, as the definition of Constructor goes, it is used to initialize an object of the class. So implicitly, they return the current instance of the class whose constructor it is. Therefore, Implicit return type of a class constructor is class type itself. For more information Refer:C++ Classes and Objects Option (B) is correct.

Predict the output of following program?

```
#include <iostream>
using namespace std;
class Test
{
private:
    int x;
public:
    Test(int i)
    {
        x = i;
        cout << "Called" << endl;
    }
};

int main()
{
    Test t(20);
    t = 30; // conversion constructor is called here.
    return 0;
}
```

Run on IDE

- A Compiler Error
- ✓
```
Called
Called
```
- C
```
Called
```

C++ Constructors
Discuss it

**Question 17 Explanation:**
If a class has a constructor which can be called with a single argument, then this constructor becomes conversion constructor because such a constructor allows automatic conversion to the class being constructed. A conversion constructor can be called anywhere when the type of single argument is assigned to the object. The output of the given program is

```
Called
Called
```

*// if x=c,operator overloading,if c=x parameterised constr. called*

Which of the following is true about the following program

```
#include <iostream>
class Test
{
public:
    int i;
    void get();
};
void Test::get()
{
    std::cout << "Enter the value of i: ";
    std::cin >> i;
}
Test t;  // Global object
int main()
{
    Test t;  // local object
    t.get();
    std::cout << "value of i in local t: "<<t.i<<'n';
    ::t.get();
    std::cout << "value of i in global t: "<<::t.i<<'n';
    return 0;
}
```

Run on IDE

Contributed by **Pravasi Meet**

A Compiler Error: Cannot have two objects with same class name

✗ Compiler Error in Line "::t.get();"

✓ Compiles and runs fine

Class and Object
Discuss it

**Question 6 Explanation:**
The above program compiles & runs fine. Like variables it is possible to create 2 objects having same name & in different scope.

Which of the following is not a member of class?

A Static function

✓ Friend function

✗ Const function

D Virtual function

UGC NET CS 2014 Dec - II  Class and Object
Discuss it

**Question 17 Explanation:**
Friend function is not a member of class. For more information on Friend class Refer:Friend class and function in C++ Option (B) is correct.

When one object reference variable is assigned to another object reference variable then

A a copy of the object is created.

✓ a copy of the reference is created.

✗ a copy of the reference is not created.

D it is illegal to assign one object reference variable to another object reference variable.

Which of the following is not a correct statement?

A Every class containing abstract method must be declared abstract.

✓ Abstract class can directly be initiated with 'new' operator.

✗ Abstract class can be initiated.

D Abstract class does not contain any definition of implementation.

Which of the following overloaded functions are NOT allowed in C++? 1) Function declarations that differ only in the return type

```
int fun(int x, int y);
    void fun(int x, int y);
```

2) Functions that differ only by static keyword in return type

```
int fun(int x, int y);
    static int fun(int x, int y);
```

3)Parameter declarations that differ only in a pointer * versus an array []

```
int fun(int *ptr, int n);
int fun(int ptr[], int n);
```

4) Two parameter declarations that differ only in their default arguments

```
int fun( int x, int y);
int fun( int x, int y = 10);
```

✓ All of the above

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

int fun(int=0, int = 0);

int main()
{
  cout << fun(5);
  return 0;
}

int fun(int x, int y) { return (x+y); }
```

Compiler Error

5

0

10

Assume that an integer takes 4 bytes and there is no alignment in following classes, predict the output.

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

class base {
    int arr[10];
};

class b1: public base { };

class b2: public base { };

class derived: public b1, public b2 {};

int main(void)
{
  cout << sizeof(derived);
  return 0;
}
```

Run on IDE

A 40

✓ 80

✗ 0

D 4

C++ Inheritance
Discuss it

**Question 3 Explanation:**
Since b1 and b2 both inherit from class base, two copies of class base are there in class derived. This kind of inheritance without virtual causes wastage of space and ambiguities. virtual base classes are used to save space and avoid ambiguities in such cases. For example, following program prints 48. 8 extra bytes are for bookkeeping information stored by the compiler (See this for details)

```
#include<iostream>

using namespace std;
class P {
public:
   void print()  { cout <<" Inside P"; }
};

class Q : public P {
public:
   void print() { cout <<" Inside Q"; }
};

class R: public Q { };

int main(void)
{
  R r;
  r.print();
  return 0;
}
```

Run on IDE

A Inside P

✓ Inside Q

✗ Compiler Error: Ambiguous call to print()

C++ Inheritance
Discuss it

**Question 4 Explanation:**
The print function is not present in class R. So it is looked up in the inheritance hierarchy. print() is present in both classes P and Q, which of them should be called? The idea is, if there is multilevel inheritance, then function is linearly searched up in the inheritance hierarchy until a matching function is found.

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

class Base1
{
public:
    char c;
};

class Base2
{
public:
    int c;
};

class Derived: public Base1, public Base2
{
public:
    void show()  { cout << c; }
};

int main(void)
{
    Derived d;
    d.show();
    return 0;
}
```

Run on I

✓ Compiler Error in "cout << c;"

B Garbage Value

C Compiler Error in "class Derived: public Base1, public Base2"

C++ Inheritance
Discuss it

**Question 13 Explanation:**
The variable 'c' is present in both super classes of Derived. So the access to 'c' is ambiguous. The ambiguity can be removed by using scope resolution operator. [sourcecode language="CPP"] #include using namespace std; class Base1 { public: char c; }; class Base2 { public: int c; }; class Derived: public Base1, public Base2 { public: void show() { cout << Base2::c; } }; int main(void) { Derived d; d.show(); return 0; } [/sourcecode]

```
class Base
{
protected:
    int a;
public:
    Base() {a = 0;}
};

class Derived1:  public Base
{
public:
    int c;
};


class Derived2:  public Base
{
public:
    int c;
};

class DerivedDerived: public Derived1, public Derived2
{
public:
    void show()  {   cout << a;  }
};

int main(void)
{
    DerivedDerived d;
    d.show();
    return 0;
}
```

Run

- ✓ Compiler Error in Line "cout << a;"
- ✗ 0
- C Compiler Error in Line "class DerivedDerived: public Derived1, public Derived2"

C++ Inheritance
Discuss it

**Question 12 Explanation:**
This is a typical example of diamond problem of multiple inheritance. Here the base class member 'a' is inherited through both *Derived1* and *Derived2*. So there are two copies of 'a' in *DerivedDerived* which makes the statement "cout << a;" ambiguous. The solution in C++ is to use virtual base classes. For example, the following program works fine and prints [sourcecode language="CPP"] #include using namespace std; class Base { protected: int a; public: Base() {a = 0;} }; class Derived1: virtual public Base { public: int c; }; class Derived2: virtual public Base { public: int c; }; class DerivedDerived: public Derived1, public Derived2 { public: void show() { cout << a; } }; int main(void) { DerivedDerived d; d.show(); return 0; }

```
class Base
{
public :
    int x, y;
public:
    Base(int i, int j){ x = i; y = j; }
};

class Derived : public Base
{
public:
    Derived(int i, int j):x(i), y(j) {}
    void print() {cout << x <<" "<< y; }
};

int main(void)
{
    Derived q(10, 10);
    q.print();
    return 0;
}
```

Run c

10 10

Compiler Error

0 0

C++ Inheritance

Discuss it

**Question 11 Explanation:**

**The base class members cannot be directly assigned using initializer list.** We should call the base class constructor in order to initialize base class members. Following is error free program and prints "10 10" [sourcecode language="CPP" highlight="15"] #include using namespace std; class Base { public : int x, y; public: Base(int i, int j){ x = i; y = j; } }; class Derived : public Base { public: Derived(int i, int j): Base(i, j) {} void print() {cout << x <<" "<< y; } }; int main(void) { Derived q(10, 10); q.print(); return 0; } [/sourcecode]

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

class Base {};

class Derived: public Base {};

int main()
{
    Base *bp = new Derived;
    Derived *dp = new Base;
}
```

Run on I

A No Compiler Error

X Compiler Error in line "Base *bp = new Derived;"

✓ Compiler Error in line " Derived *dp = new Base;"

D Runtime Error

C++ Inheritance
Discuss it

**Question 6 Explanation:**
A Base class pointer/reference can point/refer to a derived class object, but the other way is not possible.

```
class A
{
public:
     A(){ cout <<"1";}
     A(const A &obj){ cout <<"2";}
};

class B: virtual A
{
public:
    B(){cout <<"3";}
    B(const B & obj){cout<<"4";}
};

class C: virtual A
{
public:
   C(){cout<<"5";}
   C(const C & obj){cout <<"6";}
};

class D:B,C
{
public:
    D(){cout<<"7";}
    D(const D & obj){cout <<"8";}
};

int main()
{
   D d1;
   D d(d1);
}
```

Run on I

Which of the below is not printed? *This question is contributed by Sudheendra Baliga*

A 2

B 4

C 6

✓ All of the above

C++ Inheritance

Discuss it

**Question 14 Explanation:**

Output will be 13571358 as 1357 (for [illegible] or D d(d1)).....reason is that .....[illegible] inheritance we need to explicitly call copy constructor of base class otherwise only d[illegible] base class is called. One more th[illegible]s we are using virtual before base class, there

File Explorer

Which of the following is FALSE about references in C++

A References cannot be NULL

B A reference must be initialized when declared

C Once a reference is created, it cannot be later made to reference another object; it cannot be reset.

✓ References cannot refer to constant value

```
int &fun()
{
    static int x = 10;
    return x;
}
int main()
{
    fun() = 30;
    cout << fun();
    return 0;
}
```

Run on IDE

- ✗ Compiler Error: Function cannot be used as lvalue
- B 10
- ✓ 30

**C++ References**
**Discuss it**

**Question 4 Explanation:**
When a function returns by reference, it can be used as lvalue. Since x is a static variable, it is shared among function calls and the initialization line "static int x = 10;" is executed only once. The function call fun() = 30, modifies x to 30. The next call "cout << fun()" returns the modified value.

▲

```
#include<iostream>
using namespace std;
int main ()
{
       int cin;
       cin >> cin;
       cout << "cin" << cin;
       return 0;
}
```

Thanks to Gokul Kumar for contributing this question.

- ✗ error in using cin keyword
- ✓ cin+junk value
- C cin+input
- D Runtime error

How can we make a C++ class such that objects of it can only be created using new operator? If user tries to create an object directly, the program produces compiler error.

A Not possible

✓ By making destructor private

✗ By making constructor private

D By making both constructor and destructor private

C++ Misc
Discuss it

**Question 5 Explanation:**
See the following example.

```
// Objects of test can only be created using new
class Test
{
private:
    ~Test() {}
friend void destructTest(Test* );
};

// Only this function can destruct objects of Test
void destructTest(Test* ptr)
{
    delete ptr;
}

int main()
{
    // create an object
    Test *ptr = new Test;

    // destruct the object
    destructTest (ptr);

    return 0;
}
```

```
int &fun()
{
    int x = 10;
    return x;
}
int main()
{
    fun() = 30;
    cout << fun();
    return 0;
}
```

Run on IDE

✓ May cause runtime error

B May cause compiler error

✗ Always works fine.

D 0

C++ References
Discuss it

**Question 5 Explanation:**
Since we return reference to a local variable, the memory location becomes invalid after function call is over. Hence it may result in segmentation fault runtime error.

```
class a
{
public :
    ~a()
    {
        cout << "destroy";
    }
};
int main()
{
   vector <a*> *v1  = new vector<a*>;
   vector <a> *v2  = new vector<a>;
   return 0;
}
```

A v1

B v2

X v1 and v2

✓ no destructor call

Which of the following is not correct (in C++) ?

1. Class templates and function templates are instantiated in the same way
2. Class templates differ from function templates in the way they are initiated
3. Class template is initiated by defining an object using the template argument
4. Class templates are generally used for storage classes

X (1)

B (2), (4)

✓ (2), (3), (4)

D (4)

UGC-NET CS 2017 Nov - II  C++ Misc  Class and Object
Discuss it

**Question 12 Explanation:**
In C++ class template and function template are similar in the way the are initiated. Class template are not used for storage class. Class templates and function templates are instantiated in the same way and Class template is not initiated by defining an object using the template. So (2), (3), (4) are not correct in C++. So, option (C) is correct.

How can we restrict dynamic allocation of objects of a class using new?

A By overloading new operator

✗ By making an empty private new operator.

✓ By making an empty private new and new[] operators

D By overloading new operator and new[] operators

C++ Operator Overloading

Discuss it

**Question 1 Explanation:**

If we declare *new* and *[] new* operators, then the objects cannot be created anywhere (within the class and outside the class) See the following example. We can not allocate an object of type Test using new.

```
#include <stdlib.h>
#include <stdio.h>
#include <iostream>

using namespace std;

class Test {
private:
    void* operator new(size_t size) {}
    void* operator new[](size_t size) {}
};

int main()
{
    Test *obj = new Test;
    Test *arr = new Test[10];
    return 0;
}
```

Which of the following operators should be preferred to overload as a global function rather than a member method?

A Postfix ++

B Comparison Operator

✓ Insertion Operator <<

D Prefix++

C++ Operator Overloading
Discuss it

**Question 4 Explanation:**
cout is an object of ostream class which is a compiler defined class. When we do "cout << obj" where obj is an object of our class, the compiler first looks for an operator function in ostream, then it looks for a global function. One way to overload insertion operator is to modify ostream class which may not be a good idea. So we make a global method. Following is an example.

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

class Complex
{
private:
    int real;
    int imag;
public:
    Complex(int r = 0, int i =0)
    {
        real = r;
        imag = i;
    }
    friend ostream & operator << (ostream &out, const Complex &c);
};

ostream & operator << (ostream &out, const Complex &c)
{
    out << c.real;
    out << "+i" << c.imag;
    return out;
}
```

```
#include<iostream>
using namespace std;
class A
{
    int i;
public:
    A(int ii = 0) : i(ii) {}
    void show() {  cout << i << endl;  }
};

class B
{
    int x;
public:
    B(int xx) : x(xx) {}
    operator A() const {  return A(x); }
};

void g(A a)
{
    a.show();
}

int main()
{
    B b(10);
    g(b);
    g(20);
    return 0;
}
```

Run on IDE

✗ Compiler Error

✓
```
10
20
```

C
```
20
20
```

D
```
10
10
```

C++ Operator Overloading
Discuss It

**Question 6 Explanation:**
Note that the class B has as conversion operator overloaded, so an object of B can be converted to that of A. Also, class A has a constructor which can be called with single integer argument, so an int can be converted to A.

new and delete can be global but conversion operator can’t be global

```
class Test {
    int x;
public:
    void* operator new(size_t size);
    void operator delete(void*);
    Test(int i) {
        x = i;
        cout << "Constructor called n";
    }
    ~Test() { cout << "Destructor called n"; }
};

void* Test::operator new(size_t size)
{
    void *storage = malloc(size);
    cout << "new called n";
    return storage;
}

void Test::operator delete(void *p )
{
    cout<<"delete called n";
    free(p);
}

int main()
{
    Test *m = new Test(5);
    delete m;
    return 0;
}
```

Run on IDE

A
```
new called
Constructor called
delete called
Destructor called
```

✓
```
new called
Constructor called
Destructor called
delete called
```

C
```
Constructor called
new called
Destructor called
delete called
```

✗
```
Constructor called
new called
delete called
Destructor called
```

C++ Operator Overloading
Discuss It

**Question 8 Explanation:**
Consider the following statement

```
Test *ptr = new Test;
```

There are actually two things that happen in the above statement–memory allocation and object construction; the **new keyword** is responsible for both. One step in the process is to call **operator new** in order to allocate memory; the other step is to actually invoke the constructor. **Operator new** only allows us to change the memory allocation method, but does not do anything with the constructor calling method. **Keyword new** is responsible for calling the constructor, not **operator new**.

```
class Point {
private:
  int x, y;
public:
  Point() : x(0), y(0) { }
  Point& operator()(int dx, int dy);
  void show() {cout << "x = " << x << ", y = " << y; }
};

Point& Point::operator()(int dx, int dy)
{
    x = dx;
    y = dy;
    return *this;
}

int main()
{
  Point pt;
  pt(3, 2);
  pt.show();
  return 0;
}
```

Run on IDE

✔ x = 3, y = 2

B Compiler Error

C x = 2, y = 3

C++ Operator Overloading
Discuss it

**Question 9 Explanation:**
This a simple example of function call operator overloading. The function call operator, when overloaded, does not modify how functions are called. Rather, it modifies how the operator is to be interpreted when applied to objects of a given type. If you overload a function call operator for a class its declaration will have the following form:

```
return_type operator()(parameter_list)
```

In C++, const qualifier can be applied to 1) Member functions of a class 2) Function arguments 3) To a class data member which is declared as static 4) Reference variables

A Only 1, 2 and 3

B Only 1, 2 and 4

✔ All

D Only 1, 3 and 4

**C++ const keyword**
**Discuss it**

**Question 2 Explanation:**
When a function is declared as const, it cannot modify data members of its class. When we don't want to modify an argument and pass it as reference or pointer, we use const qualifier so that the argument is not accidentally modified in function. Class data members can be declared as both const and static for class wide constants. Reference variables can be const when they refer a const location.

```
class Point
{
    int x, y;
public:
 Point(int i = 0, int j =0)
   { x = i; y = j;  }
   int getX() const { return x; }
   int getY() {return y;}
};

int main()
{
    const Point t;
    cout << t.getX() << " ";
    cout << t.gety();
    return 0;
}
```

A Garbage Values

X 0 0

C Compiler Error in line cout << t.getX() << " ";

✓ Compiler Error in line cout << t.gety();

**C++ const keyword**
**Discuss it**

**Question 3 Explanation:**
There is compiler Error in line cout << t.gety(); A const object can only call const functions.

Which of the following is true about exception handling in C++? 1) There is a standard exception class like Exception class in Java. 2) All exceptions are unchecked in C++, i.e., compiler doesn't check if the exceptions are caught or not. 3) In C++, a function can specify the list of exceptions that it can throw using comma separated list like following.

```
void fun(int a, char b) throw (Exception1, Exception2, ..)
```

A 1 and 3

✓ 1, 2 and 3

C 1 and 2

D 2 and 3

```
template <typename T>
T max(T x, T y)
{
    return (x > y)? x : y;
}
int main()
{
    cout << max(3, 7) << std::endl;
    cout << max(3.0, 7.0) << std::endl;
    cout << max(3, 7.0) << std::endl;
    return 0;
}
```

Ru

A
```
7
7.0
7.0
```

✗ Compiler Error in all cout statements as data type is not specified.

✓ Compiler Error in last cout statement as call to max is ambiguous.

D None of the above

C++ Templates
Discuss it

**Question 3 Explanation:**
The first and second call to max function is a valid call as both the arguments passed are of same data type (i.e int and float respectively). But the third call to max function has arguments of different data type and hence it will generate Compiler Error in last cout statement as call to max is ambiguous. Hence option C is correct

Practice *operator overloading quiz again.*