1. Object Oriented Programming(OOP)

In Object oriented programming we write programs using classes and objects utilizing features of OOPs such as **abstraction**, **encapsulation**, **inheritance** and **polymorphism**

2.Class

A class is like a blueprint/collection of data member and member functions. Class is mere a blueprint or a template. No storage is assigned when we define a class.

It is a user defined data type (ADT).

3. Objects

Objects are instances of class, which holds the data variables declared in class and the member functions work on these class objects.

This is the basic unit of object oriented programming. but when it is instantiated (i.e. an object is created) memory is allocated

4.Abstraction

Abstraction is a process of hiding irrelevant details from user.

Data abstraction refers to, providing only essential information to the outside world and hiding their background details

5.Encapsulation

Encapsulation is a process of combining member data and member functions into a single unit like capsule. This is to avoid the access of private data members from outside the class. **To achieve encapsulation, we make all data members of class private and create public functions**, using them we can get the values from these data members or set the value to these data members.

Encapsulation is placing the data and the member functions that work on that data in the same place.

6.Inheritance

As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.

Inheritance is a feature using which an object of child class acquires the properties of parent class. When one object acquires all the properties and behaviors of parent object

7.Polymorphism

Function overloading and Operator overloading are examples of polymorphism. Polymorphism is a feature using which an object behaves differently in different situation.

In function overloading we can have more than one function with same name but different numbers, type or sequence of arguments.

The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism

8.Overloading

The concept of **overloading is also a branch of polymorphism**. When the exiting operator or function is made to operate on new data type, it is said to be overloaded.

9.class member

A class member can be defined as **public, private or protected.** By default **members would be assumed as private**.

Private keyword, means that no one can access the class members declared **private**, outside that class. If someone tries to access the private members of a class, they will get a **compile time error**. By default class variables and member functions are private.

Public, means all the class members declared under **public** will be available to everyone. The data members and member functions declared public can be accessed by other classes too. Hence there are chances that they might change them.

Protected, is the last access specifier, and it is similar to private, it makes class member inaccessible outside the class. But they can be accessed by any subclass of that class.

10. class constructor & destructor

A class constructor is a special function in a class that is called when a new object of the class is created.

Constructors have no return type, not even `void`.

A destructor is also a special function which is called when created object is deleted.

Objects are initialized using special class functions called **Constructors.**

whenever the object is out of its scope, another special class member function called **Destructor** is called, to release the memory reserved by the object**.**

Destructor get called in reverse sequence in which objects are created.

11. scope resolution : : operator

If the member function is defined inside the class definition it can be defined directly, but if its defined outside the class, then we have to use the scope resolution : : operator along with class name along with function name.

12. Different types of Member functions

0. Simple functions
1. Static functions
2. Const functions
3. Inline functions
4. Friend functions

A function is made static by using `static` keyword with function name. These functions work for the class as whole rather than for a particular object of a class.

These functions cannot access ordinary data members and member functions, but **only `static` data members and `static` member functions can be called inside them.**

It doesn't have any "this" keyword which is the reason it cannot access ordinary members.

Const keyword makes variables constant, that means once defined, there values can't be changed.

When used with member function, such member functions can never modify the object or its related data members.

All the member functions defined inside the class definition are by default declared as Inline.

Inline functions are actual functions, which are copied everywhere during compilation, like preprocessor macro, so the overhead of function calling is reduced. All the functions defined inside class definition are by default inline, but **you can also make any non-class function inline by using keyword inline with them.**

Friend functions:
Friend functions are actually not class member function. Friend functions are made to give **private** access to non-class functions. You can declare a global

function as friend, or a member function of other class as friend.

NOTE:

Friend Functions is a reason, why C++ is not called as a pure Object Oriented language. Because it violates the concept of Encapsulation

13.Function overloading

Function overloading is usually used to enhance the readability of the program. If you have to perform one single operation but with different number or types of arguments, then you can simply overload the function

Different ways to Overload a Function

1. By changing number of Arguments.
2. By having different types of argument.

Function Overloading: Different Number of Arguments

In this type of function overloading we define two functions with same names but different number of parameters of the same type

Function Overloading: Different Datatype of Arguments

In this type of overloading we define two or more functions with same name and same number of parameters, but the type of parameter is different.

Function with Placeholder Arguments

When arguments in a function are declared without any identifier they are called placeholder arguments.

14. Types of Constructors : Constructor Overloading

Default Constructor	Default constructor is the constructor which doesn't take any argument. It has no parameter.

	A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, **the compiler will provide a default constructor implicitly.**
Parametrized Constructor	These are the constructors with parameter. Using this Constructor you can provide different values to data members of different objects, by passing the appropriate values as argument.
Copy Constructors	These are special type of Constructors which takes an object as argument, and is used to copy values of data members of one object into other object

Copy Constructor:
Copy Constructor is a type of constructor which is used to create a copy of an already existing object of a class type. It is usually of the form **X (X&)**, where X is the class name. The compiler provides a default Copy Constructor to all the classes.

15. Static Keyword

Static variable in functions	**Static variables** when used inside function are initialized only once, and then they hold there value even through function calls. These static variables are stored on static storage area , not in stack.
Static Class Objects	Static keyword works in the same way for class objects too. Objects declared static are allocated storage in static storage area, and have scope till the end of program.
Static Data Member in Class	Static data members of class are those members which are shared by all the objects. Static data member has a single piece of storage, and is not available as separate copy with each object, like other non-static data members. Static member variables (data members) are not

	initialied using constructor, because these are not dependent on object initialization.
Static Member Functions	These functions work for the class as whole rather than for a particular object of a class. It can be called using an object and the direct member access `.` operator. But, its more typical to call a static member function by itself, using class name and scope resolution `::` operator. **These functions cannot access ordinary data members and member functions, but only static data members and static member functions.** It doesn't have any "this" keyword which is the reason it cannot access ordinary members

16. `const` Keyword

Defining Class Data members as `const`	These are data variables in class which are defined using `const` keyword. They are not initialized during declaration. Their initialization is done in the constructor. The above way of initializing a class member is known as **Initializer List in C++**.
Defining Class Object as `const`	When an object is declared or created using the `const` keyword, its data members can never be changed, during the object's lifetime.
Defining Class's Member function as `const`	A `const` member function never modifies data members in an object.
`mutable` Keyword	`mutable` keyword is used with member variables of class, which we want to change even if the object is of `const`

	type. Hence, `mutable` data members of a `const` objects can be modified.

17. Inheritance

The class whose properties are inherited by other class is called the **Parent** or **Base** or **Super** class. And, the class which inherits properties of other class is called **Child** or **Derived** or **Sub** class.

Code Reusability
Method Overriding (Hence, Runtime Polymorphism.)
Use of Virtual Keyword

Table showing all the Visibility Modes :

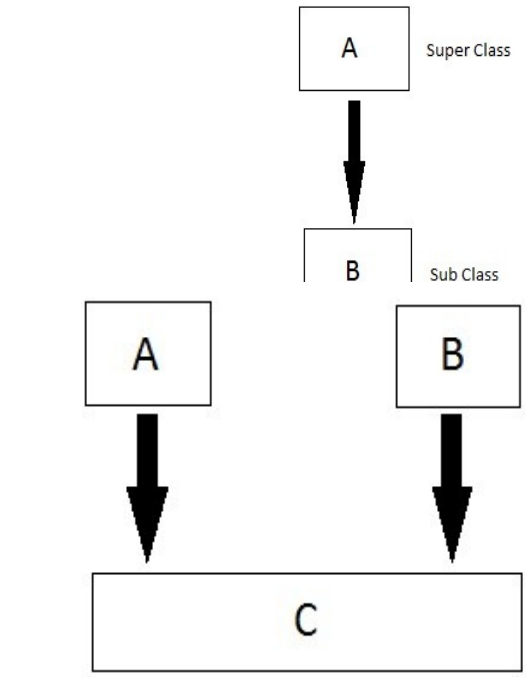
Base class	Derived Class Public Mode	Derived Class Private Mode	Derived Class Protected Mode
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

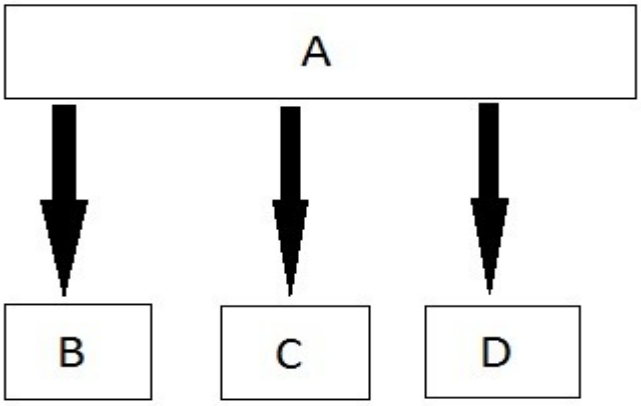
Syntax:

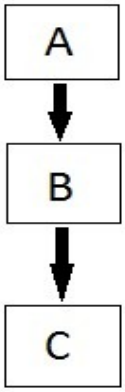
```
class Subclass_name : access_mode Superclass_name
```

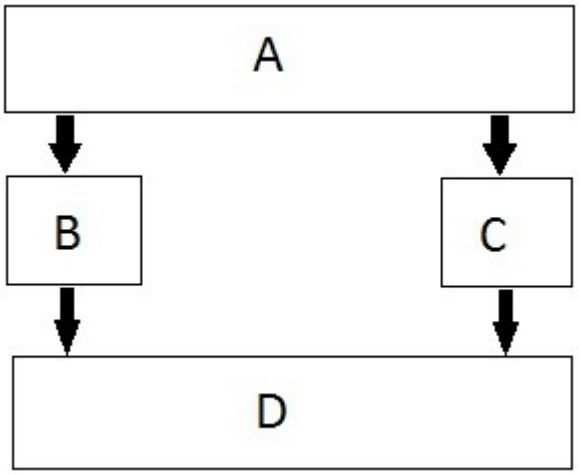
18. Types of Inheritance

1. Single Inheritance
2. Multiple Inheritance
3. Hierarchical Inheritance
4. Multilevel Inheritance
5. Hybrid Inheritance (also known as Virtual Inheritance)


A
Super Class
B
Sub Class
A
B
C


A
B
C
D


A
B
C



19. Hybrid Inheritance and Constructor call

As we all know that whenever a derived class object is instantiated, the base class constructor is always called. But in case of Hybrid Inheritance, as discussed in above example, if we create an instance of class D, then following constructors will be called :

- before class D's constructor, constructors of its super classes will be called, hence constructors of class B, class C and class A will be called.
- when constructors of class B and class C are called, they will again make a call to their super class's constructor.

20. Hybrid Inheritance and Virtual Class

In Multiple Inheritance, the derived class inherits from more than one base class. Hence, in Multiple Inheritance there are a lot chances of ambiguity.

```
class A
{
    void show();
};

class B:public A
{
    // class definition
};

class C:public A
{
```

```
    // class defintion
};

class D:public B, public C
{
    // class definition
};

int main()
{
    D obj;
    obj.show();
}
```

In this case both class B and C inherits function `show()` from class A. Hence class D has two inherited copies of function `show()`. In main() function when we call function `show()`, then ambiguity arises, because compiler doesn't know which `show()` function to call. Hence we use **Virtual** keyword while inheriting class.

```
class B : virtual public A
{
    // class definition
};

class C : virtual public A
{
    // class definition
};

class D : public B, public C
{
    // class definition
};
```

Now by adding virtual keyword, we tell compiler to call any one out of the two `show()` functions.

21.Polymorphism and Function Overriding

Polymorphism

Polymorphism means having multiple forms of one thing. In inheritance, polymorphism is done, by method overriding, when both super and sub class have member function with same declaration bu different definition

Function Overriding

If we inherit a class into the derived class and provide a definition for one of the base class's function again inside the derived class, then that function is said to be **overridden**, and this mechanism is called **Function Overriding**

Requirements for Overriding a Function

1. Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
2. Function that is redefined must have exactly the same declaration in both base and derived class, that means same name, same return type and same parameter list.

Function Call Binding using Base class Pointer

But when we use a Base class's pointer or reference to hold Derived class's object, then Function call Binding gives some unexpected results.

```
class Base
{
    public:
    void show()
    {
        cout << "Base class\n";
    }
};

class Derived:public Base
{
    public:
    void show()
    {
        cout << "Derived Class\n";
    }
}

int main()
{
    Base* b;       //Base class pointer
    Derived d;     //Derived class object
    b = &d;
    b->show();     //Early Binding Occurs
}
```

In the above example, although, the object is of Derived class, still Base class's method is called. This happens due to Early Binding.

22.Virtual Functions

Virtual Function is a function in base class, which is overrided in the derived class, and which tells the compiler to perform **Late Binding** on this function.

`Virtual` Keyword is used to make a member function of the base class Virtual.

23. Late Binding in C++

In Late Binding function call is resolved at runtime. Hence, now compiler determines the type of object at runtime, and then binds the function call. Late Binding is also called **Dynamic** Binding or **Runtime** Binding

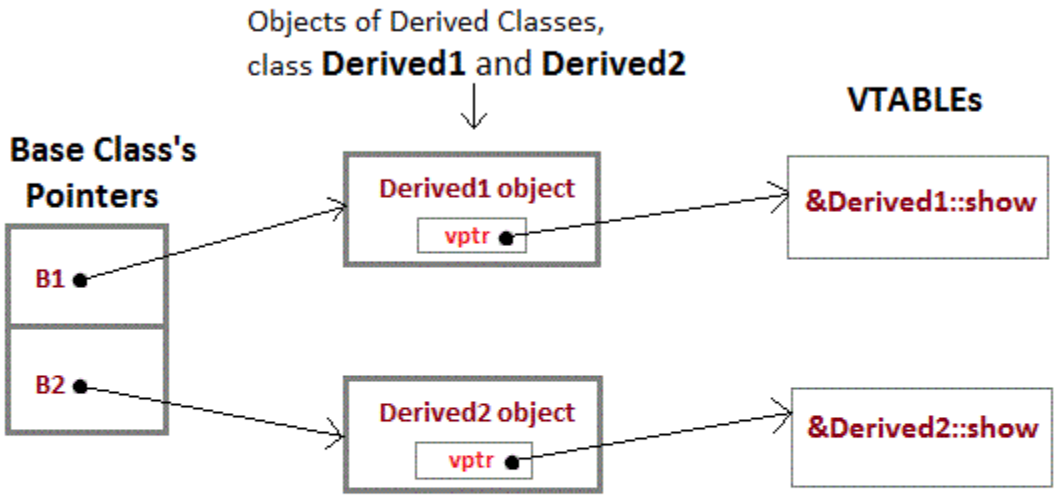
We can make base class's methods virtual by using **virtual** keyword while declaring them. Virtual keyword will lead to Late Binding of that method

```
class Base
{
    public:
    virtual void show()
    {
        cout << "Base class\n";
    }
};

class Derived:public Base
{
    public:
    void show()
    {
        cout << "Derived Class";
    }
}

int main()
{
    Base* b;       //Base class pointer
    Derived d;     //Derived class object
    b = &d;
    b->show();     //Late Binding Occurs
}
```

24.Mechanism of Late Binding in C++



vptr, is the vpointer, which points to the Virtual Function for that object.

VTABLE, is the table containing address of Virtual Functions of each class.

To accomplish late binding, Compiler creates VTABLEs, for each class with virtual function. The address of virtual functions is inserted into these tables. Whenever an object of such class is created the compiler secretly inserts a pointer called vpointer, pointing to VTABLE for that object. Hence when function is called, compiler is able to resolve the call by binding the correct function using the vpointer.

Important Points to Remember

1. Only the Base class Method's declaration needs the **Virtual** Keyword, not the definition.
2. If a function is declared as **virtual** in the base class, it will be virtual in all its derived classes.
3. The address of the virtual Function is placed in the **VTABLE** and the compiler uses **VPTR**(vpointer) to point to the Virtual Function.

25.Pure Virtual Functions in C++

Pure virtual Functions are virtual functions with no definition. They start with **virtual** keyword and ends with = `0`. Here is the syntax for a pure virtual

function,

```
virtual void f() = 0;
```

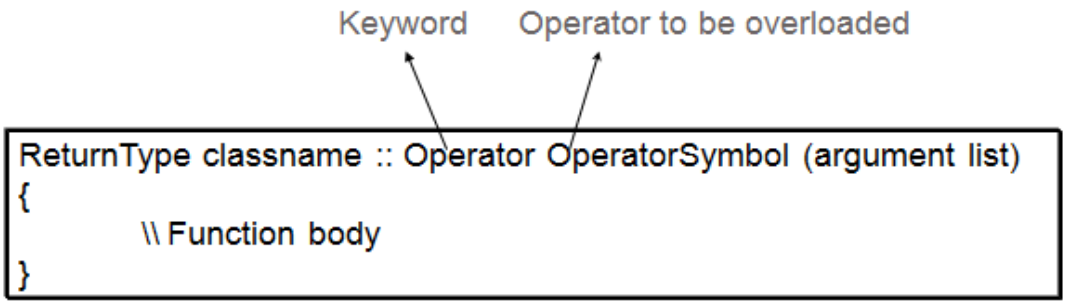
```
Constructors are never Virtual, only Destructors can be Virtual.
```

26.Abstract Class

Abstract Class is a class which contains at-least **one Pure Virtual function in it**. **Abstract classes are used to provide an Interface for its sub classes**.

27.Operator Overloading in C++

Operator overloading is an important concept in C++. It is a type of polymorphism in which an operator is overloaded to give user defined meaning to it. Overloaded operator is used to perform operation on user-defined data type. For example '+' operator can be overloaded to perform addition on various data types, like for Integer, String(concatenation) etc.



28.Implementing Operator Overloading in C++

Operator overloading can be done by implementing a function which can be :

1. Member Function
2. Non-Member Function
3. Friend Function

Friend function:

```
class Time
{
    int hr, min, sec;
    public:
```

```
    // default constructor
    Time()
    {
        hr=0, min=0; sec=0;
    }

    // overloaded constructor
    Time(int h, int m, int s)
    {
        hr=h, min=m; sec=s;
    }

    //overloading '==' operator
    friend bool operator==(Time &t1, Time &t2);
};

bool operator== (Time &t1, Time &t2)
{
    return ( t1.hr == t2.hr && t1.min == t2.min && t1.sec == t2.sec );
}

void main()
{
    Time t1(3,15,45);
    Time t2(4,15,45);
    if(t1 == t2)
    {
        cout << "Both the time values are equal";
    }
    else
    {
        cout << "Both the time values are not equal";
    }
}
```

29.Copy Constructor vs. Assignment Operator (=)

Assignment operator is used to copy the values from one object to another **already existing object**. For example:

```
Time tm(3,15,45); // tm object created and initialized
Time t1;    // t1 object created
t1 = tm;    // initializing t1 using tm
```

Whereas, **Copy constructor** is a special constructor that initializes a **new object** from an existing object.

```
Time tm(3,15,45); // tm object created and initialized
Time t1(tm);    //t1 object created and initialized using tm object
```

30. Exception Handling in C++

In C++, Error handling is done using three keywords:

- try

- catch
- throw

```
try
{
    //code
    throw parameter;
}
catch(exceptionname ex)
{
    //code to handle exception
}

int main()
{
    int x[3] = {-1,2};
    for(int i=0; i<2; i++)
    {
        int ex = x[i];
        try
        {
            if (ex > 0)
                // throwing numeric value as exception
                throw ex;
            else
                // throwing a character as exception
                throw 'ex';
        }
        catch (int ex)  // to catch numeric exceptions
        {
            cout << "Integer exception\n";
        }
        catch (char ex) // to catch character/string exceptions
        {
            cout << "Character exception\n";
        }
    }
}
```

31.Templates

Templates are powerful features of C++ which allows you to write generic programs. In simple terms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger code base for the purpose of code re usability and flexibility of the programs.

The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

Function Template to find the largest number

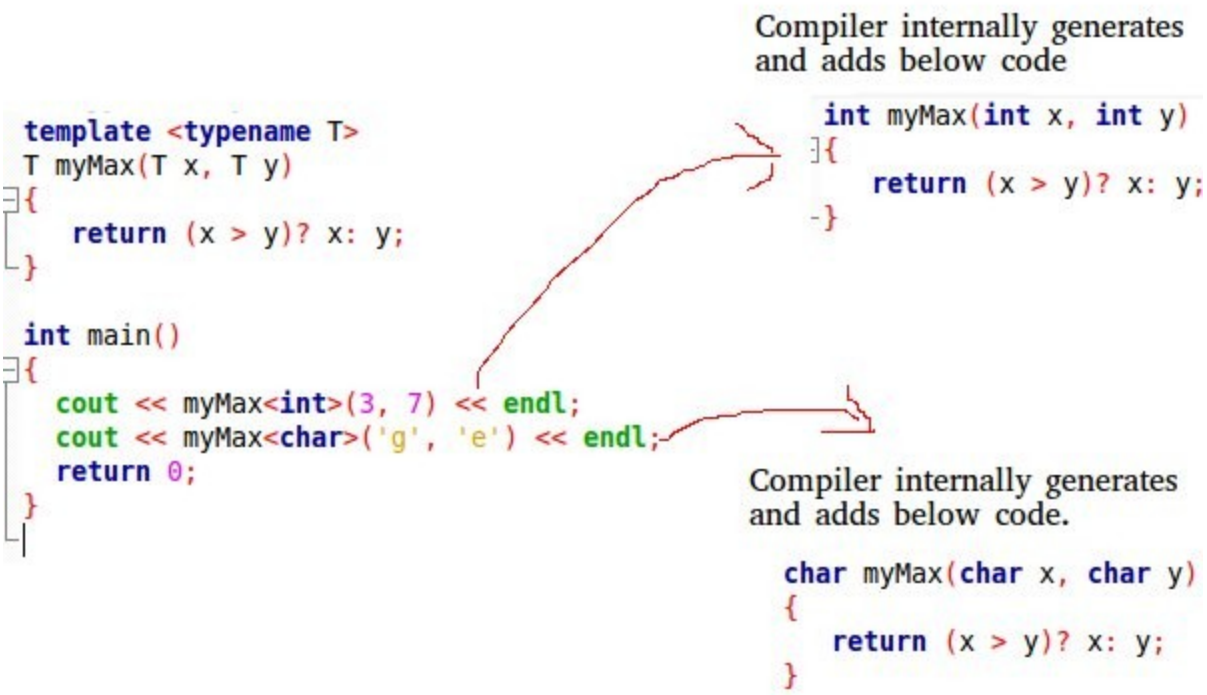
Program to display largest among two numbers using function templates.

```
template <typename T>

T myMax(T x, T y)
{
   return (x > y)? x: y;
}


int main()
{
  cout << myMax<int>(3, 7) << endl;  // Call myMax for int
  cout << myMax<double>(3.0, 7.0) << endl; // call myMax for double
  cout << myMax<char>('g', 'e') << endl;   // call myMax for char

  return 0;
}
```



32. What is a Smart Pointer?
Smart Pointer is used to manage the lifetimes of dynamically allocated objects

33. What is a Dangling Pointer?

A pointer pointing to a memory location of already deleted object is known as a dangling pointer.

34. this pointer
Every object in C++ has access to its own address through an important pointer called **this** pointer. The **this** pointer is an implicit parameter to all member functions. Therefore, inside a member function, this may be used to refer to the invoking object.

The this pointer holds the address of current object, in simple words you can say that this pointer points to the current object of the class.