What is Inheritance?

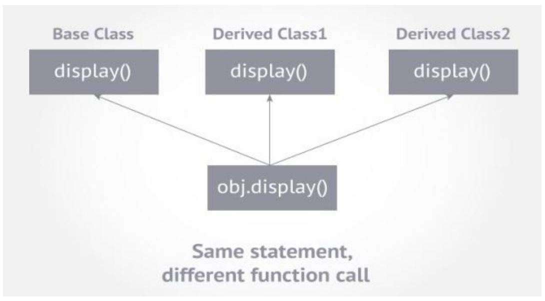
Inheritance in Object Oriented Programming can be described as a process of creating new classes from existing classes. New classes **inherit** some of the properties and behavior of the existing classes. An existing class that is "parent" of a new class is called a base class.

Types of Inheritance

- Single Inheritance.
- Multiple Inheritance.
- Hierarchical Inheritance.
- Multilevel Inheritance.
- Hybrid Inheritance (also known as Virtual Inheritance)

What is Diamond Problem in Inheritance?

The “**diamond problem**” is an ambiguity that can arise as a consequence of allowing **multiple inheritance**. It is a serious **problem** for languages (like C++) that allow for **multiple inheritance** of state. In Java, however, **multiple inheritance** is not allowed for classes, only for interfaces, and these do not contain state.



What is Virtual Function?

A **virtual function** a member **function** which is declared within base class and is re-defined (Overridden) by derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a **virtual function** for that object and execute the derived class's version of the **function**.

Virtual Functions. **Virtual** Function is a function in base class, which is overridden 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**.

What is Pure Virtual Function?

A **virtual function** which is initialized with zero called **Pure Virtual Function**.

A virtual function whose declaration ends with `=0` is called a pure virtual function. For example,

```
class Base
{
    public:
      virtual void disp() = 0;
};
```

What is an Abstract Class?

An **abstract class** is a **class** that is designed to be specifically used as a base**class**. An **abstract class** contains at least one pure virtual function. You declare a pure virtual function by using a pure specifier (= 0) in the declaration of a virtual member function in the **class** declaration.

```
class Abstract
{
    public:
      virtual void disp() = 0;
};
```

What is Function Overloading?

Function Overloading in C++ You can have multiple definitions for the same **function name** in the same scope. The definition of the function must differ from each other by the types and/or the number of arguments in the argument list. You cannot overload function declarations that differ only by **return type**.

Function overloading is a feature in C++ where two or more functions can have the same name but different parameters.
Function overloading can be considered as an example of polymorphism feature in C++.

Following is a simple C++ example to demonstrate function overloading.

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

void print(int i) {
  cout << " Here is int " << i << endl;
}
void print(double f) {
  cout << " Here is float " << f << endl;
}
void print(char* c) {
  cout << " Here is char* " << c << endl;
}

int main() {
  print(10);
  print(10.10);
  print("ten");
  return 0;
}
```

What is Function Overriding?

Overloading a method (or function) in C++ is the ability for functions of the same name to be defined as long as these methods have different signatures (different set of parameters). **Overriding** means having two methods with the same signature and parameters. One of the methods is **in the** parent class and the other is **in the** child class.

```
Class a

{

public:

      virtual void display(){ cout << "hello"; }

}


Class b:public a

{

public:

       void display(){ cout << "bye";};

}
```

What is the interface of a class C++?

The **C++ interfaces** are implemented using abstract **classes** and these abstract **classes** should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data. A **class** is made abstract by declaring at least one of its functions as pure virtual function.

Example:

```
#include <iostream>

using namespace std;

// Base class

class Shape {

   public:

      // pure virtual function providing interface framework.

      virtual int getArea() = 0;

      void setWidth(int w) {
         width = w;

      }

      void setHeight(int h) {

         height = h;

      }

   protected:

      int width;

      int height;

};

// Derived classes

class Rectangle: public Shape {

   public:

      int getArea() {

         return (width * height);

      }

};

class Triangle: public Shape {

   public:

      int getArea() {

         return (width * height)/2;

      }

};

int main(void) {

   Rectangle Rect;

   Triangle  Tri;

   Rect.setWidth(5);

   Rect.setHeight(7);

   // Print the area of the object.

   cout << "Total Rectangle area: " << Rect.getArea() << endl;

   Tri.setWidth(5);
   Tri.setHeight(7);

   // Print the area of the object.

   cout << "Total Triangle area: " << Tri.getArea() << endl;

   return 0;

}
```

Output:

```
Total Rectangle area: 35
Total Triangle area: 17
```

What is Exception Handling?

A **C++ exception** is a response to an **exceptional** circumstance that arises while a program is running, such as an attempt to divide by zero. **Exceptions** provide a way to transfer control from one part of a program to another. **C++ exception handling** is built upon three keywords: try, catch, and throw.

Example:

```
#include <iostream>

using namespace std;

double division(int a, int b) {

   if( b == 0 ) {

      throw "Division by zero condition!";

   }

   return (a/b);

}

int main () {

   int x = 50;

   int y = 0;

   double z = 0;

   try {

      z = division(x, y);

      cout << z << endl;
```

```
    } catch (const char* msg) {
      cerr << msg << endl;
    }
    return 0;
}
```

What is Final Keyword in C++?

Once a function is **final**, it can never be declared again in any derived class. The**final keyword** allows you to declare a virtual method, override it N times, and then mandate that 'this can no longer be overridden'. ... **Final keyword in C++** when added to a function, prevents it from being overridden by a base class

final is not a keyword; it is a specifier. It has a special meaning (defined by the standard) only when it is applied to a class (appears immediately after the name of the class) or a virtual member function (appears immediately after the declarator of the function).

Example:

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

class Base
{
public:
    virtual void myfun() final
    {
        cout << "myfun() in Base";
    }
};
class Derived : public Base
{
    void myfun()
    {
        cout << "myfun() in Derived\n";
    }
};

int main()
{
    Derived d;
    Base &b = d;
    b.myfun();
    return 0;
}
```

}

What is Aggregation and Composition?

This kind of relationship is termed as association. **Aggregation and Composition** are subsets of association meaning they are specific cases of association. In both **aggregation and composition** object of one class "owns" object of another class. But there is a subtle difference.

In Object-Oriented programming, an Object communicates to other Object to use functionality and services provided by that object. **Composition** and **Aggregation** are the two forms of association.

What is Run-Time Polymorphism?

In **c++ runtime polymorphism** is function overriding that means same function name and same signature . In other words in Base class method Display(int, int) and derived class same method Display(int, int) .to achive both method at **runtime** using super keyword its called **runtime polymorphism**.

Example:

```
class Base
{
.....
.....
.....
public:
virtual void display();

}


class Derived
{
..............
.........
public:
void display();


}

int main()
{
Base*ptr=new Derived;
ptr->display();

return 0;
}
```

What is File Handling in C++?

These classes are derived directly or indirectly from the classes istream and ostream. We have already used objects whose types were these classes: cin is an object of class istream and cout is an object of class ostream. Therefore, we have already been using classes that are related to our file streams. And in fact, we can use our file streams the same way we are already used to
use cin and cout, with the only difference that we have to associate these streams with physical files. Let's see an example:

```cpp
// basic file operations
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  ofstream myfile;
  myfile.open ("example.txt");
  myfile << "Writing this to a file.\n";
  myfile.close();
  return 0;
}
```