About the content

This C++ material taken from the following websites:

- https://www.tutorialspoint.com/cplusplus
- https://www.w3schools.com/CPP
- https://www.geeksforgeeks.org/cpp-tutorial
- https://www.javatpoint.com/cpp-tutorial
- https://www.programiz.com/cpp-programming

UNIT-III
Module-1
Static and Const members of class

1. Static keyword in C++

Static Data Members of class

We can define class members static using **static** keyword. When we declare a member of a class as static it means no matter how many objects of the class are created, there is only one copy of the static member.
A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator **::** to identify which class it belongs to.

Syntax

static data_type data_member;

Here, the **static** is a keyword of the predefined library.
The **data_type** is the variable type in C++, such as int, float, string, etc.
The **data_member** is the name of the static data.

Let us try the following example to understand the concept of static data members −

```
#include <iostream>

using namespace std;

class Box {
  public:
    static int objectCount;

    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
      cout <<"Constructor called." << endl;
```

```
      length = l;
      breadth = b;
      height = h;

      // Increase every time object is created
      objectCount++;
    }
    double Volume() {
      return length * breadth * height;
    }

  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void) {
  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2

  // Print total number of objects.
  cout << "Total objects using class name: " << Box::objectCount << endl;

// Print total number of objects using objects.
  cout << "Total objects using Box1: " << Box1.objectCount << endl;
  cout << "Total objects using Box2: " << Box1.objectCount << endl;

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Constructor called.
Constructor called.
Total objects using class name: 2
Total objects using Box1: 2
Total objects using Box2: 2
```

Static Member Function of a class

By declaring a function member as static, you make it independent of any particular object of the class. A static member function can be called even if no objects of the class exist and the **static** functions are accessed using only the class name and the scope resolution operator **::**.

A static member function can only access static data member, other static member functions and any other functions from outside the class.

Static member functions have a class scope and they do not have access to the **this** pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not.

Example 2: Let's create another program to access the static member function using the class name and object name in the C++ programming language.

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

```
class Note
{
// declare a static data member
static int num;

public:
// create static member function
static int func ()
{
return num;
}
};
// initialize the static data member using the class name and the scope resolution operator
int Note :: num = 5;

int main ()
{
Note n;
// access static member function using the class name and the scope resolution
cout << " The value of the num is: " << Note:: func () << endl;

// access static member function using the object
cout << " The value of the num is:  " << n.func () << endl;

return 0;
}
```

Output

```
The value of the num is: 5
The value of the num is: 5
```

Let us try the following example to understand the concept of static function members −

```
#include <iostream>

using namespace std;

class Box {
  public:
    static int  objectCount;

    // Constructor definition
    Box(double l = 2.0, double b = 2.0, double h = 2.0) {
      cout <<"Constructor called." << endl;
      length = l;
      breadth = b;
      height = h;

      // Increase every time object is created
      objectCount++;
    }
    double Volume() {
      return length * breadth * height;
    }
    static int getCount() {
      return objectCount;
    }
```

```
  private:
    double length;     // Length of a box
    double breadth;    // Breadth of a box
    double height;     // Height of a box
};

// Initialize static member of class Box
int Box::objectCount = 0;

int main(void) {
  // Print total number of objects before creating object.
  cout << "Inital Stage Count: " << Box::getCount() << endl;

  Box Box1(3.3, 1.2, 1.5);    // Declare box1
  Box Box2(8.5, 6.0, 2.0);    // Declare box2

  // Print total number of objects after creating objects using class name.
  cout << "Final Stage Count using class name: " << Box::getCount() << endl;

// Print total number of objects after creating objects using object name.
  cout << "Final Stage Count using Box1 object: " << Box1.getCount() << endl;

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count using class name: 2
Final Stage Count using Box1 object: 2
```

2. Const keyword in C++

This section will discuss the const keyword in the C++ programming language. It is the const keywords used to define the constant value that cannot change during program execution. It means once we declare a variable as the constant in a program, the variable's value will be fixed and never be changed. If we try to change the value of the const type variable, it shows an error message in the program.

Use const keywords in different parameters:

- Use const variable
- Use const with pointers
- Use const pointer with variables
- Use const with function arguments
- Use const with class member functions
- Use const with class data members
- Use const with class objects

In this module we will discuss about using **const** keyword with class **data members** and **member functions** only

Refer the below link to know how const keyword can be used in C++
https://www.javatpoint.com/const-keyword-in-cpp

Constant Data Members of class

Data members are like the variable that is declared inside a class, but once the data member is initialized, it never changes, not even in the constructor or destructor. The constant data member is initialized using the const keyword before the data type inside the class. The const data members cannot be assigned the values during its declaration; however, they can assign the constructor values.

Example: Program to use the const keyword with the Data members of the class

```
/* create a program to demonstrate the data member using the const keyword in C++. */
#include <iostream>
using namespace std;
// create a class ABC
class ABC
{

public:
 // use const keyword to declare const data member
const int A;
// create class constructor
ABC ( int y) : A(y)
{
cout << " The value of y: " << y << endl;
}
};
int main ()
{
ABC obj( 10); // here 'obj' is the object of class ABC
cout << " The value of constant data member 'A' is: " << obj.A << endl;
// obj.A = 5; // it shows an error.
// cout << obj.A << endl;
return 0;
}
```

Output

```
The value of y: 10
The value of constant data member 'A' is: 10
```

In the above program, the object obj of the ABC class invokes the ABC constructor to print the value of y, and then it prints the value of the const data member 'A' is 10. But when the 'obj.A' assigns a

new value of 'A' data member, it shows a compile-time error because A's value is constant and cannot be changed.

Const Member function of class

1. A const is a constant member function of a class that never changes any class data members, and it also does not call any non-const function.
2. It is also known as the read-only function.
3. We can create a constant member function of a class by adding the const keyword after the name of the member function.

Syntax

```
return_type mem_fun() const
{
}
```

In the above syntax, mem_fun() is a member function of a class, and the const keyword is used after the name of the member function to make it constant.

Example1: Program to use the const keyword with the member function of class

```
#include <iostream>
using namespace std;
// create a class ABC
class ABC
{
// define the access specifier
public:

// declare int variables
int A;
// declare member function as constant using const keyword
void fun () const
{
 A = 0; // it shows compile time error
}
};

int main ()
{
   ABC obj;
   obj.fun();
   return 0;
}
```

Output

The above code throws a compilation error because the fun() function is a const member function of class ABC, and we are trying to assign a value to its data member 'x' that returns an error.

Example2: Program to use the const keyword with the member function of class. adding two number using const member function.

```
#include <iostream>
using namespace std;
// create a class Test

class Test
{
        // declare int variables
        int a,b;

        // define the access specifier
        public:

        // normal member function to store values of a and b
        void set(int x, int y)
        {
                a =x;
                b=y;
        }

        // declare member function as constant using const keyword
        void disp() const
        {
                 cout << "the sum is " << a+b<<endl; //it shows the sum of a and b
        }
};

int main ()
{
   Test obj;
   obj.set(10,20);
   obj.disp();
   return 0;
}
```

Output:

the sum is 30

Module-2

Inheritance

The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important features of Object-Oriented Programming. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class.

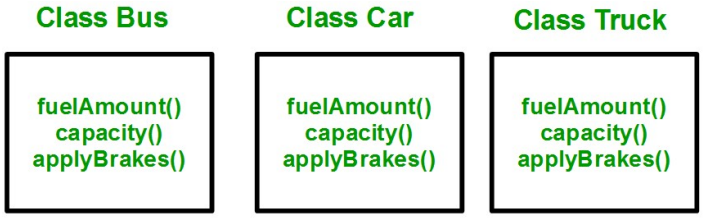
Inheritance is a feature or a process in which, new classes are created from the existing classes. The new class created is called “derived class” or “child class” or “sub class” and the existing class is known as the “base class” or “parent class” or “super class”. The derived class now is said to be inherited from the base class.

When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own. These new features in the derived class will not affect the base class. The derived class is the specialized class for the base class.

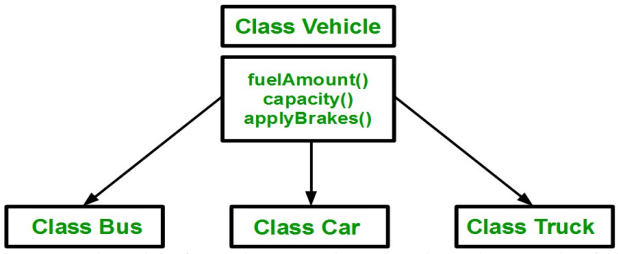
- **Sub Class:** The class that inherits properties from another class is called Subclass or Derived Class.
- **Super Class:** The class whose properties are inherited by a subclass is called Base Class or Superclass.

Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car, and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be the same for all three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown below figure:



You can clearly see that the above process results in duplication of the same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited the rest of the three classes from the base class (Vehicle).

Implementing inheritance in C++: For creating a sub-class that is inherited from the base class we have to follow the below syntax.

Derived Classes: A Derived class is defined as the class derived from the base class.

Syntax:

```
class  <derived_class_name> : <access-specifier> <base_class_name>
{
      //body
}
```

here

class — keyword to create a new class

derived_class_name — name of the new class, which will inherit the base class
access-specifier — either of private, public or protected. If neither is specified, PRIVATE is taken as default
base-class-name — name of the base class

Note: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

Consider a base class **Shape** and its derived class **Rectangle** as follows −

```
#include <iostream>

using namespace std;

// Base class
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};

// Derived class
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height);
    }
};

int main(void) {
  Rectangle Rect;

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

  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Total area: 35
```

Modes of Inheritance:

There are 3 modes of inheritance.

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.
We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied −

- **Public Inheritance** − When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
- **Protected Inheritance** − When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
- **Private Inheritance** − When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

Note: The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C, and D all contain the variables x, y, and z in the below example. It is just a question of access.

```
/* C++ Implementation to show that a derived class doesn't inherit access to private data members.
However, it does inherit a full parent object.*/
class A {
public:
   int x;

protected:
   int y;

private:
   int z;
};

class B : public A {
   // x is public
   // y is protected
   // z is not accessible from B
};

class C : protected A {
   // x is protected
   // y is protected
   // z is not accessible from C
};

class D : private A // 'private' is default for classes
{
   // x is private
   // y is private
   // z is not accessible from D
```

};

The below table summarizes the above three modes and shows the access specifier of the members of the base class in the subclass when derived in public, protected and private modes:

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)

Types Of Inheritance:-

1. Single inheritance
2. Multilevel inheritance
3. Multiple inheritance
4. Hierarchical inheritance
5. Hybrid inheritance

1. Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.

Syntax:

```
class A
{
... .. ...
};

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

```
// C++ program to explain
// Single inheritance
#include<iostream>
using namespace std;

// base class
class Vehicle {
  public:
    Vehicle()
    {
      cout << "This is a Vehicle\n";
```

```
    }
};

// sub class derived from a single base classes
class Car : public Vehicle {
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}
```

Output

This is a Vehicle

Example2:

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

class A
{
   protected:
   int a;

   public:
      void set_A()
      {
         cout<<"Enter the Value of A=";
         cin>>a;
      }
      void disp_A()
      {
         cout<<endl<<"Value of A="<<a;
      }
};

class B: public A
{
   int b,p;

   public:
      void set_B()
      {
         set_A();
         cout<<"Enter the Value of B=";
         cin>>b;
      }

      void disp_B()
```

```
    {
       disp_A();
       cout<<endl<<"Value of B="<<b;
    }

    void cal_product()
    {
       p=a*b;
       cout<<endl<<"Product of "<<a<<" * "<<b<<" = "<<p;
    }

};

main()
{

   B b;
   b.set_B();
   b.cal_product();

   return 0;

}
```

Output:-

```
Enter the Value of A=3
Enter the Value of B=8
Product of 3 * 8 = 24
```

Example3:

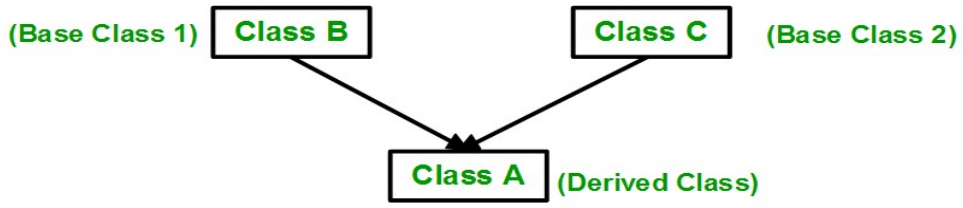
```
#include<iostream>
using namespace std;

class A
{
   protected:
   int a;

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

      void disp_A()
      {
         cout<<endl<<"Value of A="<<a;
      }
};

class B: public A
{
   int b,p;
```

```
    public:
        void set_B(int x,int y)
        {
            set_A(x);
            b=y;
        }

        void disp_B()
        {
            disp_A();
            cout<<endl<<"Value of B="<<b;
        }

        void cal_product()
        {
            p=a*b;
            cout<<endl<<"Product of "<<a<<" * "<<b<<" = "<<p;
        }

};

main()
{
    B  b;
    b.set_B(4,5);
    b. disp_B();
    b.cal_product();

    return 0;
}
```

Output

```
Value of A=4
Value of B=5
Product of 4 * 5 = 20
```

2. Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one **subclass** is inherited from more than one **base class**.



Syntax:

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

```
{
... ... ...
};
```

Here, the number of base classes will be separated by a comma (‘, ‘) and the access mode for every base class must be specified.

```
// C++ program to explain
// multiple inheritance
#include <iostream>
using namespace std;

// first base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};

// second base class
class FourWheeler {
public:
   FourWheeler()
   {
      cout << "This is a 4 wheeler Vehicle\n";
   }
};

// sub class derived from two base classes
class Car : public Vehicle, public FourWheeler {
};

// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base classes.
   Car obj;
   return 0;
}
```

Output

```
This is a Vehicle
This is a 4 wheeler Vehicle
```

EXAMPLE2:

```
#include <iostream>
#include <string>
using namespace std;
 // Base class
class Author {
   int a_id;
   string a_name;
  public:
   void a_set(int n, string s) {
     a_id=n;
     a_name=s;
   }
```

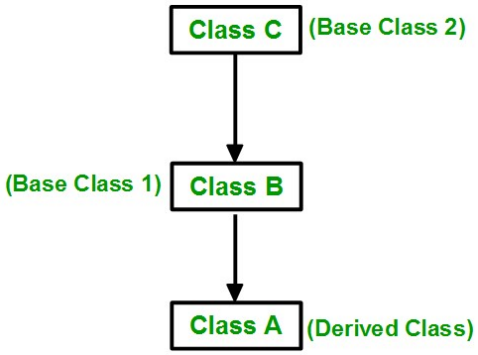
```
    void a_disp(){
      cout<<"author id = "<<a_id<<endl;
      cout<<"author name = "<<a_name<<endl;
    }
};

// Another base class
class Publisher
{
    int p_id;
    string p_name;
  public:
    void p_set(int n, string s) {
      p_id=n;
      p_name=s;
    }
     void p_disp(){
      cout<<"publisher id = "<<p_id<<endl;
      cout<<"publisher name = "<<p_name<<endl;
    }
};

// Derived class
class Book: public Author, public Publisher {
    int b_id;
    string b_name;
  public:
    void b_set(int n, string s) {
      b_id=n;
      b_name=s;
    }
     void b_disp(){
      cout<<"book id = "<<b_id<<endl;
      cout<<"bookr name = "<<b_name<<endl;
    }
};

int main() {
  Book bk;
  bk.a_set(10,"saidavali");
  bk.p_set(200,"RGUKTN");
  bk.b_set(4,"OOPS WITH CPP");
  bk.a_disp();
  bk.p_disp();
  bk.b_disp();
  return 0;
}
```

Output:

```
author id = 10
```

```
author name = saidavali
publisher id = 200
publisher name = RGUKTN
book id = 4
bookr name = OOPS WITH CPP
```

3. Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.



Syntax:-

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

```
// C++ program to implement
// Multilevel Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};

// first sub_class derived from class vehicle
class fourWheeler : public Vehicle {
public:
   fourWheeler()
   {
      cout << "Objects with 4 wheels are vehicles\n";
   }
};
```

```
// sub class derived from the derived base class fourWheeler
class Car : public fourWheeler {
public:
   Car() { cout << "Car has 4 Wheels\n"; }
};

// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base classes.
   Car obj;
   return 0;
}
```

Output

```
This is a Vehicle
Objects with 4 wheels are vehicles
Car has 4 Wheels
```

Example2:

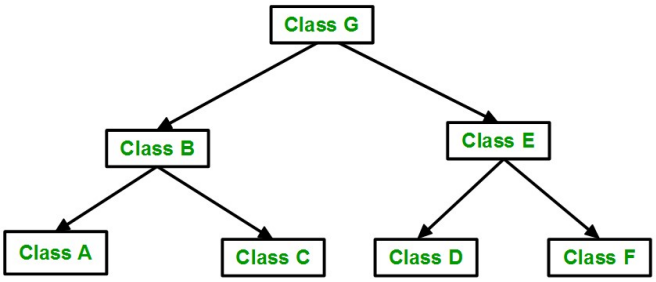
```
#include<iostream>
using namespace std;

class A
{
        protected:
        int a;

        public:
          void set_A()
          {
              cout<<"Enter the Value of A=";
              cin>>a;

          }

          void disp_A()
          {
              cout<<endl<<"Value of A="<<a;
          }
};

class B: public A
{
      protected:
         int b;

      public:
           void set_B()
         {
             cout<<"Enter the Value of B=";
              cin>>b;
```

```
        }


      void disp_B()
       {
         cout<<endl<<"Value of B="<<b;
       }
};

class C: public B
{
    int c,p;

    public:
       void set_C()
       {
            cout<<"Enter the Value of C=";
            cin>>c;
       }

        void disp_C()
        {
            cout<<endl<<"Value of C="<<c;
        }

         void cal_product()
         {
              p=a*b*c;
             cout<<endl<<"Product of "<<a<<" * "<<b<<" * "<<c<<" = "<<p;
          }
};

main()
{

   C  c;
   c.set_A();
   c.set_B();
   c.set_C();
   c.disp_A();
   c.disp_B();
   c.disp_C();
   c.cal_product();

   return 0;
}
```

OUTPUT:

Enter the Value of A=4
Enter the Value of B=6
Enter the Value of C=7
Value of A=4
Value of B=6
Value of C=7
Product of 4 * 6 * 7 = 168

4. Hierarchical Inheritance: In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.



Syntax:-

```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

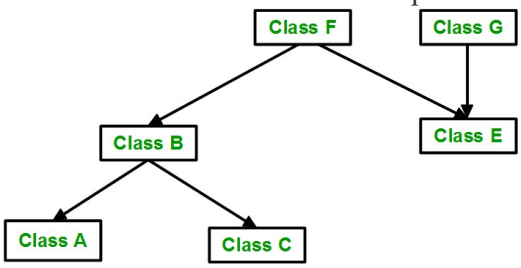
```
// C++ program to implement
// Hierarchical Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {
};

// second sub class
class Bus : public Vehicle {
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base class.
    Car obj1;
    Bus obj2;
```

```
    return 0;
}
```

Output

This is a Vehicle
This is a Vehicle

5. Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.
Below image shows the combination of hierarchical and multiple inheritances:




```
// C++ program for Hybrid Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// base class
class Fare {
public:
    Fare() { cout << "Fare of Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {
};

// second sub class
class Bus : public Vehicle, public Fare {
};

// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base class.
    Bus obj2;
    return 0;
}
```

Output

This is a Vehicle

Inheritance Generalization and Specialization

Generalization:

The process of extracting common characteristics from two or more classes and combining them into a generalized superclass is called Generalization. The common characteristics can be attributes or methods.

The Generalization is the process that does the grouping entities into broader or distributed categories based on certain common attributes. All the common attributes bind together to form a higher-level component or element is called a generalized entity. Generalization is represented by a triangle followed by a line.

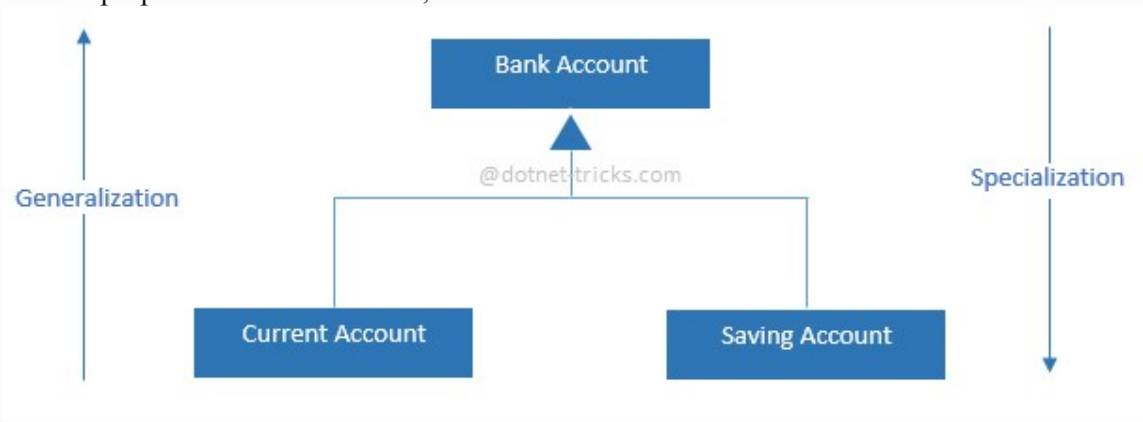
Specialization:

Specialization is the reverse process of Generalization means creating new subclasses from an existing class.

Specialization is the process of dividing a parent-level entity into narrower categories accordingly to all the possible child categories. By having the behavior of the opposite of the generalization process, specialization requires the separation of entities based on certain uncommon attributes.

Specialization is quite useful in a situation where you block out the unnecessary data so you can locate or identify the specific information whenever it requires. specialized kinds of entities can be specialized further due to the higher chance of further modularity.

Let's take an example of a Bank Account; A Bank Account is of two types – A current Account and Saving Account. Current Account and Saving Account inherits the common/ generalized properties like Account Number, Account Balance, etc. from a Bank Account and also have their own specialized properties like interest rate, etc.



The difference between the Generalization vs Specialization

1. Generalization works in a bottom-up manner while the specialized works complete ever than the generalization where it follows the top-down approach
2. The entity of the higher-level must always have the lower-level entity in Generalization whereas, in Specialization, the entities of the top-level may not consist of the entities of the lower level periodically

3. Generalization can be defined as a process where the grouping is created from multiple entity sets, and the Specialization takes a sub-set of the higher level entity and it formulates a lower-level entity set
4. Generalization helps to reduce the schema of the data by unifying the multiple components. and in Specialization, it expands the schema by multiplying the multiple components
5. Generalization leads to a reduction in the overall size of a schema and focuses on the specialization, on the other side, Specialization increase the size of schemas

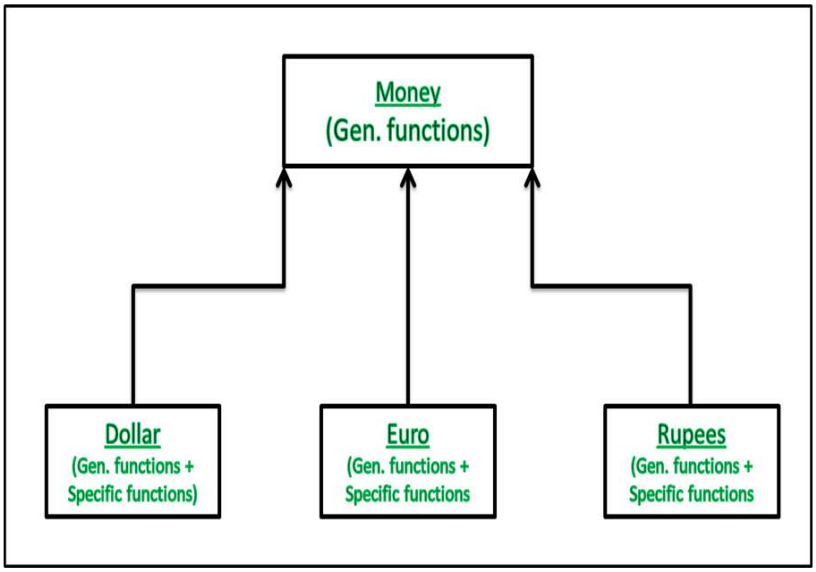
General class?

Loosely speaking, a class which tells the main features but not the specific details. The classes situated at the top of the inheritance hierarchy can be said as General.

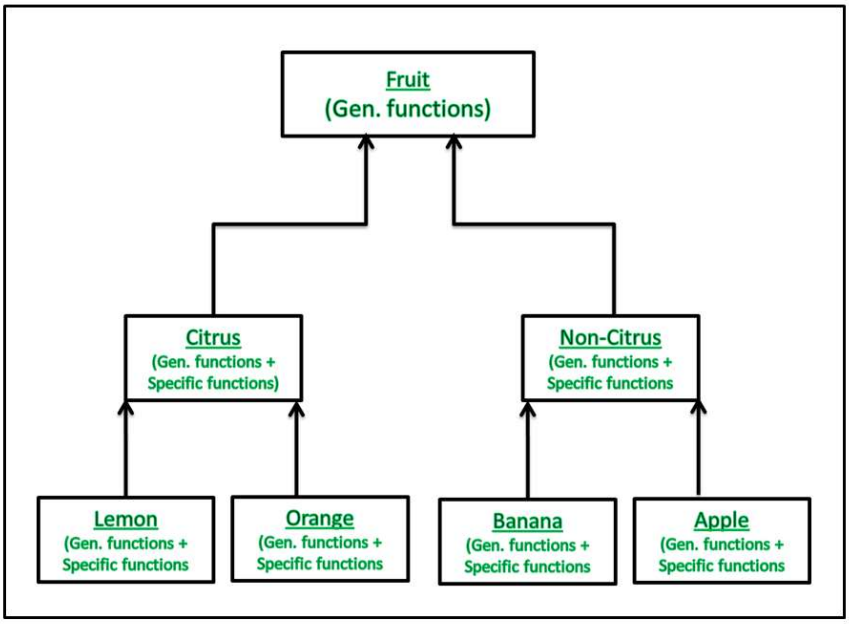
Specific class?

A class which is very particular and states the specific details. The classes situated at the bottom of the inheritance hierarchy can be said as Specific.

Example 1:



Relatively General Class: Money
Relatively Specific Class: Dollar, Euro, Rupees
Example 2:



Lemon, Orange are more Specific than Citrus
Banana, Apple are more Specific than Non-Citrus
Citrus, Non-Citrus are more Specific than Fruit
Fruit is most general class

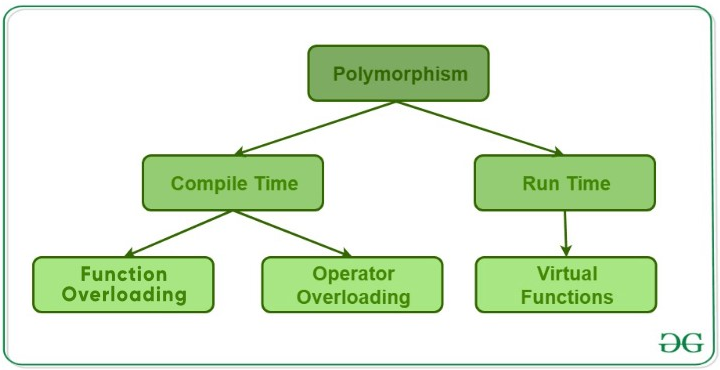
Module-3

Polymorphism in C++

The word "polymorphism" means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism is a person who at the same time can have different characteristics. Like a man at the same time is a father, a husband and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object-Oriented Programming.

In C++, polymorphism is mainly divided into two types:

1. Compile-time Polymorphism
2. Runtime Polymorphism



Types of Polymorphism

1. Compile-time polymorphism

This type of polymorphism is achieved by function overloading or operator overloading.

1. **Function Overloading**: When there are multiple functions with the same name but different parameters, then the functions are said to be **overloaded**. Functions can be overloaded by **changing the number of arguments** or/and **changing the type of arguments**.

```
// C++ program for function overloading
#include < iostream >

using namespace std;
class Geeks
{
   public:

   // function with 1 int parameter
   void func(int x)
   {
      cout << "value of x is " << x << endl;
   }

   // function with same name but 1 double parameter
   void func(double x)
   {
      cout << "value of x is " << x << endl;
   }

   // function with same name and 2 int parameters
   void func(int x, int y)
   {
      cout << "value of x and y is " << x << ", " << y << endl;
   }
};

int main() {

   Geeks obj1;

   // Which function is called will depend on the parameters passed
   // The first 'func' is called
   obj1.func(7);

   // The second 'func' is called
   obj1.func(9.132);

   // The third 'func' is called
   obj1.func(85,64);
   return 0;
}
```

Output:

```
value of x is 7
value of x is 9.132
value of x and y is 85, 64
```

In the above example, a single function named *func* acts differently in three different situations, which is a property of polymorphism.

2. Operator Overloading:

C++ also provides the option to overload operators. For example, we can make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands. So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them. **Example**:

```
// CPP program to illustrate
// Operator Overloading
#include<iostream>
using namespace std;

class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i =0)  {real = r;   imag = i;}

    // This is automatically called when '+' is used with
    // between two Complex objects
    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }
    void print() { cout << real << " + i" << imag << endl; }
};

int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2; // An example call to "operator+"
    c3.print();
}
```

Output:

```
12 + i9
```

In the above example, the operator '+' is overloaded. Usually, this operator is used to add two numbers (integers or floating point numbers), but here the operator is made to perform the addition of two imaginary or complex numbers.

2. **Runtime polymorphism**:

This type of polymorphism is achieved by Function Overriding.

Function overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

```
// C++ program for function overriding

#include <iostream>
using namespace std;

class base
{
public:
   virtual void print ()
   { cout<< "print base class" <<endl; }

   void show ()
   { cout<< "show base class" <<endl; }
};

class derived:public base
{
public:
   void print () //print () is already virtual function in derived class, we could also declared as
virtual void print () explicitly
   { cout<< "print derived class" <<endl; }

   void show ()
   { cout<< "show derived class" <<endl; }
};

//main function
int main()
{
   base *bptr;
   derived d;
   bptr = &d;

   //virtual function, binded at runtime (Runtime polymorphism)
   bptr->print();

   // Non-virtual function, binded at compile time
   bptr->show();

   return 0;
```

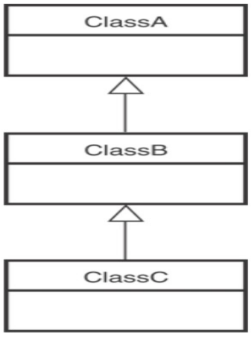
```
}
```

Output:

```
print derived class
show base class
```

Virtual Functions and Runtime Polymorphism in C++

A **virtual function** is a *member function* that is declared in the *base* class using the keyword **virtual** and is re-defined (Overriden) in the *derived* class. It tells the compiler to perform late binding where the compiler matches the object with the right called function and executes it during the runtime. This technique of falls under Runtime Polymorphism.
The term **Polymorphism** means the ability to take many forms. It occurs if there is a hierarchy of classes that are all related to each other by *inheritance*. In simple words, when we break down Polymorphism into *'Poly – Many'* and *'morphism – Forms'* it means showing different characteristics in different situations.



Class Hierarchy

***Note**: In C++ what calling a virtual functions means is that; if we call a member function then it could cause a different function to be executed instead depending on what type of object invoked it. Because overriding from derived classes hasn't happened yet, the virtual call mechanism is disallowed in constructors. Also to mention that objects are built from the ground up or follows a bottom to top approach.*

Consider the following simple program as an example of runtime polymorphism. The main thing to note about the program is that the derived class's function is called using a base class pointer. The idea is that virtual functions are called according to the type of the object instance pointed to or referenced, not according to the type of the pointer or reference. In other words, virtual functions are resolved late, at runtime.
Now, we'll look at an ***example without using the concepts of virtual function*** to clarify your understanding.

```
// C++ program to demonstrate how we will calculate
// area of shapes without virtual function
#include <iostream>
using namespace std;

// Base class
class Shape {
public:
   // parameterized constructor
   Shape(int l, int w)
```

```
    {
        length = l;
        width = w;
    }
    int get_Area()
    {
        cout << "This is call to parent class area\n";
        // Returning 1 in user-defined function means true
        return 1;
    }

protected:
    int length, width;
};

// Derived class
class Square : public Shape {
public:
    Square(int l = 0, int w = 0)
        : Shape(l, w)
    {
    } // declaring and initializing derived class
    // constructor
    int get_Area()
    {
        cout << "Square area: " << length * width << '\n';
        return (length * width);
    }
};
// Derived class
class Rectangle : public Shape {
public:
    Rectangle(int l = 0, int w = 0)
        : Shape(l, w)
    {
    } // declaring and initializing derived class
    // constructor
    int get_Area()
    {
        cout << "Rectangle area: " << length * width
             << '\n';
        return (length * width);
    }
};

int main()
{
    Shape* s;

    // Making object of child class Square
    Square sq(5, 5);

    // Making object of child class Rectangle
    Rectangle rec(4, 5);
    s = &sq; // reference variable
    s->get_Area();
```

```
    s = &rec; // reference variable
    s->get_Area();

    return 0; // too tell the program executed
    // successfully
}
```

Output

```
This is call to parent class area
This is call to parent class area
```

In the above example:

- We store the address of each child's class **Rectangle** and **Square** object in **s** and
- Then we call the **get_Area()** function on it,
- Ideally, it should have called the respective **get_Area()** functions of the child classes but
- Instead, it calls the **get_Area()** defined in the base class.
- This happens due to static linkage which means the call to **get_Area()** is getting set only once by the compiler which is in the base class.

Example: C++ Program to Calculate the Area of Shapes using **Virtual** Function

```
// C++ program to demonstrate how we will calculate
// the area of shapes USING VIRTUAL FUNCTION
#include <fstream>
#include <iostream>
using namespace std;

// Declaration of Base class
class Shape {
public:
    // Usage of virtual constructor
    virtual void calculate()
    {
        cout << "Area of your Shape ";
    }
    // usage of virtual Destuctor to avoid memory leak
    virtual ~Shape()
    {
        cout << "Shape Destuctor Call\n";
    }
};

// Declaration of Derived class
class Rectangle : public Shape {
public:
    int width, height, area;

    void calculate()
    {
        cout << "Enter Width of Rectangle: ";
        cin >> width;

        cout << "Enter Height of Rectangle: ";
        cin >> height;
```

```
        area = height * width;
        cout << "Area of Rectangle: " << area << "\n";
    }

    // Virtual Destuctor for every Derived class
    virtual ~Rectangle()
    {
        cout << "Rectangle Destuctor Call\n";
    }
};

// Declaration of 2nd derived class
class Square : public Shape {
public:
    int side, area;

    void calculate()
    {
        cout << "Enter one side your of Square: ";
        cin >> side;

        area = side * side;
        cout << "Area of Square: " << area << "\n";
    }

   // Virtual Destuctor for every Derived class
    virtual ~Square()
    {
        cout << "Square Destuctor Call\n";
    }
};

int main()
{

    // base class pointer
    Shape* S;
    Rectangle r;

    // initialization of reference variable
    S = &r;

    // calling of Rectangle function
    S->calculate();
    Square sq;

    // initialization of reference variable
    S = &sq;

    // calling of Square function
    S->calculate();
```

```
    // return 0 to tell the program executed
    // successfully
    return 0;
}
```

Output:

```
Enter Width of Rectangle: 10
Enter Height of Rectangle: 20
Area of Rectangle: 200
Enter one side your of Square: 16
Area of Square: 256
```

Pure Virtual Functions

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following −

```
class Shape {
protected:
int width, height;

public:
Shape(int a = 0, int b = 0) {
width = a;
height = b;
}

// pure virtual function
virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called **pure virtual function**.

What is the use?

Virtual functions allow us to create a list of base class pointers and call methods of any of the derived classes without even knowing the kind of derived class object.

This C++ material taken from the following websites:

- https://www.tutorialspoint.com/cplusplus
- https://www.w3schools.com/CPP
- https://www.geeksforgeeks.org/cpp-tutorial
- https://www.javatpoint.com/cpp-tutorial
- https://www.programiz.com/cpp-programming