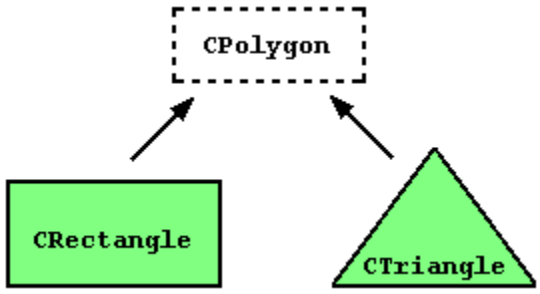
## Inheritance between classes

Classes in C++ can be extended, creating new classes which retain characteristics of the base class. This process, known as inheritance, involves a *base class* and a *derived class*: The *derived class* inherits the members of the *base class*, on top of which it can add its own members.

For example, let's imagine a series of classes to describe two kinds of polygons: rectangles and triangles. These two polygons have certain common properties, such as the values needed to calculate their areas: they both can be described simply with a height and a width (or base).

This could be represented in the world of classes with a class Polygonfrom which we would derive the two other ones: Rectangleand Triangle:




The Polygonclass would contain members that are common for both types of polygon. In our case: width and height. And Rectangle and Triangle would be its derived classes, with specific features that are different from one type of polygon to the other.

Classes that are derived from others inherit all the accessible members of the base class. That means that if a base class includes a member A and we derive a class from it with another member called B, the derived class will contain both member Aand member B.

The inheritance relationship of two classes is declared in the derived class. Derived classes definitions use the following syntax:

class derived_class_name: public base_class_name
{ /*...*/ };

Where derived_class_nameis the name of the derived class and base_class_nameis the name of the class on which it is based. The public access specifier may be replaced by any one of the other access specifiers (protected or private). This access specifier limits the most accessible level for the members inherited from the base class: The members with a more accessible level are inherited with this level instead, while the members with an equal or more restrictive access level keep their restrictive level in the derived class.

```
1 // derived classes
2 #include <iostream>
3 using namespace std;
4
5 class Polygon {
6   protected:
7     int width, height;
```

```
20
10
```

```
  public:
    void set_values (int a, int b)
      { width=a; height=b;}
 };

class Rectangle: public Polygon {
  public:
    int area ()
      { return width * height; }
 };

class Triangle: public Polygon {
  public:
    int area ()
      { return width * height / 2; }
  };

int main () {
  Rectangle rect;
  Triangle trgl;
  rect.set_values (4,5);
  trgl.set_values (4,5);
  cout << rect.area() << '\n';
  cout << trgl.area() << '\n';
  return 0;
}
```

The objects of the classes Rectangleand Triangleeach contain members inherited from Polygon. These are: width, heightand set_values.

The protectedaccess specifier used in class Polygonis similar to private. Its only difference occurs in fact with inheritance: When a class inherits another one, the members of the derived class can access the protected members inherited from the base class, but not its private members.

By declaring widthand heightas protectedinstead of private, these members are also accessible from the derived classes Rectangleand Triangle, instead of just from members of Polygon. If they were public, they could be accessed just from anywhere.

We can summarize the different access types according to which functions can access them in the following way:

| Access | public | protected | private |
|---|---|---|---|
| members of the same class | yes | Yes | yes |
| members of derived class | yes | Yes | no |
| not members | yes | No | no |

Where "not members" represents any access from outside the class, such as from main, from another class or from a function.

In the example above, the members inherited by Rectangleand Trianglehave the same access permissions as they had in their base class Polygon:

```
Polygon::width           // protected access
Rectangle::width         // protected access

Polygon::set_values()    // public access
Rectangle::set_values()  // public access

```

This is because the inheritance relation has been declared using the publickeyword on each of the derived classes:

```
class Rectangle: public Polygon { /* ... */ }
```

This publickeyword after the colon (:) denotes the most accessible level the members inherited from the class that follows it (in this case Polygon) will have from the derived class (in this case Rectangle). Since publicis the most accessible level, by specifying this keyword the derived class will inherit all the members with the same levels they had in the base class.

With protected, all public members of the base class are inherited as protectedin the derived class. Conversely, if the most restricting access level is specified (private), all the base class members are inherited as private.

For example, if daughter were a class derived from mother that we defined as:

```
class Daughter: protected Mother;
```

This would set protectedas the less restrictive access level for the members of Daughterthat it inherited from mother. That is, all members that were publicin Motherwould become protectedin Daughter. Of course, this would not restrict Daughterfrom declaring its own public members. That *less restrictive access level* is only set for the members inherited from Mother.

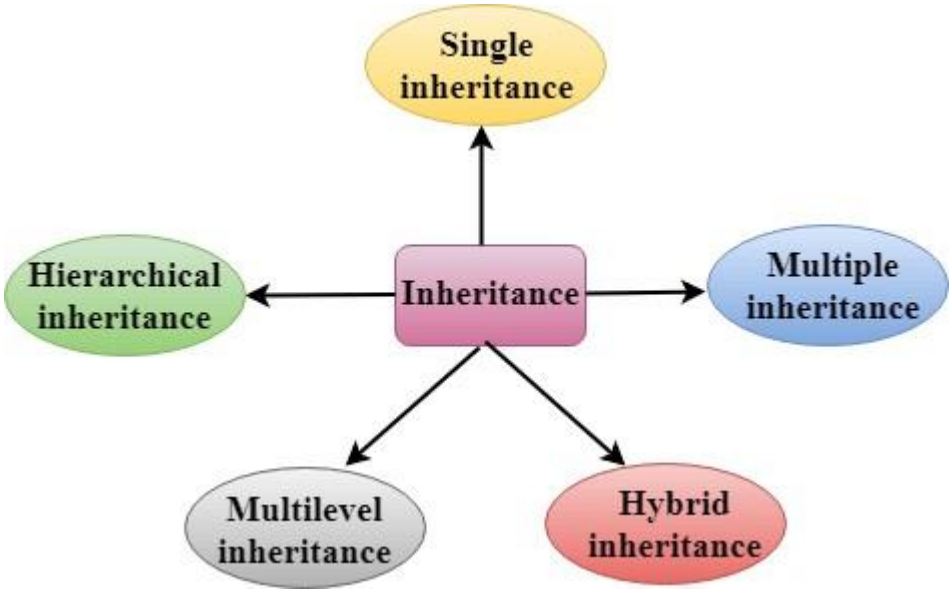
If no access level is specified for the inheritance, the compiler assumes private for classes declared with keyword classand public for those declared with struct.

Actually, most use cases of inheritance in C++ should use public inheritance. When other access levels are needed for base classes, they can usually be better represented as member variables instead.

## Types Of Inheritance

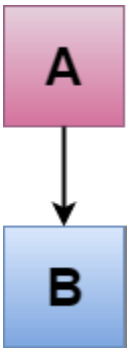
**C++ supports five types of inheritance:**

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance
- Hybrid inheritance




## C++ Single Inheritance

**Single inheritance** is defined as the inheritance in which a derived class is inherited from the only one base class.




Where 'A' is the base class, and 'B' is the derived class.

C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
1. #include <iostream>
2. using namespace std;
3.  class Account {
4.    public:
5.  float salary = 60000; 6.
    };
7.    class Programmer: public Account {
8.    public:
9.    float bonus = 5000;
10.   };
11. int main(void) {
12.     Programmer p1;
13.     cout<<"Salary: "<<p1.salary<<endl;
14.     cout<<"Bonus: "<<p1.bonus<<endl;
15.    return 0;
16. }
```

Output:

```
Salary: 60000
Bonus: 5000
```

In the above example, Employee is the **base** class and Programmer is the **derived** class.

C++ Single Level Inheritance Example: Inheriting Methods

Let's see another example of inheritance in C++ which inherits methods only.

```
1. #include <iostream>
2. using namespace std;
3.  class Animal {
4.    public:
5.  void eat() {
6.  cout<<"Eating..."<<endl; 7.
    }
8.    };
```

```
9.      class Dog: public Animal
10.    {
11.        public:
12.      void bark(){
13.     cout<<"Barking...";
14.      }
15.    };
16. int main(void) {
17.     Dog d1;
18.     d1.eat();
19.     d1.bark();
20.     return 0;
21. }
```

Output:

```
Eating...
Barking...
```

## C++ Multilevel Inheritance

**Multilevel inheritance** is a process of deriving a class from another derived class.


C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

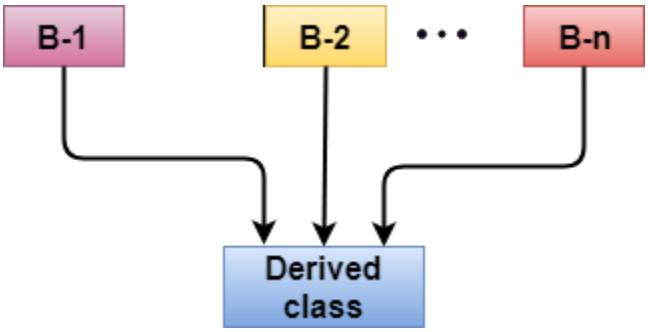
```
1. #include <iostream>
2. using namespace std;
3.  class Animal {
4.    public:
5.  void eat() {
6.  cout<<"Eating..."<<endl; 7.
    }
8.    };
9.    class Dog: public Animal
10.   {
11.      public:
12.    void bark(){
13.    cout<<"Barking..."<<endl;
14.     }
15.   };
16.   class BabyDog: public Dog
17.   {
18.      public:
19.    void weep() {
20.    cout<<"Weeping...";
21.     }
22.   };
23. int main(void) {
24.    BabyDog d1;
25.    d1.eat();
26.    d1.bark();
27.     d1.weep();
28.    return 0;
29. }
```

Output:

```
Eating...
Barking...
Weeping...
```

C++ Multiple Inheritance

**Multiple inheritance** is the process of deriving a new class that inherits the attributes from two or more classes.




**Syntax of the Derived class:**

```
1. class D : visibility B-1, visibility B-2, ? 2.
{
3.      // Body of the class;
4. }
```

Let's see a simple example of multiple inheritance.

```
1.  #include <iostream>
2.  using namespace std;
3.  class A
4. {
5.      protected:
6.       int a;
7.      public:
8.      void get_a(int n)
9.      {
10.         a = n;
11.     }
12. };
13.
14. class B
15. {
16.     protected:
17.     int b;
18.     public:
19.     void get_b(int n)
20.     {
21.         b = n;
```

```
    }
};
class C : public A,public B
{
  public:
   void display()
   {
       std::cout << "The value of a is : " <<a<< std::endl;
       std::cout << "The value of b is : " <<b<< std::endl;
       cout<<"Addition of a and b is : "<<a+b;
   }
};
int main()
{
  C c;
  c.get_a(10);
  c.get_b(20);
  c.display();

   return 0;
}
```

Output:

```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

In the above example, class 'C' inherits two base classes 'A' and 'B' in a public mode.

Ambiquity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
class A
{
    public:
    void display()
```

```
7.      {
8.          std::cout << "Class A" << std::endl;
9.      }
10. };
11. class B
12. {
13.     public:
14.     void display()
15.     {
16.         std::cout << "Class B" << std::endl;
17.     }
18. };
19. class C : public A, public B
20. {
21.     void view()
22.     {
23.         display();
24.     }
25. };
26. int main()
27. {
28.     C c;
29.     c.display();
30.     return 0;
31. }
```

Output:

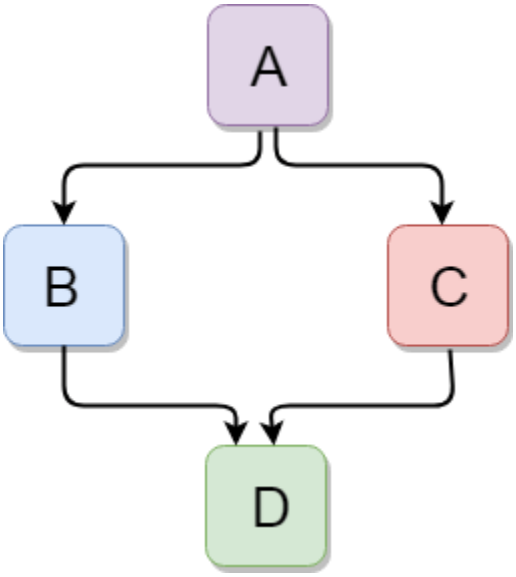
```
error: reference to 'display' is ambiguous
        display();
```

The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
1. class C : public A, public B 2.
{
3.      void view()
4.      {
5.          A :: display();         // Calling the display() function of class A.
6.          B :: display();         // Calling the display() function of class B.
7.
8.      }
9. };
```

## C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.




Let's see a simple example:

```
#include <iostream>
using namespace std;
class A
{
    protected:
    int a;
    public:
    void get_a()
    {
       std::cout << "Enter the value of 'a' : " << std::endl;
       cin>>a;
    }
};

class B : public A
{
    protected:
    int b;
    public:
```

```
20.    void get_b()
21.    {
22.        std::cout << "Enter the value of 'b' : " << std::endl;
23.       cin>>b;
24.    }
25. };
26. class C
27. {
28.    protected:
29.    int c;
30.    public:
31.    void get_c()
32.    {
33.        std::cout << "Enter the value of c is : " << std::endl;
34.        cin>>c;
35.    }
36. };
37.
38. class D : public B, public C
39. {
40.    protected:
41.    int d;
42.    public:
43.    void mul()
44.    {
45.         get_a();
46.         get_b();
47.         get_c();
48.         std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
49.    }
50. };
51. int main()
52. {
53.    D d;
54.    d.mul();
55.    return 0;
56. }
```

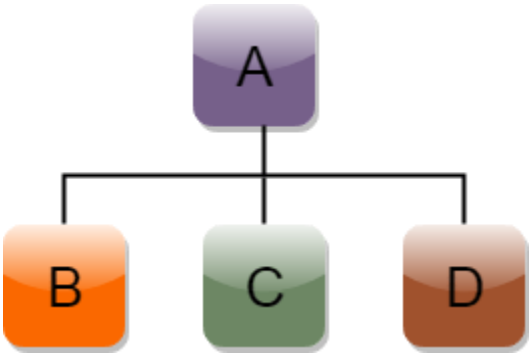
Output:

```
Enter the value of 'a' :
10
```

```
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

## C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.




**Syntax of Hierarchical inheritance:**

```
1. class A 2.
{
3.      // body of the class A.
4. }
5. class B : public A 6.
{
7.      // body of class B.
8. }
9. class C : public A
10. {
11.     // body of class C.
12. }
13. class D : public A
14. {
15.     // body of class D.
16. }
```

Let's see a simple example:

```
1. #include <iostream>
2. using namespace std;
```

```
3.  class Shape                    // Declaration of base class.
4. {
5.      public:
6.      int a;
7.      int b;
8.      void get_data(int n,int m) 9.
        {
10.         a= n;
11.         b = m;
12.     }
13.};
14. class Rectangle : public Shape // inheriting Shape class
15. {
16.     public:
17.     int rect_area()
18.     {
19.         int result = a*b;
20.         return result;
21.     }
22.};
23. class Triangle : public Shape        // inheriting Shape class
24. {
25.     public:
26.     int triangle_area()
27.     {
28.         float result = 0.5*a*b;
29.         return result;
30.     }
31.};
32. int main()
33. {
34.     Rectangle r;
35.     Triangle t;
36.     int length,breadth,base,height;
37.     std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
38.     cin>>length>>breadth;
39.     r.get_data(length,breadth);
40.     int m = r.rect_area();
41.     std::cout << "Area of the rectangle is : " <<m<< std::endl;
42.     std::cout << "Enter the base and height of the triangle: " << std::endl;
43.     cin>>base>>height;
```

```
    t.get_data(base,height);
    float n = t.triangle_area();
    std::cout <<"Area of the triangle is : " << n<<std::endl;
    return 0;
}
```

Output:

```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

## Difference between Containership and Inheritance in C++

**Containership:** When an object of one class is created into another class then that object will be a member of that class, this type of relationship between the classes is known as **containership** or **has_a** relationship as one class contains the object of another class.
The class which contains the object and members of another class in this kind of relationship is called a **container class** and the object that is part of another object is called a **contained object** whereas the object that contains another object as its part or attribute is called a **container object**.

```
// C++ program to illustrate the
// concept of containership
#include <iostream>
using namespace std;

class first {
public:
    void showf()
    {
        cout << "Hello from first class\n";
    }
};

// Container class
class second {

    // Create object of the first-class
    first f;

public:
    // Define Constructor
    second()
    {
        // Call function of first-class
        f.showf();
```

```
    }
};

// Driver Code
int main()
{
    // Create object of second class
    second s;
}
```

**Output:**

Hello from first class

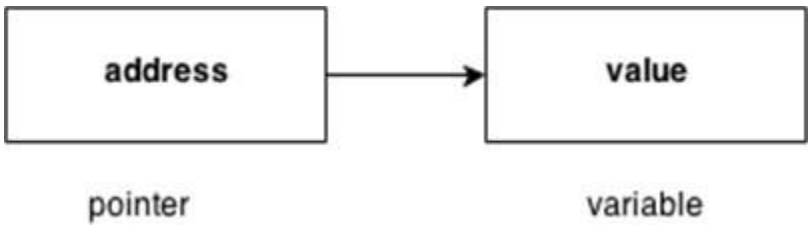
Explanation: In the second class above, there is an object of class first. This type of inheritance is known as has_a relationship as the class second has an object of class first as its member. From object f, the function of class first is called.

## Difference between Inheritance and Containership:

| Inheritance | Containership |
|---|---|
| It enables a class to inherit data and functions from a base class | It enables a class to contain objects of different classes as its data member. |
| The derived class may override the functionality of the base class. | The container class can't override the functionality of the contained class. |
| The derived class may add data or functions to the base class. | The container class can't add anything to the contained class. |
| Inheritance represents a **"is-a"** relationship. | Containership represents a **"has-a"** relationship. |

## C++ Pointers

The pointer in C++ language is a variable, it is also known as locator or indicator that points to an address of a value.



**Advantage of pointer**

1) Pointer reduces the code and improves the performance, it is used to retrieving strings, trees etc. and used with arrays, structures and functions.

2) We can return multiple values from function using pointer.

3) It makes you able to access any memory location in the computer's memory.

**Usage of pointer**

There are many usage of pointers in C++ language.

**1) Dynamic memory allocation**

In c language, we can dynamically allocate memory using malloc() and calloc() functions where pointer is used.

**2) Arrays, Functions and Structures**

Pointers in c language are widely used in arrays, functions and structures. It reduces the code and improves the performance.

Symbols used in pointer

| Symbol | Name | Description |
|---|---|---|
| & (ampersand sign) | Address operator | Determine the address of a variable. |
| * (asterisk sign) | Indirection operator | Access the value of an address. |

Declaring a pointer

The pointer in C++ language can be declared using * (asterisk symbol).

```
1. int *  a; //pointer to int
2. char * c; //pointer to char
```

Pointer Example

Let's see the simple example of using pointers printing the address and value.

```
1. #include <iostream>
```

```
2. using namespace std;
3. int main()
4. {
5. int number=30;
6. int * p;
7. p=&number;//stores the address of number variable
8. cout<<"Address of number variable is:"<<&number<<endl;
9. cout<<"Address of p variable is:"<<p<<endl;
10. cout<<"Value of p variable is:"<<*p<<endl;
```

```
11.    return 0;
12. }
```

**Output:**

```
Address of number variable is:0x7ffccc8724c4
Address of p variable is:0x7ffccc8724c4
Value of p variable is:30
```

Pointer Program to swap 2 numbers without using 3rd variable

```
1.  #include <iostream>
2.  using namespace std;
3.  int main()
4. {
5. int a=20,b=10,*p1=&a,*p2=&b;
6. cout<<"Before swap: *p1="<<*p1<<" *p2="<<*p2<<endl; 7.
*p1=*p1+*p2;
8.  *p2=*p1-*p2;
9.  *p1=*p1-*p2;
10. cout<<"After swap: *p1="<<*p1<<" *p2="<<*p2<<endl;
11.    return 0;
12. }
```

**Output:**

```
Before swap: *p1=20 *p2=10
After swap: *p1=10 *p2=20
```

## C++ this Pointer

In C++ programming, **this** is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

- It can be used **to pass current object as a parameter to another method.**
- It can be used **to refer current class instance variable.**
- It can be used **to declare indexers.**

C++ this Pointer Example

Let's see the example of this keyword in C++ that refers to the fields of current class.

```
1. #include <iostream>
2. using namespace std;
3. class Employee {
4.    public:
5.        int id; //data member (also instance variable)
6.        string name; //data member(also instance variable)
7.        float salary;
8.        Employee(int id, string name, float salary) 9.
          {
10.            this->id = id;
11.           this->name = name;
12.           this->salary = salary;
13.       }
14.       void display()
15.       {
16.           cout<<id<<" "<<name<<" "<<salary<<endl;
17.       }
18. };
19. int main(void) {
20.    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object of Employee
21.    Employee e2=Employee(102, "Nakul", 59000); //creating an object of Employee
22.    e1.display();
23.    e2.display();
24.    return 0;
25. }
```

Output:

```
101  Sonoo  890000
102  Nakul  59000
```

**int const***

**int const*** is pointer to constant integer
This means that the variable being declared is a pointer, pointing to a constant integer. Effectively, this implies that the pointer is pointing to a value that shouldn't be changed. Const qualifier doesn't affect the pointer in this scenario so the pointer is allowed to point to some other address.
The first const keyword can go either side of data type, hence **int const*** is equivalent to **const int***.

```
#include <stdio.h

int main(){
```

```
    const int q = 5;

    int const* p = &q;



    //Compilation error

    *p = 7;

    const int q2 = 7;

    //Valid

    p = &q2;

    return 0;

}
```

**int *const**

**int *const** is a constant pointer to integer

This means that the variable being declared is a constant pointer pointing to an integer. Effectively, this implies that the pointer shouldn't point to some other address. Const qualifier doesn't affect the value of integer in this scenario so the value being stored in the address is allowed to change.

```
#include <stdio.h>



int main(){

        const int q = 5;

        //Compilation error

        int *const p = &q;



        //Valid

        *p = 7;



        const int q2 = 7;
```

```
    //Compilation error

    p = &q2;


    return 0;

}
```

**const int* const**

**const int* const** is a constant pointer to constant integer
This means that the variable being declared is a constant pointer pointing to a constant integer. Effectively, this implies that a constant pointer is pointing to a constant value. Hence, neither the pointer should point to a new address nor the value being pointed to should be changed.
The first const keyword can go either side of data type, hence **const int* const** is equivalent to **int const* const**.

```
#include <stdio.h>

int main(){
   const int q = 5;

   //Valid
   const int* const p = &q;

   //Compilation error
   *p = 7;

   const int q2 = 7;

   //Compilation error
   p = &q2;

   return 0;
}
```

**Static Variables**

- **Static variables in a Function**: When a variable is declared as static, space for **it gets allocated for the lifetime of the program**. Even if the function is called multiple times, space for the static variable is allocated only once and the value of variable in the previous call gets carried through the next function call.

```
// C++ program to demonstrate
```

```
// the use of static Static
// variables in a Function
#include <iostream>
#include <string>
using namespace std;

void demo()
{
    // static variable
    static int count = 0;
    cout << count << " ";

    // value is updated and
    // will be carried to next
    // function calls
    count++;
}

int main()
{
```

```
    for (int i=0; i<5; i++)

        demo();

    return 0;

}
```

Output:

0 1 2 3 4

You can see in the above program that the variable count is declared as static. So, its value is carried through the function calls. The variable count is not getting initialized for every time the function is called.

**Static variables in a class**: As the variables declared as static are initialized only once as they are allocated space in separate static storage so, the static variables **in a class are shared by the objects.** There can not be multiple copies of same static variables for different objects.

```
// C++ program to demonstrate static

// variables inside a class



#include<iostream>

using namespace std;



class ABC

{

public:

    static int i;
```

```
    ABC()
    {
        // Do nothing
    };
};

int ABC::i = 1;

int main()
{
    ABC obj;
    // prints value of i
    cout << obj.i;
}
```

Output:

1

**Static functions in a class**: Just like the static data members or static variables inside the class, static member functions also does not depend on object of class. We are allowed to invoke a static member function using the object and the '.' operator but it is recommended to invoke the static members using the class name and the scope resolution operator.
**Static member functions are allowed to access only the static data members or other static member functions**, they can not access the non-static data members or member functions of the class.

```
// C++ program to demonstrate static
```

```
// member function in a class
#include<iostream>
using namespace std;

class ABC
{
  public:

    // static member function
    static void printMsg()
    {
        cout<<"Welcome!";
    }
};

// main function
int main()
{
    // invoking a static member function
    ABC::printMsg();
```

}

Output:

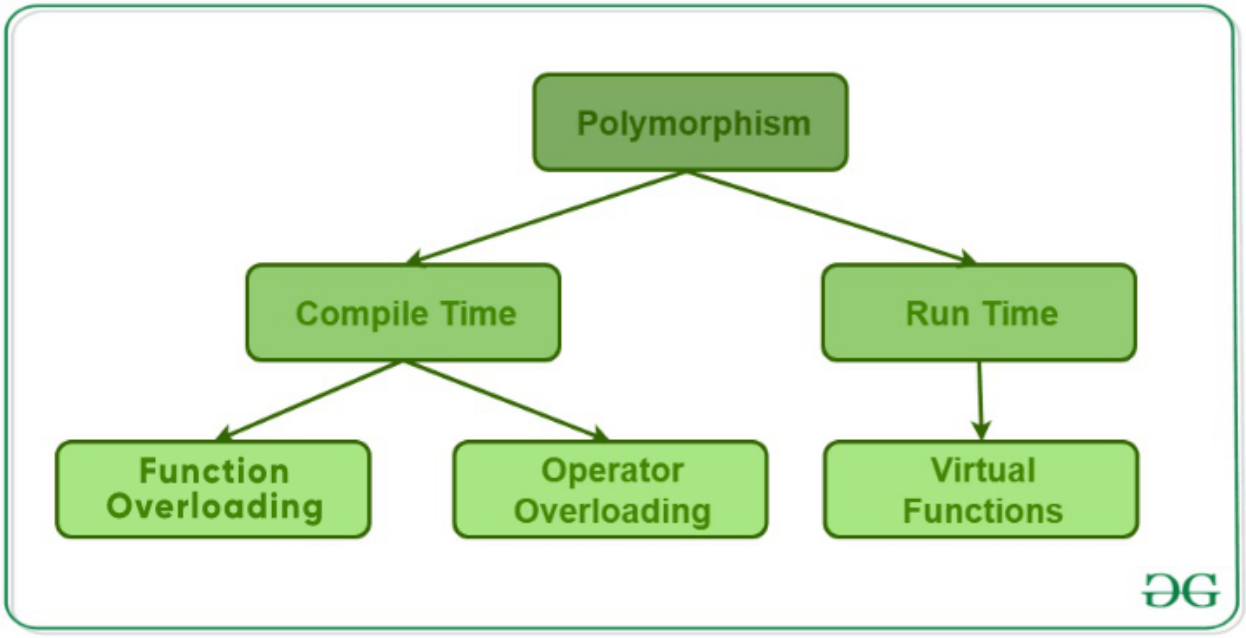
Welcome!

## C++ Polymorphism

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.

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

- Compile time Polymorphism
- Runtime Polymorphism




1. **Compile time polymorphism**: This type of polymorphism is achieved by function overloading or operator overloading.

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

```
// C++ program for function overloading
#include <bits/stdc++.h>

using namespace std;
class XYZ
{
   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() {

    XYZ 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 the property of polymorphism.

**Operator Overloading:**

C++ also provide option to overload operators. For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task 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. The operator '+' is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers.

**Runtime polymorphism**: This type of polymorphism is achieved by Function Overriding and virtual functions .

**Function overriding** on the other hand 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**.

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

### C++ Function Overriding Example

Let's see a simple example of Function overriding in C++. In this example, we are overriding the eat() function

```
#include <iostream>
using namespace std;
class Animal {
   public:
void eat(){
cout<<"Eating...";
   }
};
class Dog: public Animal
{
 public:
 void eat()
   {
     cout<<"Eating bread...";
   }
};
int main(void) {
  Dog d = Dog();
  d.eat();
  return 0;
}
```

Output:

```
Eating bread...
```

## C++ virtual function

Virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.

- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

### Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

```
1. #include <iostream>
2. using namespace std;
3. class A
4. {
5.     int x=5;
```

```
6.      public:
7.      void display()
8.      {
9.          std::cout << "Value of x is : " << x<<std::endl;
10.     }
11. };
12. class B: public A
13. {
14.     int y = 10;
15.     public:
16.     void display()
17.     {
18.         std::cout << "Value of y is : " <<y<< std::endl;
19.     }
20. };
21. int main()
22. {
23.     A *a;
24.     B b;
25.     a = &b;
26.    a->display();
27.     return 0;
28. }
```

**Output:**

```
Value of x is : 5
```

In the above example, * a is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

C++ virtual function Example

Let's see the simple example of C++ virtual function used to invoked the derived class in a program.

```
1. #include <iostream>
 2. {
3.   public:
4.   virtual void display()
```

```
5.    {
6.     cout << "Base class is invoked"<<endl; 7.
      }
8.  };
9. class B:public A
10. {
11.  public:
12.  void display()
13. {
14. cout << "Derived Class is invoked"<<endl;
15. }
16. };
17. int main()
18. {
19.  A* a;     //pointer of base class
20.  B b;      //object of derived class
21. a = &b;
22.  a->display();      //Late Binding occurs
23. }
```

**Output:**

```
Derived Class is invoked
```

## Pure Virtual Function

- A virtual function is not used for performing any task. It only serves as a placeholder.
- When the function has no definition, such function is known as "**do-nothing**" function.
- The "**do-nothing**" function is known as a **pure virtual function**. A pure virtual function is a function declared in the base class that has no definition relative to the base class.
- A class containing the pure virtual function cannot be used to declare the objects of its own, such classes are known as abstract base classes.
- The main objective of the base class is to provide the traits to the derived classes and to create the base pointer used for achieving the runtime polymorphism.

**Pure virtual function can be defined as:**

```
1. virtual void display() = 0;
```

**Let's see a simple example:**

```
1. #include <iostream>
2. using namespace std;
3. class Base
4. {
5.     public:
6.     virtual void show() = 0; 7.
};
8. class Derived : public Base 9. {
10.    public:
11.    void show()
12.    {
13.        std::cout << "Derived class is derived from the base class." << std::endl;
14.    }
15. };
16. int main()
17. {
18.    Base *bptr;
19.    //Base b;
20.    Derived d;
21.    bptr = &d;
22.    bptr->show();
23.    return 0;
24. }
```

**Output:**

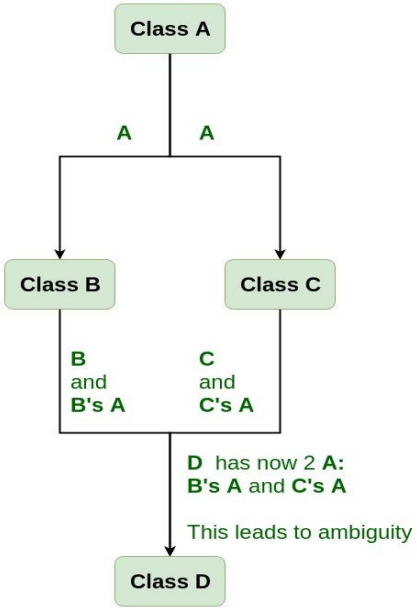
```
Derived class is derived from the base class.
```

In the above example, the base class contains the pure virtual function. Therefore, the base class is an abstract base class. We cannot create the object of the base class.

## Virtual base class in C++

Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.

**Need for Virtual Base Classes:**
Consider the situation where we have one class **A** .This class is **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.




As we can see from the figure that data members/function of class **A** are inherited twice to class **D**. One through class **B** and second through class **C**. When any data / function member of class **A** is accessed by an object of class **D**, ambiguity arises as to which data/function member would be called? One inherited through **B** or the other inherited through **C**. This confuses compiler and it displays error.
To resolve this ambiguity when class **A** is inherited in both class **B** and class **C**, it is declared as **virtual base class** by placing a keyword **virtual** as :

**Syntax for Virtual Base Classes:**

**Syntax 1:**
```
class B : virtual public A
{
};
```

**Syntax 2:**
```
class C : public virtual A
{
};
```

```
#include <iostream>

using namespace std;



class A {

public:
```

```
    int a;
    A() // constructor
    {
        a = 10;
    }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

int main()
{
    D object; // object creation of class d
    cout << "a = " << object.a << endl;
```

```
    return 0;

}
```

**Output:**
a = 10

**Explanation :**The class **A** has just one data member **a** which is **public**. This class is virtually inherited in class **B** and class **C**. Now class **B** and class **C** becomes virtual base class and no duplication of data member **a** is done.