Object-Oriented C++

Abstraction, Inheritance, Polymorphism

Computer Engineering
Shahed University
2021 – winter

Teacher Assistants : **Amirreza Tavakoli**
amirrezatav2@gmail.com

C++ OOPs Concepts

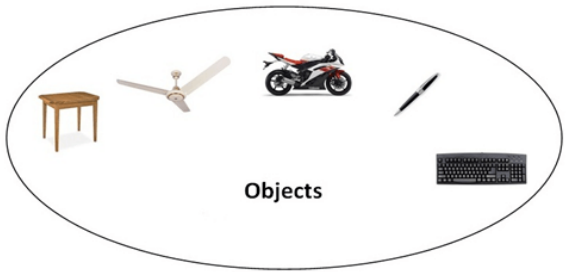
- Object means a real word entity such as pen, chair, table etc.
- Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects.
- It simplifies the software development and maintenance by providing some concepts:
  - Object

1. Class
2. Inheritance
3. Polymorphism
4. Abstraction
5. Encapsulation

C++ OOPs Concepts

- **Object**

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.



- **Class**

Collection of objects is called class. It is a logical entity.

- **Inheritance**

When one object acquires all the properties and behaviors of parent object known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

C++ OOPs Concepts

- **Polymorphism**

When one task is performed by different ways known as polymorphism. For example: to convince the customer differently, to draw something shape or rectangle etc.
In C++, we use Function overloading to achieve polymorphism.

- **Abstraction**

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.
In C++, we use abstract class and interface to achieve abstraction.

- **Encapsulation**

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

C++ Object and Class Example

```
#include <iostream>
using namespace std;
class Student {
   public:
       int id;//data member (also instance variable)
       string name;//data member(also instance variable)
       void insert(int i, string n)
        {
            id = i;
            name = n;
        }
       void display()
        {
            cout<<id<<"  "<<name<<endl;
        }
};
int main(void) {
    Student s1; //creating an object of Student
    Student s2; //creating an object of Student
    s1.insert(201, "Sonoo");
    s2.insert(202, "Nakul");
    s1.display();
    s2.display();
    return 0;
}
```

Output :
201 Sonoo
202 Nakul

C++ Constructor

There can be two types of constructors in C++.

- Default constructor
  - A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.
- Parameterized constructor
  - A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

C++ Default Constructor

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
        Employee()
        {
            cout<<"Default Constructor Invoked"<<endl;
        }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2;
    return 0;
}
```

Output :
Default Constructor Invoked
Default Constructor Invoked

C++ Parameterized Constructor

```
#include <iostream>
using namespace std;
class Employee {
   public:
       int id;//data member (also instance variable)
       string name;//data member(also instance variable)
       float salary;
       Employee(int i, string n, float s)
        {
            id = i;
            name = n;
            salary = s;
        }
       void display()
        {
            cout<<id<<"  "<<name<<"  "<<salary<<endl;
        }
};
int main(void) {
    Employee e1 =Employee(101, "Sonoo", 890000); //creating an object
of Employee
    Employee e2=Employee(102, "Nakul", 59000);
    e1.display();
    e2.display();
    return 0;
}
```

Output :

101 Sonoo 890000

102 Nakul 59000

C++ Copy Constructor

C++ Copy Constructor
A Copy constructor is an overloaded constructor used to declare and initialize an object from another object.

Copy Constructor is of two types:

- **Default Copy constructor:** The compiler defines the default copy constructor. If the user defines no copy constructor, compiler supplies its constructor.
- **User Defined constructor:** The programmer defines the user-defined constructor.

C++ Copy Constructor

C++ Copy Constructor
A Copy constructor is an overloaded constructor used to declare and initialize an object from another object.

Copy Constructor is of two types:

- **Default Copy constructor:** The compiler defines the default copy constructor. If the user defines no copy constructor, compiler supplies its constructor.
- **User Defined constructor:** The programmer defines the user-defined constructor.

C++ Parameterized Constructor

```
#include <iostream>
using namespace std;
class A
{
public:
    int x;
    A(int a)                // parameterized constructor.
    {
        x = a;
    }
};
int main()
{
    A a1(20);               // Calling the parameterized constructor.
    A a2(a1);                //  Calling the copy constructor.
    cout << a2.x;
    return 0;
}
```

Output :
20

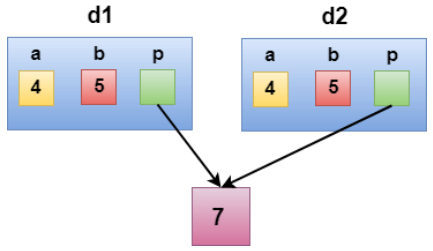
Two types of copies are produced by the constructor:

- Shallow copy
  - The default copy constructor can only produce the shallow copy.
  - A Shallow copy is defined as the process of creating the copy of an object by copying data of all the member variables as it is.
- Deep copy
  - Deep copy dynamically allocates the memory for the copy and then copies the actual value, both the source and copy have distinct memory locations
  - In this way, both the source and copy are distinct and will not share the same memory location.
  - Deep copy requires us to write the user-defined constructor.

C++ Shallow Copy Example

```
#include <iostream>
using namespace std;
class Demo
{
    int a;
    int b;
public:
    int* p;
    Demo()
    {
        p = new int;
    }
    void setdata(int x, int y, int z)
    {
        a = x;
        b = y;
        *p = z;
    }
    void showdata()
    {
        std::cout << "value of a is : " << a << std::endl;
        std::cout << "value of b is : " << b << std::endl;
        std::cout << "value of *p is : " << *p << std::endl;
    }
};
```

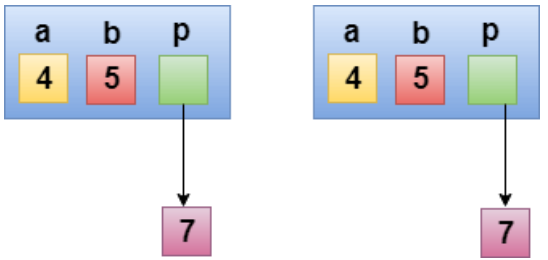
```
int main()
{
    Demo d1;
    d1.setdata(4, 5, 7);
    Demo d2 = d1;
    *d1.p = 8;
    d2.showdata();
    return 0;
}
```



Output :
value of a is : 4
value of b is : 5
value of *p is : 8

Deep copy

```
#include <iostream>
using namespace std;
class Demo
{
public:
    int a;
    int b;
    int* p;
    Demo()
    {
        p = new int;
    }
    Demo(Demo& d)
    {
        a = d.a;
        b = d.b;
        p = new int;
        *p = *(d.p);
    }
    void setdata(int x, int y, int z)
    {
        a = x;
        b = y;
        *p = z;
    }
    void showdata()
    {
        std::cout << "value of a is :
" << a << std::endl;
        std::cout << "value of b is :
" << b << std::endl;
        std::cout << "value of *p is :
" << *p << std::endl;
    }
};
int main()
{
    Demo d1;
    d1.setdata(4, 5, 7);
    Demo d2 = d1;
    *d1.p = 8;
    d2.showdata();
    return 0;
}
```



Output :
value of a is : 4
value of b is : 5
value of *p is : 7

Differences b/w Copy constructor and Assignment operator(=)

Copy Constructor	Assignment Operator
It is an overloaded constructor.	It is a bitwise operator.
It initializes the new object with the existing object.	It assigns the value of one object to another object.

Syntax of copy constructor:
Class_name(const class_name &object_name)
{
// body of the constructor.
}
	Syntax of Assignment operator:
	Class_name a,b;
	b = a;

• The **copy constructor** is invoked when the new object is initialized with the existing object. • The object is passed as an argument to the function. • It returns the object.	The **assignment operator** is invoked when we assign the existing object to a new object.
Both the existing object and new object shares the different memory locations.	Both the existing object and new object shares the same memory location.
If a programmer does not define the copy constructor, the compiler will automatically generate the implicit default copy constructor.	If we do not overload the "=" operator, the bitwise copy will occur.

C++ Destructor

```
#include <iostream>
using namespace std;
class Employee
{
public:
    Employee()
    {
        cout << "Constructor Invoked" << endl;
    }
    ~Employee()
    {
        cout << "Destructor Invoked" << endl;
    }
};
int main(void)
{
    Employee e1; //creating an object of Employee
    Employee e2; //creating an object of Employee
    return 0;
}
```

Output :
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked

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

```
#include <iostream>
using namespace std;
class Employee {
public:
    int id; //data member (also instance variable)
    string name; //data member(also instance variable)
    float salary;
    Employee(int id, string name, float salary)
    {
        this->id = id;
        this->name = name;
        this->salary = salary;
    }
    void display()
    {
        cout << id << "  " << name << "  " << salary << endl;
    }
};
int main(void) {
    Employee e1 = Employee(101, "Sonoo", 890000); //creating an object of Employee
    Employee e2 = Employee(102, "Nakul", 59000); //creating an object of Employee
    e1.display();
    e2.display();
    return 0;
}
```

Output :
101 Sonoo 890000
102 Nakul 59000

C++ static

- In C++, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C++, static can be field, method, constructor, class, properties, operator and event.
- **Advantage of C++ static keyword**
  - **Memory efficient** : Now we don't need to create instance for accessing the static members, so it saves memory. Moreover, it belongs to the type, so it will not get memory each time when instance is created.

C++ static field example

```
#include <iostream>
using namespace std;
class Account {
public:
    int accno; //data member (also instance variable)
    string name; //data member(also instance variable)
    static float rateOfInterest;
    Account(int accno, string name)
    {
        this->accno = accno;
        this->name = name;
    }
    void display()
    {
        cout << accno << "<<name<< " "<<rateOfInterest"<<endl;
    }
};
float Account::rateOfInterest = 6.5;
int main(void) {
    Account a1 = Account(201, "Sanjay"); //creating an object of Employee
    Account a2 = Account(202, "Nakul"); //creating an object of Employee
    a1.display();
    a2.display();
    return 0;
}
```

Output :
201 Sanjay 6.5
202 Nakul 6.5

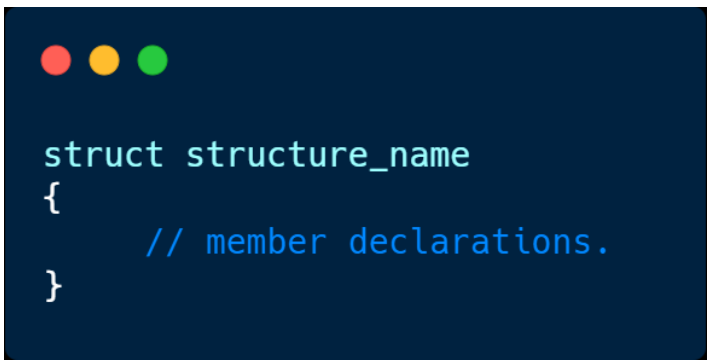
C++ static field example: Counting Objects

```
#include <iostream>
class Account {
public:
    int accno; //data member (also instance variable)
    std::string name;
    static int count;
    Account(int accno, string name)
    {
        this->accno = accno;
        this->name = name;
        count++;
    }
    void display()
    {
    std::cout << accno << " " << name << std::endl;
    }
};
int Account::count = 0;
int main(void) {
    Account a1 = Account(201, "Sanjay"); //creating an object of Account
    Account a2 = Account(202, "Nakul");
    Account a3 = Account(203, "Ranjana");
    a1.display();
    a2.display();
    a3.display();
    std::cout << "Total Objects are: " << Account::count;
    return 0;
}
```

Output :
201 Sanjay
202 Nakul
203 Ranjana
Total Objects are: 3

C++ Structs

- In C++, classes and structs are blueprints that are used to create the instance of a class. Structs are used for lightweight objects such as Rectangle, color, Point, etc.
- Unlike class, structs in C++ are value type than reference type.
- It is useful if you have data that is not intended to be modified after creation of struct.
- C++ Structure is a collection of different data types. It is similar to the class that holds different types of data.
- The Syntax Of Structure:



C++ Structs

```
struct Student
{
    char name[20];
    int id;
    int age;
}
```

in the above case, Student is a structure contains three variables name, id, and age. When the structure is declared, no memory is allocated. When the variable of a structure is created, then the memory is allocated.

Let's understand this scenario.

How to create the instance of Structure?

Structure variable can be defined as:

Student s; → Here, s is a structure variable of type **Student**

When the structure variable is created, the memory will be allocated.
Student structure contains one array char variable and two integer variable.
Therefore, the memory for one array char variable is 1*20 byte and two ints will be 2*4 = 8.
The total memory occupied by the s variable is 28 byte.

How to access the variable of Structure:

The variable of the structure can be accessed by simply using the instance of the structure followed by the dot (.) operator and then the field of the structure.

For example:

```
s.id = 4;
```

In the above statement, we are accessing the id field of the structure Student by using the dot(.) operator and assigns the value 4 to the id field.

C++ Struct Example

```
#include <iostream>
using namespace std;
struct Rectangle
{
    int width, height;

};
int main(void) {
    struct Rectangle rec;
    rec.width = 8;
    rec.height = 5;
    cout << "Area of Rectangle is: " << (rec.width * rec.height)
<< endl;
    return 0;
}
```

Output :
Area of Rectangle is: 40

C++ Struct Example: Using Constructor and Method

```
#include <iostream>
using namespace std;
struct Rectangle {
    int width, height;
    Rectangle(int w, int h)
    {
        width = w;
        height = h;
    }
    void areaOfRectangle() {
        cout << "Area of Rectangle is: " << (width *
height);
    }
};
int main(void) {
    struct Rectangle rec = Rectangle(4, 6);
    rec.areaOfRectangle();
    return 0;
}
```

Output :
Area of Rectangle is: 24

Structure v/s Class

Structure	Class
If access specifier is not declared explicitly, then by default access specifier will be public.	If access specifier is not declared explicitly, then by default access specifier will be private.

Syntax of Structure:
 	
struct structure_name
{
// body of the structure.
};
	Syntax of Class:
	
	class class_name
	{
	// body of the class.
	};

The instance of the structure is known as "Structure variable".	The instance of the class is known as "Object of the class".

C++ Enumeration

- Enum in C++ is a data type that contains fixed set of constants.
- It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY and SATURDAY) , directions (NORTH, SOUTH, EAST and WEST) etc.
- The C++ enum constants are static and final implicitly.
- C++ Enums can be thought of as classes that have fixed set of constants.
- **Points to remember for C++ Enum**
  - **enum improves type safety**
  - **enum can be easily used in switch**
  - **enum can be traversed**
  - **enum can have fields, constructors and methods**
  - **enum may implement many interfaces but cannot extend any class because it internally extends Enum class**

C++ Enumeration Example

```
#include <iostream>
using namespace std;
enum week { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday };
int main()
{
    week day;
    day = Friday;
    cout << "Day: " << day + 1 << endl;
    return 0;
}
```

Output :
Day: 5

C++ Enumeration Example

```
#include <iostream>
using namespace std;
enum colors { red = 5, black };
enum suit { heart, diamond = 8, spade = 3, club };
int main() {
cout << "The value of enum color : " << red << "," << black;
cout << "\nThe default value of enum suit : " << heart << "," << diamond << "," << spade << ","
<< club;
return 0;
}
```

Output :
The value of enum color : 5,6
The default value of enum suit : 0,8,3,4

Q

```
#include <iostream>
using namespace std;
class Box
{
private:
    int length;
public:
    Box() : length(0) { }
};
int printLength(Box b)
{
    b.length += 10;
    return b.length;
}
int main()
{
    Box b;
    cout << "Length of box: " << printLength(b) << endl;
    return 0;
}
```

C++ Friend function

- If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
- By using the keyword friend compiler knows the given function is a friend function.
- For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.
- **Declaration of friend function in C++**

```
class class_name
{
    friend data_type function_name(argument/s);// syntax of friend function.
};
```

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

C++ friend function Example

```
#include <iostream>
using namespace std;
class Box
{
private:
    int length;
public:
    Box() : length(0) { }
    friend int printLength(Box); //friend function
};
int printLength(Box b)
{
    b.length += 10;
    return b.length;
}
int main()
{
    Box b;
    cout << "Length of box: " << printLength(b) << endl;
    return 0;
}
```

Output :
Length of box: 10

C++ friend function Example

```
#include <iostream>
using namespace std;
class B;// forward declarartion.
class A
{
    int x;
public:
    void setdata(int i)
    {
        x = i;
    }
    friend void min(A, B); // friend function.
};
class B
{
    int y;
public:
    void setdata(int i)
    {
        y = i;
    }
    friend void min(A, B); // friend function
};
void min(A a, B b)
{
    if (a.x <= b.y)
        std::cout << a.x << std::endl;
    else
        std::cout << b.y << std::endl;
}
int main()
{
    A a;
    B b;
    a.setdata(10);
    b.setdata(20);
    min(a, b);
    return 0;
}
```

Output :
10

C++ Friend class

```
#include <iostream>

using namespace std;

class A
{
    int x = 5;
    friend class B;// friend class.
};
class B
{
public:
    void display(A& a)
    {
        cout << "value of x is : " << a.x;
    }
};
int main()
{
    A a;
    B b;
    b.display(a);
    return 0;
}
```

Output :
value of x is : 5

C++ Inheritance

- In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically.
- In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.
- In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

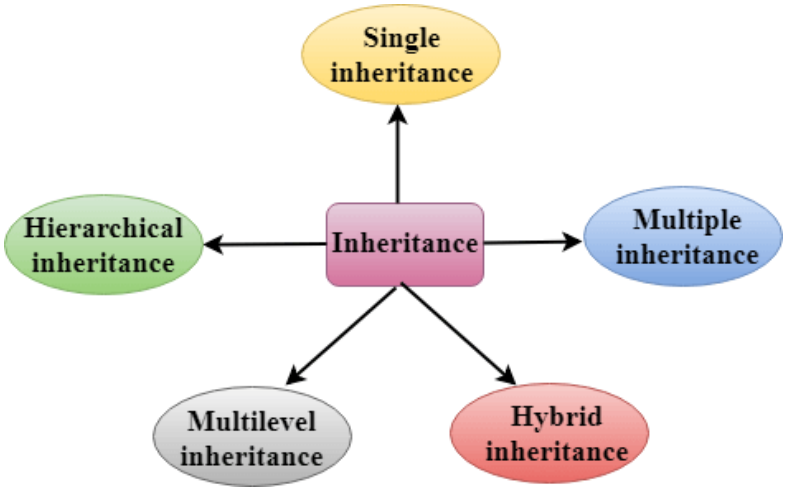
Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

Types Of Inheritance

C++ supports five types of inheritance:

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



Derived Classes

- A Derived class is defined as the class derived from the base class.
- **The Syntax of Derived class:**

derived_class_name: It is the name of the derived class.
visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.
base_class_name: It is the name of the base class.

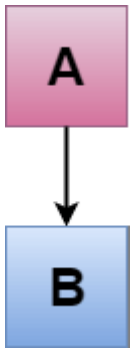
visibility mode

• When the base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class. Therefore, the public members of the base class are not accessible by the objects of the derived class only by the member functions of the derived class.
• When the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the base class.

• In C++, the default mode of visibility is private.
• The private members of the base class are never inherited.

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

```
#include <iostream>
using namespace std;
class Account {
public:
    float salary = 60000;
};
class Programmer : public Account {
public:
    float bonus = 5000;
};
int main(void) {
    Programmer p1;
    cout << "Salary: " << p1.salary << endl;
    cout << "Bonus: " << p1.bonus << endl;
    return 0;
}
```

Output :
Salary: 60000
Bonus: 5000

C++ Single Level Inheritance Example: Inheriting Methods

```
#include <iostream>
using namespace std;
class Animal {
public:
    void eat() {
        cout << "Eating..." << endl;
    }
};
class Dog : public Animal
{
public:
    void bark() {
        cout << "Barking...";
    }
};
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
}
```

Output :
Eating...
Barking...

Let's see a simple example.

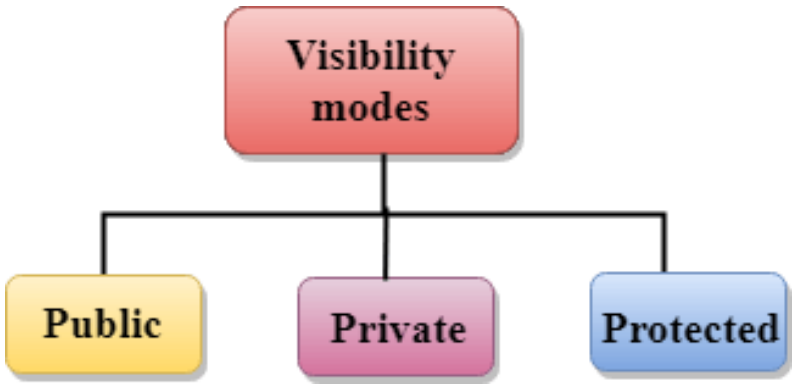
```
#include <iostream>
using namespace std;
class A
{
    int a = 4;
    int b = 5;
public:
    int mul()
    {
        int c = a * b;
        return c;
    }
};

class B : private A
{
public:
    void display()
    {
        int result = mul();
        std::cout << "Multiplication of a and b is : " << result <<
std::endl;
    }
};
```

```
int main()
{
    B b;
    b.display();
    // b.mul(); error
    return 0;
}
```

Output :
Multiplication of a and b is : 20

How to make a Private Member Inheritable

- The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.
- C++ introduces a third visibility modifier, i.e., protected. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.



Visibility of Inherited Members

• **Public**: When the member is declared as public, it is accessible to all the functions of the program.
• **Private**: When the member is declared as private, it is accessible within the class only.
• **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Base class visibility	Derived class visibility		
	Public	Private	Protected
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
Public	Public	Private	Protected

C++ Multilevel Inheritance

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

C++ Multi Level Inheritance Example

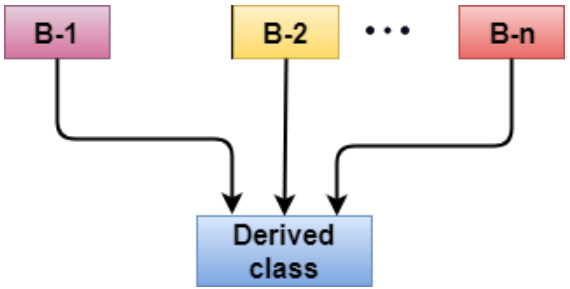
```
#include <iostream>
using namespace std;
class Animal {
public:
    void eat() {
        cout << "Eating..." << endl;
    }
};
class Dog : public Animal
{
public:
    void bark() {
        cout << "Barking..." << endl;
    }
};
class BabyDog : public Dog
{
public:
    void weep() {
        cout << "Weeping...";
    }
};
```

```
int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
    d1.weep();
    return 0;
}
```

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:

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

Let's see a simple example of multiple inheritance.

```
#include <iostream>
using namespace std;
class A
{
protected:
    int a;
public:
    void get_a(int n)
    {
        a = n;
    }
};

class B
{
protected:
    int b;
public:
    void get_b(int n)
    {
        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

Ambiguity 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.

Ambiguity Resolution in Inheritance

```cpp
#include <iostream>
using namespace std;
class A
{
public:
    void display()
    {
        std::cout << "Class A" << std::endl;
    }
};
class B
{
public:
    void display()
    {
        std::cout << "Class B" << std::endl;
    }
};
class C : public A, public B
{
    void view()
    {
        display();
    }
};
```

```cpp
int main()
{
    C c;
    c.display();
    return 0;
}
```

Output :

error: reference to 'display' is ambiguous

display();

Ambiguity Resolution in Inheritance

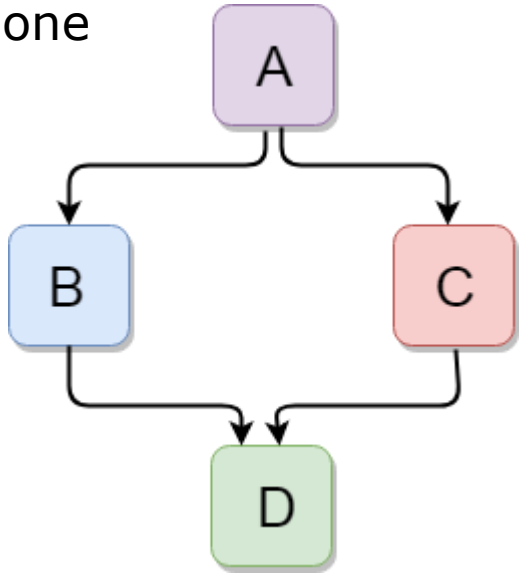
- The last page issue can be resolved by using the class resolution operator with the function.
- In the last page example, the derived class code can be rewritten as:

```
#include <iostream>
using namespace std;
class A
{
public:
    void display()
    {
        std::cout << "Class A" << std::endl;
    }
};
class B
{
public:
    void display()
    {
        std::cout << "Class B" << std::endl;
    }
};
```

```
class C : public A, public B
{
    void view()
    {
        A::display();
    }
};
int main()
{
    C c;
    c.B::display();
    return 0;
}
```

C++ Hybrid Inheritance

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



C++ Hybrid Inheritance 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:
    void get_b()
    {
        std::cout << "Enter the value of 'b' : " <<
std::endl;
        cin >> b;
    }
};
```

```
class C
{
protected:
    int c;
public:
    void get_c()
    {
        std::cout << "Enter the value of c is : "
<< std::endl;
        cin >> c;
    }
};

class D : public B, public C
{
protected:
    int d;
public:
    void mul()
    {
        get_a();
        get_b();
        get_c();
        std::cout << "Multiplication of a,b,c is :
" << a * b * c << std::endl;
    }
};
```

```
int main()
{
    D d;
    d.mul();
    return 0;
}
```

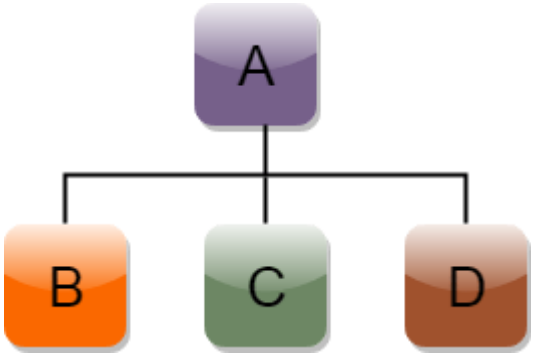
Output :
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000

C++ Hierarchical Inheritance

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

amirrezatav2@gmai

```
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++ Hierarchical Inheritance

```
#include <iostream>
using namespace std;
class Shape                  // Declaration of base class.
{
public:
    int a;
    int b;
    void get_data(int n, int m)
    {
        a = n;
        b = m;
    }
};
class Rectangle : public Shape  // inheriting Shape class
{
public:
    int rect_area()
    {
        int result = a * b;
        return result;
    }
};
```

```
class Triangle : public Shape    // inheriting Shape class
{
public:
    int triangle_area()
    {
        float result = 0.5 * a * b;
        return result;
    }
};
int main()
{
    Rectangle r;
    Triangle t;
    int length, breadth, base, height;
    std::cout << "Enter the length and breadth of a rectangle: " <<
std::endl;
    cin >> length >> breadth;
    r.get_data(length, breadth);
    int m = r.rect_area();
    std::cout << "Area of the rectangle is : " << m << std::endl;
    std::cout << "Enter the base and height of the triangle: " << std::endl;
    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

C++ Aggregation (HAS-A Relationship)

In C++, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.

C++ Aggregation Example

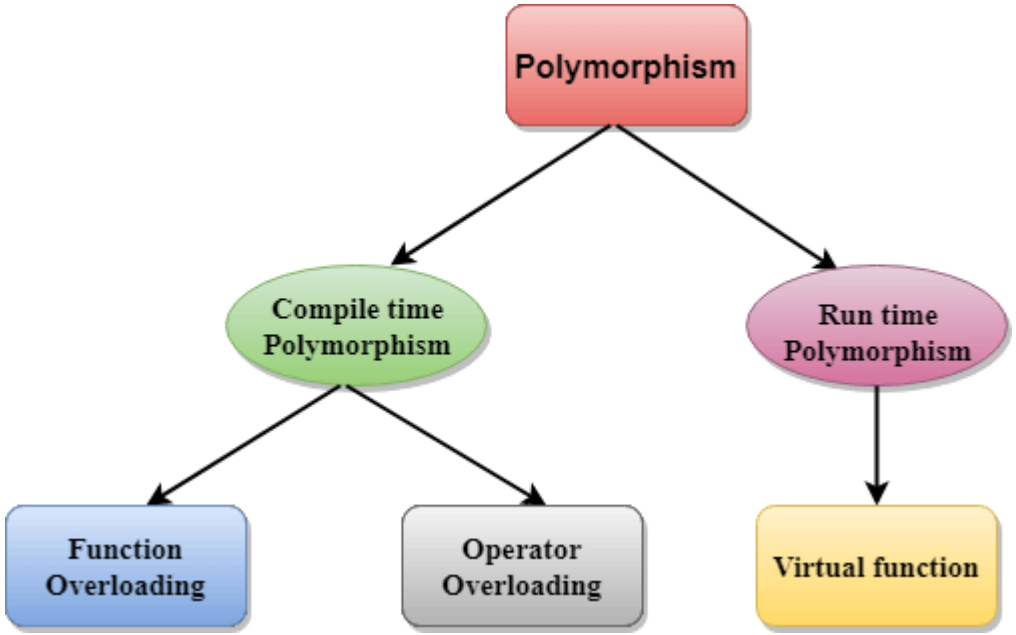
```
#include <iostream>
using namespace std;
class Address {
public:
    string addressLine, city, state;
    Address(string addressLine, string city, string state)
    {
        this->addressLine = addressLine;
        this->city = city;
        this->state = state;
    }
};
class Employee
{
private:
    Address* address;  //Employee HAS-A Address
public:
    int id;
    string name;
    Employee(int id, string name, Address* address)
    {
        this->id = id;
        this->name = name;
        this->address = address;
    }
 void display()
    {
        cout << id << " " << name << " " <<
            address->addressLine << " " << address->city << " " << address-
>state << endl;
    }
};
int main(void) {
    Address a1 = Address("C-146, Sec-15", "Noida", "UP");
    Employee e1 = Employee(101, "Nakul", &a1);
    e1.display();
    return 0;
}
```

Output :
101 Nakul C-146, Sec-15 Noida UP

C++ Polymorphism

- The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.
- **Real Life Example Of Polymorphism**
- Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

There are two types of polymorphism in C++



Compile time polymorphism

The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

Compile time polymorphism

```
// compile-time polymorphism
#include <iostream>
using namespace std;
class GFG {
    // First addition function
public:static int add(int a, int b)
{
    return a + b;
}
      // Second addition function
public:static double add(
    double a, double b)
{
    return a + b;
}
};
// Driver code
int main()
    {
        // Here, the first addition
        // function is called
    cout << GFG::add(2, 3) << endl;

        // Here, the second addition
        // function is called
    cout << GFG::add(2.5, 3.4);
    }
```

Output :
5
5/9

Run time polymorphism

Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

C++ Runtime Polymorphism Example

```
#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...

Runtime Polymorphism with Data Members

```
#include <iostream>
using namespace std;
class Animal {                                    //  base class
declaration.
public:
    string color = "Black";
};
class Dog : public Animal                       // inheriting Animal class.
{
public:
    string color = "Grey";
};
int main(void) {
    Animal d = Dog();
    cout << d.color;
}
```

Output :
Black

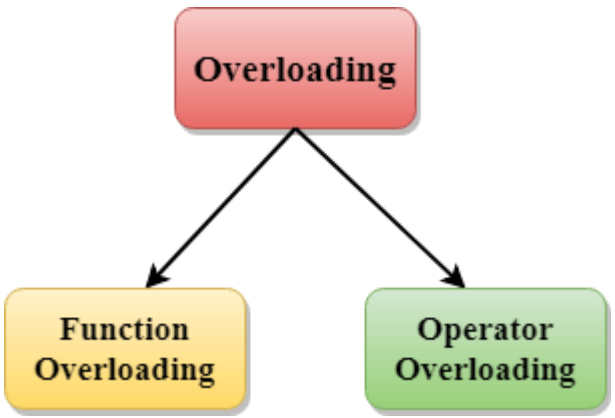
C++ Overloading (Function and Operator)

If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading.
In C++, we can overload:

- methods,
- constructors, and
- indexed properties

Types of overloading in C++ are:

- Function overloading
- Operator overloading



C++ Function Overloading Example

```
#include <iostream>
using namespace std;
class Cal {
public:
    static int add(int a, int b) {
        return a + b;
    }
    static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
int main(void) {
    Cal C; //class object declaration.
    cout << C.add(10, 20) << endl;
    cout << C.add(12, 20, 23);
    return 0;
}
```

Output :
30
55

C++ Function Overloading Example

```
#include <iostream>
using namespace std;
class Cal {
public:
    static int add(int a, int b) {
        return a + b;
    }
    static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
int main(void) {
    Cal C; //class object declaration.
    cout << C.add(10, 20) << endl;
    cout << C.add(12, 20, 23);
    return 0;
}
```

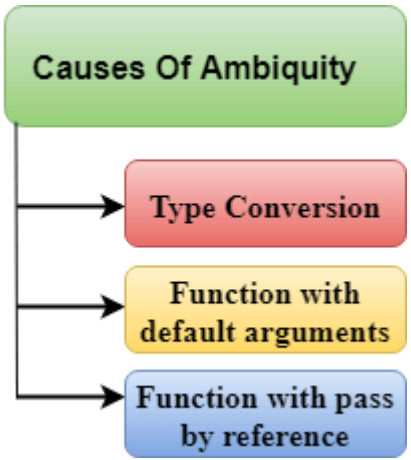
Output :
30
55

C++ Function Overloading Example

```
#include<iostream>
using namespace std;
int mul(int, int);
float mul(float, int);


int mul(int a, int b)
{
    return a * b;
}
float mul(double x, int y)
{
    return x * y;
}
int main()
{
    int r1 = mul(6, 7);
    float r2 = mul(0.2, 3);
    std::cout << "r1 is : " << r1 << std::endl;
    std::cout << "r2 is : " << r2 << std::endl;
    return 0;
}
```

Output :
r1 is : 42
r2 is : 0.6

Function Overloading and Ambiguity



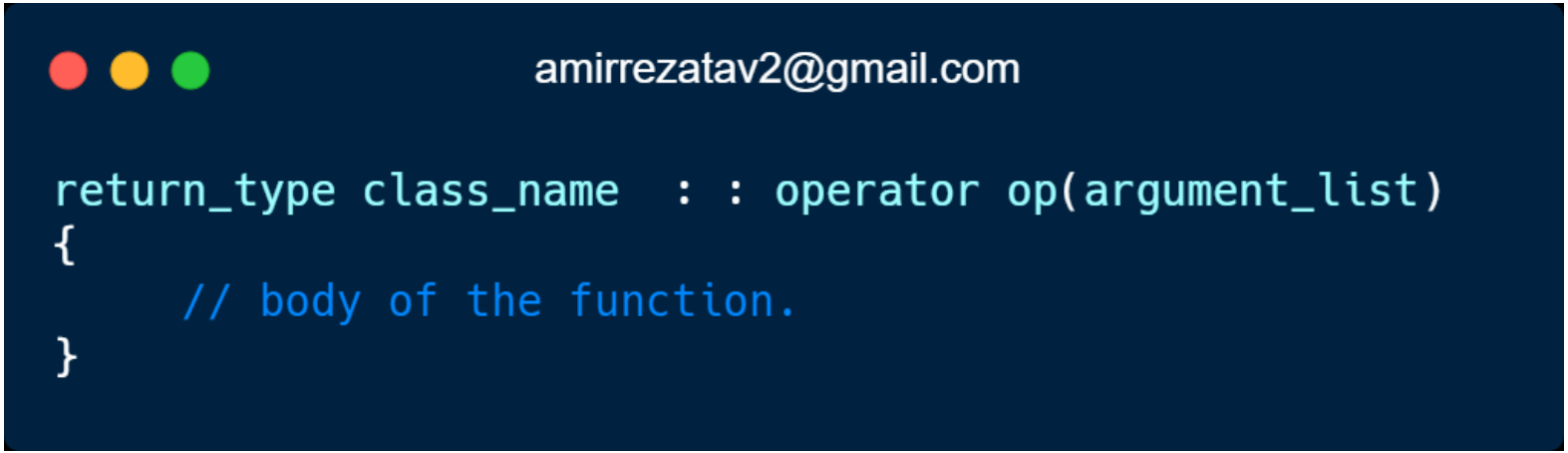
C++ Operators Overloading

- Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.
- Operator overloading is used to overload or redefines most of the operators available in C++.
- It is used to perform the operation on the user-defined data type.
- For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

Operator that cannot be overloaded are as follows:

- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Syntax of Operator Overloading



Where the **return type** is the type of value returned by the function.
class_name is the name of the class.
operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.
- The overloaded operator contains atleast one operand of the user-defined data type.
- We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
- When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
- When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

overload the unary operator ++.

```
#include <iostream>
using namespace std;
class Test
{
private:
    int num;
public:
    Test() : num(8) {}
    void operator ++() {
        num = num + 2;
    }
    void Print() {
        cout << "The Count is: " << num;
    }
};
int main()
{
    Test tt;
    ++tt;  // calling of a function "void operator ++()"
    tt.Print();
    return 0;
}
```

Output :
The Count is: 10

C++ Hierarchical Inheritance

```
#include <iostream>
using namespace std;
class A
{
    int x;
public:
    A() {}
    A(int i)
    {
        x = i;
    }
    void operator+(A);
    void display();
};
void A :: operator+(A a)
{
    int m = x + a.x;
    cout << "The result of the addition of two objects is : " << m;

}
int main()
{
    A a1(5);
    A a2(4);
    a1 + a2;
    return 0;
}
```

Output :
The result of the addition of two objects is : 9

C++ Function Overriding

- 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

```
#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

• A C++ 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.

C++ virtual function Example

```
#include <iostream>
using namespace std;
class A
{
public:
virtual void display()
{
cout << "Base class is invoked" << endl;
}
};
class B :public A
{
public:
void display()
{
cout << "Derived Class is invoked" << endl;
}
};
int main()
{
A* a;    //pointer of base class
B b;     //object of derived class
a = &b;
a->display();   //Late Binding occurs
}
```

Output :
Derived Class is invoked

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Base
{
public:
    virtual void show() = 0;
};
class Derived : public Base
{
public:
    void show()
    {
        std::cout << "Derived class is derived from the base class." <<
std::endl;
    }
};
int main()
{
    Base* bptr;
    //Base b;
    Derived d;
    bptr = &d;
    bptr->show();
    return 0;
}
```

In this 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.

Output :
Derived class is derived from the base class.

Interfaces in C++ (Abstract Classes)

Abstract classes are the way to achieve abstraction in C++. Abstraction in C++ is the process to hide the internal details and showing functionality only. Abstraction can be achieved by two ways:

1.Abstract class
2.Interface
Abstract class and interface both can have abstract methods which are necessary for abstraction.

C++ Abstract class

In C++ class is made abstract by declaring at least one of its functions as <>strong>pure virtual function.
A pure virtual function is specified by placing "= 0" in its declaration.
Its implementation must be provided by derived classes.

example of abstract class in C++

```
#include <iostream>
using namespace std;
class Shape
{
public:
    virtual void draw() = 0;
};
class Rectangle : Shape
{
public:
    void draw()
    {
        cout < < "drawing rectangle..." < < endl;
    }
};
class Circle : Shape
{
public:
    void draw()
    {
        cout << "drawing circle..." < < endl;
    }
};
int main() {
    Rectangle rec;
    Circle cir;
    rec.draw();
    cir.draw();
    return 0;
}
```

Output :
drawing rectangle...
drawing circle...

Data Encapsulation in C++

All C++ programs are composed of the following two fundamental elements −

Program statements (code) − This is the part of a program that performs actions and they are called functions.

Program data − The data is the information of the program which gets affected by the program functions.

Data Encapsulation in C++

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of data hiding.

Data encapsulation is a mechanism of bundling the data, and the functions that use them and data abstraction is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. We already have studied that a class can contain private, protected and public members.

Good luck!