Copy Constructor

A Copy constructor is an overloaded constructor used to declare and initialize an object from another object. Prototype of copy constructor is like ClassName (const ClassName &oldobj). Copy Constructor is of two types:

1. Default Copy constructor:
   The compiler defines the default copy constructor. If the user defines no copy constructor, compiler supplies its constructor.

2. User Defined constructor:
   The programmer defines the user-defined constructor.

When Copy Constructor is called

Copy Constructor is called in the following scenarios:

1. When we initialize the object with another existing object of the same class type. For example, Student s1 = s2, where Student is the class.

2. When the object of the same class type is passed by value as an argument.

3. When the function returns the object of the same class type by value.

Two types of copies are produced by the constructor:

1. Shallow copy
2. Deep copy

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. When the memory of any object is freed, the memory of another object is also automatically freed as both the objects point to the same memory location. This problem is solved by the user-defined constructor that creates the Deep copy.

Example Shallow Copy

```cpp
class GPL {
    private:
            int data;
            int *ptr;
```

```cpp
public:
        GPL() { p=new int; }

        void setdata(int data, int ptr) {
                this.data = data;
                this.ptr = &ptr;
        }

        void print() {
                cout << " data : " <<data<<endl;
                cout << " *ptr : " <<<*ptr<<endl;
        }
};

int main()
{
    GPL gpl1, gpl2;
    gpl1.setdata(4,7);
    gpl2.showdata();

    gpl2 = gpl1;
    gpl2.showdata();
    return 0;
}
```
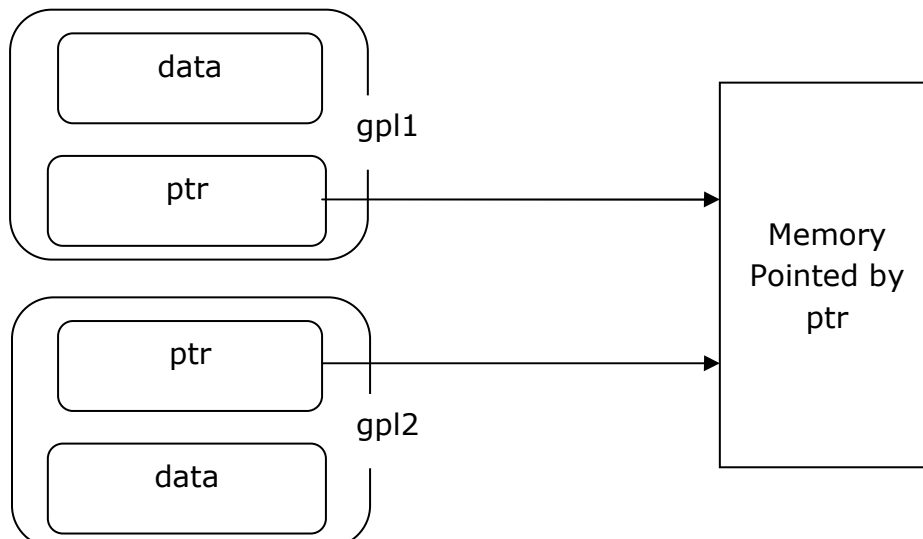
In the above case, a programmer has not defined any constructor, therefore, the statement Demo gpl2 = gpl1 calls the default constructor defined by the compiler. The default constructor creates the exact copy or shallow copy of the existing object. Thus, the pointer ptr of both the objects point to the same memory location.

Therefore, when the memory of any object is freed, the memory of another object is also automatically freed as both the objects point to the same memory location. This problem is solved by the user-defined constructor that creates the Deep copy.

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.

```cpp
class GPL
{
        private:
                int data;
                int *ptr;

        public:
                GPL () { p=new int; }

                GPL (GPL &obj)
                {
                        data = obj.data;
                        ptr = new int;
                        *ptr = *(obj.ptr);
                }

                void setdata(int data, int ptr)
                {
                        this.data = data;
                        this.ptr = &ptr;
                }

                void print()
                {
                        cout << " data : " <<data<<endl;
                        cout << " *ptr : " <<*ptr<<endl;
                }
};

int main()
{
        GPL gpl1, gpl2;
        gpl1.setdata(4,7);
```
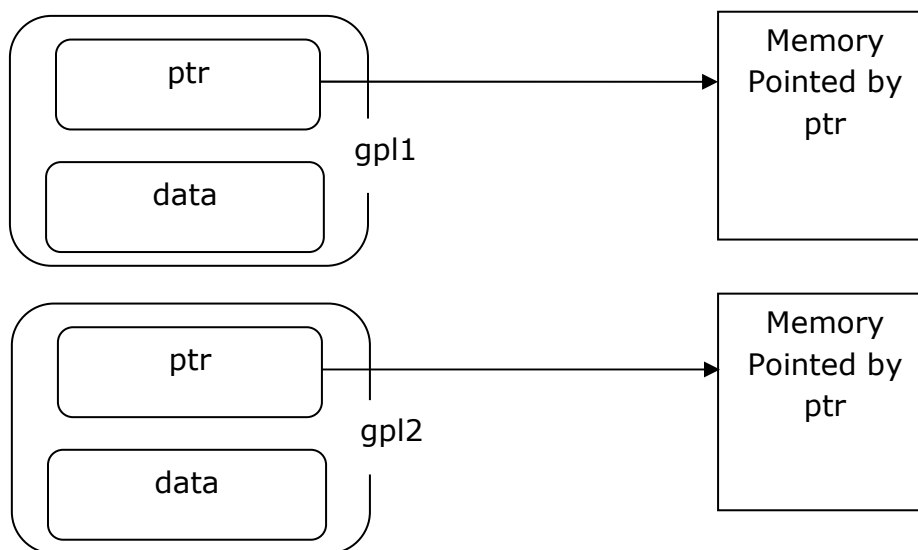
```
        gpl2.showdata();

        gpl2 = gpl1;
        gpl2.showdata();
        eturn 0;
}
```

In the above case, a programmer has defined its own constructor, therefore the statement gpl2 = gpl1 calls the copy constructor defined by the user. It creates the exact copy of the value type data and separate copy of the memory pointed by the pointer ptr.

When should we write our own copy constructor?

The problem with default copy constructor (and assignment operator) is – When we have members which dynamically gets initialized at run time, default copy constructor copies this members with address of dynamically allocated memory and not real copy of this memory. Now both the objects points to the same memory and changes in one reflects in another object, Further the main disastrous or fatal effect is, when we delete one of this object other object still points to same memory, which will be dangling pointer, and memory leak is also possible problem with this approach.

Hence, in such cases, we should always write our own copy constructor (and assignment operator).

Structure vs Class in C++

In C++, a structure is same as class except the following differences:

1) Members of a class are private by default and members of struct are public by default.

2) When deriving a struct from a class or struct, default access-specifier for a base class or struct is public. When deriving a class, default access specifier is private.

```
// class Derived : private Base {}

class Base
{
    public:    int x;
};

class Derived : Base { };

int main()
{
  Derived d;
  d.x = 20;        //Error
  return 0;
}

// compiler error as inheritance is private
```

```
// struct Derived : public Base {}

class Base
{
    public:    int x;
};

struct Derived : Base { };

int main()
{
  Derived d;
  d.x = 20;
  return 0;
}

// works fine as inheritance is public
```

Polymorphism

The word polymorphism means having many forms. Polymorphism can be defined as the ability of a message to be displayed in more than one form. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming where object is behaving different way for same behavior name. In C++ polymorphism is mainly divided into two types:

1. Compile time Polymorphism (Early Binding, Static Binding)
2. Runtime Polymorphism (Dynamic Binding or Late Binding)

Compile time Polymorphism

When it is decided at compile time before the program starts execution that what will be the behavior of an object of a class called Compile time Polymorphism or Early Binding or Static Binding. We have two functions with same name but

different number of arguments. Based on how many parameters we pass during function call determines which function is to be called, this is why it is considered as an example of polymorphism because in different conditions the output is different. Since, the call is determined during compile time so it is called compile time polymorphism. In C++ Programming Compile time Polymorphism is achieved by following features

1. Function Overloading
2. Operator Overloading
3. Constructor Overloading

Polymorphism

Compile Time Polymorphism

Run Time Polymorphis

Function Overloadin

Operator Overloadin

Virtual Function
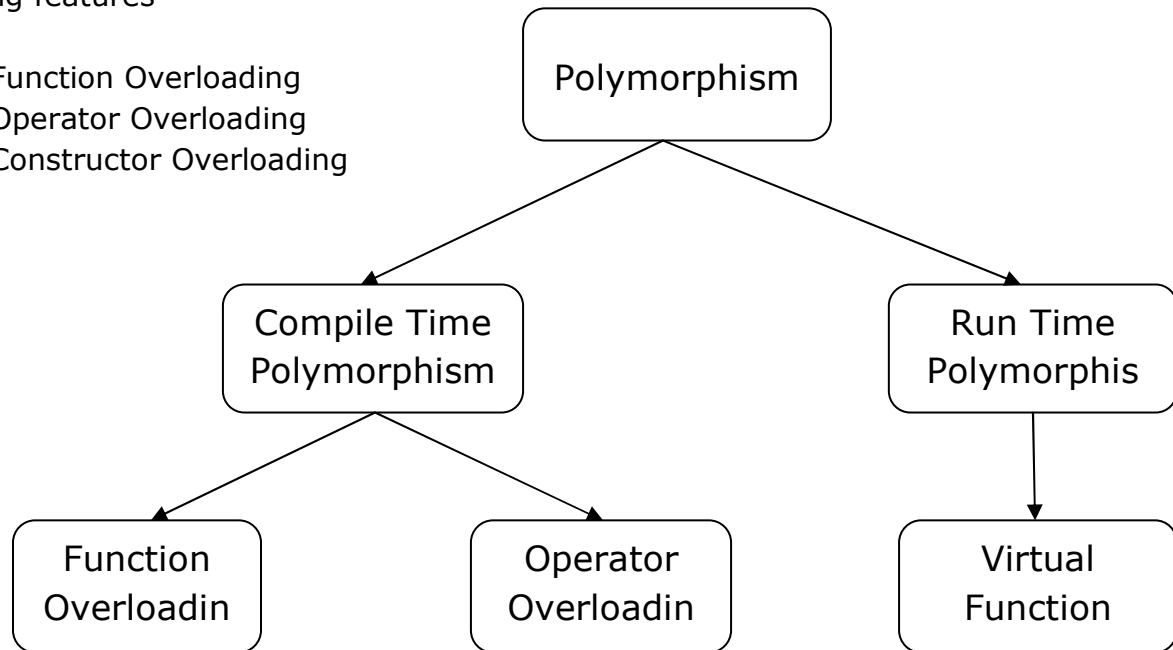
Example 1

```
#include <iostream.h>

class CompileTimePoly
{
  public:
      int sum(int a, int b){
            return a+b;
      }
      int sum(int a, int b, int numc3){
            return a+b+c;
      }
};

int main()
{
      CompileTimePoly obj;
      cout<<"First Function: "<<obj.sum(10, 20)<<endl;
      cout<<" Second Function: "<<obj.sum(11, 22, 33);
```

```
        return 0;
}
```

Output:
First Function: 30
Second Function: 66

Example 2

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

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

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

Runtime Polymorphism

Function overriding or redefinition of function is an example of Runtime polymorphism which is achieved IN C++ using virtual function. When child class declares a method, which is already present in the parent class then this is called function overriding, here child class overrides the parent class. In case of function overriding we have two definitions of the same function- one is parent class and

one in child class. The call to the function is determined at runtime to decide which definition of the function is to be called, that is the reason it is called runtime polymorphism.

A virtual function

A virtual function is a member function that you expect to be redefined in derived classes. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

Virtual functions ensure that the correct function is called for an object, regardless of the expression used to make the function call. Functions in derived classes override virtual functions in base classes only if their type is the same. When calling a function using pointers or references, the following rules apply:

1. A call to a virtual function is resolved according to the underlying type of object for which it is called.
2. A call to a non-virtual function is resolved according to the type of the pointer or reference.

Rules of Virtual Function
1. Virtual functions must be members of some class.
2. Virtual functions cannot be static members.
3. They are accessed through object pointers.
4. They can be a friend of another class.
5. 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.

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

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

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

Using Virtual Keyword and Accessing Private Method of Derived class. We can call private function of derived class from the base class pointer with the help of virtual keyword. Compiler checks for access specifier only at compile time. So at run time when late binding occurs it does not check whether we are calling the private function or public function.

Example

```cpp
#include <iostream>

class A
{
   public:
      virtual void show()
      {
            cout << "Public Member in Base class\n";
      }
};

class B: public A
{
   private:
      void show()
      {
            cout << "Private Member in Derived class\n";
```

```cpp
    }
};

int main()
{
    A *a;
    B b;
    a = &b;
    a->show();
}
```

Output: Private Member in Derived class

The above main function is able to call private member of Class B. This creates security threats in program because private member is not accessible outside the scope of the class.

Mechanism of Late Binding in C++

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

vptr : Virtual Pointer that points to function of class to which Object belongs to.   **VTable** : Virtual Table that stores addresses of virtual functions.
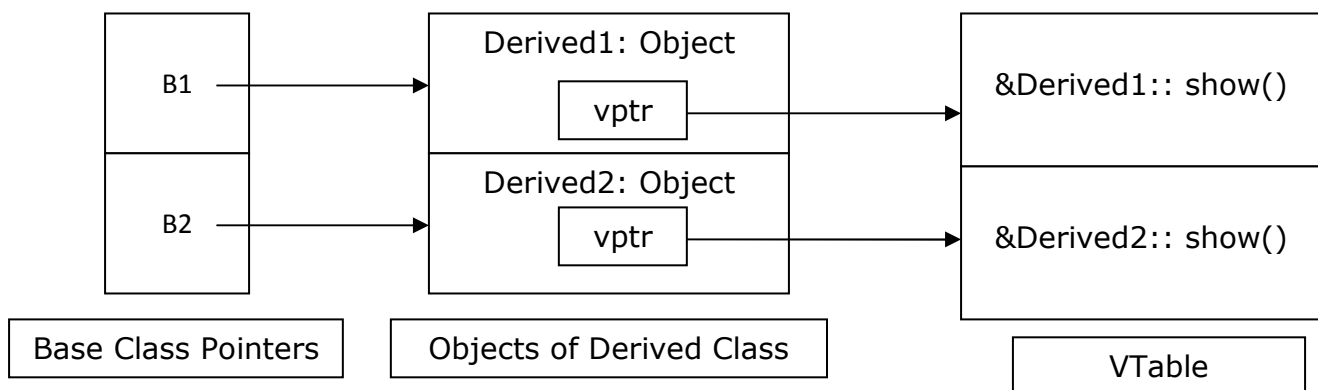
Ambiguity Caused in Function Overloading

When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as function overloading. When the compiler shows the ambiguity error, the compiler does not run the program. Ambiguity in function overloading is caused by

1. Type Conversion.
2. Function with default arguments.
3. Function with pass by reference.

Type Conversion

```cpp
void fun(int i)
{
   cout << "Value of i is : " <<i<< endl;
}
void fun(float j)
{
   cout << "Value of j is : " <<j<< endl;
}
int main()
{
   fun(12);
   fun(1.2);  // This Will Cause Error 1.2 is double
   return 0;
}
```

The statement fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double in void fun (float j), the program works. Therefore, this is a type conversion from float to double.

Function with default arguments.

```cpp
void fun(int a)
{
     cout << "Value of i is : " << a << endl;
}


void fun(int a, int b=9)
{
   cout << "Value of a is : " << a << endl;
   cout << "Value of b is : " << b << endl;
}
```

```
int main()
{
    fun(12);
    return 0;
}
```

The above example shows an error "call of overloaded **fun(int)** is ambiguous". The **fun (int a, int b=9)** can be called in two ways: first is by calling the function with one argument **fun(12)** and another way is calling the function with two arguments **fun(4,5)**. Therefore, the compiler could not be able to select among **fun(int a)** and **fun(int a, int b=9)**.

Function with pass by reference.

```
void fun(int x) {
    cout << "Value of x is : " << x << endl;
}

void fun(int &b) {
    cout << "Value of b is : " << b << endl;
}

int main() {
    int a=10;
    fun(a);      // Error, which f()?
    return 0;
}
```

The above example shows an error "call of overloaded **'fun(int&)'** is ambiguous". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the **fun(int)** and **fun(int &)**.

Operator 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 and the operator can be operated on user defined data variables as well. 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.

The advantage of Operators overloading is to perform different operations on the same operand.

Operators that cannot be overloaded are as follows:

1. Scope operator (::)
2. sizeof
3. member selector(.)
4. member pointer selector(*)
5. ternary operator(?:)

Rules for Operator Overloading

1. Existing operators can only be overloaded, but the new operators cannot be overloaded.

2. The overloaded operator contains at least one operand of the user-defined data type.

3. We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.

4. 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.

5. 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.

Example

```
class GPLG
{
    private:
        int branches;

    public:
        Test() { branches = 8;}

    void operator ++(int)
    {
        branches = branches + 2;
```

```cpp
        }

        // Overload Binary Operator Using Member Function
        GPLG operator+ (const GPLG arg)
        {
                GPLG temp;
                temp. branches = arg. branches + this. Branches;
                return temp
        }

        // Overload Binary Operator Using Friend  Function
        friend GPLG operator + (const GPLG &arg1, const GPLG &arg2)
        {
                    GPLG temp;
                    temp. branches = arg1. branches + arg2. Branches;
                    return temp
        }

        void print()
        {
                cout<<"The Branch Count is: "<< branches;
        }
};

int main()
{
        GPLG gplg1, gplg2, gplg3;

        gplg1++;                       // calling : "void operator ++()"
        gplg3 = gplg1 + gplg2;   // calling : "GPLG operator +( GPLG arg)"
        print (gplg1);
}
```

Output: The Branch Count is: 10

Operator Overloading: Examples

```cpp
class Person
{
  private:
        int height, weight;
```

```cpp
    public:
        Person (int h = 0, int w =0) {
            real = h;    imag = w;
        }

        friend ostream & operator << (ostream &out, const Person &p);
        friend istream & operator >> (istream &out, const Person &p);
};

ostream & operator << (ostream &out, const Person &p);
{
   out << "Height: " << p.height;
   out << "Weight: " << p.weight << endl;
   return out;
}

istream & operator >> (istream &in, const Person &p);
{
   cout << "Enter Height: ";
   in >> p.height;
   cout << "Enter Weight: ";
   in >> p.weight;
   return in;
}
```

Example Overloading Array [] Operator

Function int &Array::operator[](int index) returns a reference as array element can be put on left side of assignment.

```cpp
class Array
{
    private:
        int *ptr;
        int size;

    public:
        Array(int *, int);

        int &operator[] (int);

        void print() const;
};
```

```cpp
// Constructor for Array Class Act as Default and Parameterized.
Array::Array (int *p = NULL, int s = 0)
{
    size = s;
    ptr = NULL;
    if (s != 0)
    {
        ptr = new int[s];
        for (int i = 0; i < s; i++)
            ptr[i] = p[i];
    }
}

// Overloading [] operator to access elements in array style.

int &Array::operator[](int index)
{
    if (index >= size)
    {
        cout << "Array index out of bound, exiting";
        exit(0);
    }
    return ptr[index];
}

void Array::print() const
{
    for(int i = 0; i < size; i++)
        cout<<ptr[i]<<" ";
    cout<<endl;
}

int main()
{
    int a[] = {1, 2, 4, 5};        // Integer Array

    Array arr1(a, 4);              // Constructor Call

    arr1[2] = 6;                   // Overloaded Operator [ ]

    arr1[8] = 6;                   // Overloaded Operator [ ]

    return 0;
}
```

Example Overloading () Function Call Operator

```cpp
class GPL
{
        private:
                int branches;
                int laboratory;
                int classrooms;

        public:
                //default constructor of Class GPL
                GPL () {
                        branches = 0;
                        baboratory = 0;
                        classrooms = 0;
                }

                // Following Implementation is overloaded () function call operator
                GPL operator()(int branch, int lab, int room)
                {
                        GPL gpl;
                        gpl.branches = branch;
                        gpl.baboratory = lab;
                        gpl.classrooms = room;
                        return gpl;
                }
};

int main()
{
        GPL gpl1, gpl2;          // Constructor Call
        gpl2 = gpl1 (5, 12, 10);  // Overloaded Function Call Operator Call
}
```