

# OOP

## Lecture 10

9/11/2021

### Defining Member Functions Outside the class :

Member

functions are the functions, which have their declaration inside the class definition (inside class brackets) and work on the data members of the class. The definition of member functions can be inside or outside the definition of class. If the member function is defined inside the class definition it can be defined directly, but if it's defined outside the class, then we have to use the scope resolution :: operator along with class name and with function name.

### Sample of Defining member functions inside class :

```
class Cube
{
    int side;

    public:
    // Declaration and definition
    int getVolume()
    { return side*side*side; }
};
```

## Sample of Defining member functions outside the class :

```
class Cube
{

    int side;

    public:
    // Declaration
    int getVolume();

};

// Definition

int Cube :: getVolume()
{ return side*side*side; }
```

## Why do we define Function outside the class?

There are several reasons for which we have to define the member functions outside the class. Like,

- In some cases we are working on the existing classes rather than creating our own classes, in such cases we don't manipulate the whole class we just add the prototype(declaration) of the function inside the class and write our function outside the class.
- Secondly, it makes it easier to understand the structure of the class by just going through all the prototypes of the functions. If we define a lot of functions inside the class then it will make a lot of excess code that we have to go through to find our required function.
- Defining Function outside the class is a good practice regarding memory management. The classes we make are loaded in the memory before the execution of the program. If we have defined all

the functions inside the class, then all the functions will also be loaded in the memory at once, no matter if we are using them in the class or not. On the other hand, if we define functions outside the class then all the prototypes will be loaded with the class and the function will only be loaded if we call that function. It makes a lot of memory efficient code.

### Sample Program :

```
using namespace std;
//This code is written to define member function outside the class
class Point
{
    int x, y;

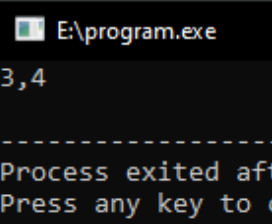
public:
    void setX(int); //Only declaration here

    void setY(int y1)
    {
        if (y1 < 0)    y = 0;
        y = y1;
    }

    void show() const{ cout << x << ',' << y << '\n';}
};

int main(){
    Point p1;
    p1.setX(3);
    p1.setY(4);
    p1.show();
    return 0;
}

//Member function definition outside the class
void Point::setX(int x1)
{
    if (x1 < 0)    x = 0;
    x = x1;
}
```



### Passing Objects to Member Functions :

## Why do we need to pass the objects to a member function?

Sometimes, we need to pass the object to a member function if we are working with more than one object.

There are 2 important terms regarding passing objects,

### 1. Current Object:

Current object is the caller the object which is actually calling the member function using dot (.) operator.

### 2. Passed Object :

Passed object is the object which is passed to the member Function.

### Example

```
p1.isSame( p2 );
```

In this case p1 is the caller object and p2 is the passed object.

### Sample Program :

```
#include <iostream>

using namespace std;
//This code is written to show member function with class object as parameter
class Point
{
    int x, y;
public:
    Point(int x1, int y1){
        x = x1;
        y = y1;
    }

    //Function with object as parameter
    bool isSame(Point &p)
    {
        return (x==p.x && y==p.y);
    }
    bool isDifferent(Point &p)
    {
        return (x!=p.x || y!=p.y);
    }

    void show() const{ cout << x << ',' << y << '\n';}
};

int main(){
    Point p1(1, 2), p2(4,3);
    p1.show();
    p2.show();
    if (p1.isSame(p2))        cout << "Both are same \n";
    else                     cout << "Both are different \n";
    if (p2.isDifferent(p1))  cout << "Both are different \n";
    else                     cout << "Both are same \n";
    return 0;
}
```

E:\program.exe

```
1,2
4,3
Both are different
Both are different

-----
Process exited after 0.09536 s
Press any key to continue . .
```

## Explanations :

Here, while passing the objects to the member functions we used this prototype :

```
bool isSame( Point &p)
```

- **bool** is a datatype in C++ which can only take 2 values i.e false (0) and true (1). Here, bool is used as the return type of the function; it says that the function can only return true or false.
- **isSame** is the name of the function. We can specify any name here just following the naming rules of variables.
- **Point &p** is the parameter which is passed to the function. **Point** is the name of the class which now acts as a datatype itself. **&p** is the name of the object which is passed into the function. Similarly, we can use any name here as well.

## Why did we use & sign here?

We can understand this question by following a simpler example,  
Consider a prototype of a function:

- void func ( int abc );
- void func ( int &abc );

Both of the above prototypes are valid and can be used, But what's the difference here?

The Difference is simple 😊

In the first case, when we didn't use the & sign, a new local variable is created which can only be used in the function and will be destroyed when the function ends. This definition is known as Passing by Value.

In the Second case, by using & sign we can define an alias of the same variable which is passed to this function. Alias means عكس. When we create an alias of an object it means that we are operating on the same

object but with a different name. The changes we do in the function will also be applicable in the main variable which is passed to the function. You can try by passing different variables to such a function and manipulating their values in the function and printing them in the main function. The key advantage of using an alias is that it is memory efficient. In the above example when we don't use & sign, then a new variable is created and ultimately it will take an extra 4 bytes in the memory. While alias works on the same data so it will not take any extra space in the memory. The difference of passing is that we cannot pass constant values to an alias, we have to create and pass a variable to an alias. Like, if an alias is defined in the function so calling a function like `func( 4);` is not valid it will give an error. However, it is possible and valid without using aliases.

The same concept applies in the case of objects of class. Using local variables instead of aliases is somehow accepted because the size of a variable is too small as compared to the size of the whole object of a class. While passing the objects of a class to a member function it is highly recommended to use aliases of the object to save memory. Like, imagine if the size of an object is 1000 bytes then we are wasting 1000 extra bytes just to make the copy of the object into the class which is a huge loss of memory. Also, it will call the copy constructor everytime we call the function, to make a separate copy of the object, which ultimately affects the execution time of the program and makes it slower.

- Inside the Function, whenever we use simple `x` and `y`, it means that we are accessing the data members of the current object. We can also use `this->x` and `this->y` instead. And when we use `p.x` and `p.y` it means that we are accessing the data members of the passed object.

Remember Caller and Passed Objects? 😊

if ( !remember )

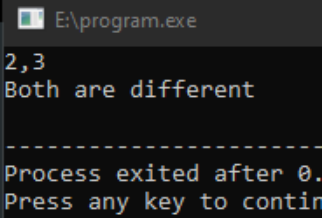
goto page4;

## Passing objects to a Normal (Global) function.

When we pass class objects to the normal functions which are not part of the class( not a member function) Then, the data members of the class are private for the function. In this case the concept of caller function ends as we are no longer calling the function with dot (.) operator. In these functions the best practice is to use getter functions.

### Sample Program :

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Point
6  {
7      int x, y;
8  public:
9      Point(int x1, int y1){
10
11          x = x1;
12          y = y1;
13      }
14      int getX() const{ return x; }
15      int getY() const{ return y; }
16      void show() const{ cout << x << ',' << y << '\n';}
17  };
18  //Global function with two objects, no concept of current object
19  bool isDifferent(Point &p1, Point &p2)
20  {
21      //Getters required to access private data members
22      return (p1.getX()!=p2.getX() || p1.getY()!=p2.getY());
23  }
24
25  int main(){
26      Point p1(2, 3), p2(4,3);
27      p1.show();
28
29      //To call global function no object is required
30      //However, we may have to pass object to global function
31      if (isDifferent(p1, p2)) cout << "Both are different \n";
32      else cout << "Both are same \n";
33      return 0;
34  }
```



E:\program.exe

2,3  
Both are different

-----  
Process exited after 0.  
Press any key to continue

# Concrete / Deep Copy

## vs

# Shallow Copy

In general, creating a copy of an object means to create an exact replica of the object with the same datatype, value and prototype.

There exists a default copy constructor in every class which is used to create a replica of an object.

Consider the following program,

```
#include <ctime>
using namespace std;
//Class is created, where data member has pointer, requires dy
class MyList{
    int *x;
    int size;
public:
    MyList(){
        size = rand()%5+5;
        x = new int[size];
        for (int i=0;i<size;i++)
            x[i] = rand() % 100;
    }
    void show() const{
        for (int i=0;i<size;i++)
            cout << x[i] << ' ';
        cout << '\n';
    }
    ~MyList(){
        delete[] x;
    }
};

int main(){
    srand(time(0));
    MyList list1;
    list1.show();
    MyList list2 = list1;
    list2.show();
    return 0;
}
```

```
E:\program.exe
85 5 0 10 47 97 95
85 5 0 10 47 97 95
-----
Process exited after 3.263
Press any key to continue .
```



**Explanation :** In this program, two data members are created. A pointer named x and a variable named size. When we created the object list1 it called the constructor of the class and created a dynamic array of random size and stored its address in the pointer x. Then , we called the show function. And after that we used a statement “MyList list2 = list1;” what this statement does is it creates a new object named list2 but instead of calling the constructor of the class it calls the **Default Copy Constructor** of the class due to the assignment operator which we used after the object name. The default copy constructor of the class does member - wise copying. It means that it just copied the contents of pointer x and int size into the data members of list2. Such type of **Member - Wise** copying is known as **Shallow Copy**. For shallow copy we don't need to define our own copy constructor explicitly as the default copy constructor does the same job. Shallow works best when we are not using any dynamic memory. Because in that case, we just need to copy all of our variables into the new object. However, there are some limitations while using Shallow copy with dynamic memory.

Check out the previous sample program again,

Now the question arises, what would happen if we change the value of one location in any one list? Does it affect the second list? Here is the sample program to answer this question.

```
program.cpp
7   int *x;
8   int size;
9   public:
10  MyList(){
11      size = rand()%5+5;
12      x = new int[size];
13      for (int i=0;i<size;i++)
14          x[i] = rand() % 100;
15  }
16  void show() const{
17      for (int i=0;i<size;i++)
18          cout << x[i] << ' ';
19      cout << '\n';
20  }
21  void set(int value, int index){
22      if (index<size)
23          x[index]=value;
24  }
25  };
26  int main(){
27      srand(time(0));
28      MyList list1;
29      list1.show();
30      MyList list2=list1;
31      list2.show();//Both lists are same
32      list1.set(-30, 2);//Value is set in list1
33      list1.show();//Lists are printed to check whether they are same
34      list2.show();//or different
35      return 0;
36  }
```

```
79 43 25 85 56
79 43 25 85 56
79 43 -30 85 56
79 43 -30 85 56
-----
Process exited after 0.0818 seconds w
Press any key to continue . . .
```

We can clearly see that when we changed the value of list1 at index 2 it is changed at both of the indexes.... But Why?

Because we did the member wise copy of the lists it means that when the contents of x are copied it just copies the address of the array into the new list. So, now both of the lists are pointing to the same memory locations and accessing the same data from the memory.

To overcome this problem, we need to move towards the concept of Deep Copy.

## Concrete / Deep Copy :

In Deep copy, an object is created by copying data of all variables and it also allocates the same size of dynamic memory with the same values to the new object. In order to perform Deep copy, we need to explicitly define the copy constructor and assign dynamic memory as well if required. Also, it is required to dynamically allocate memory to the variables in the other constructors, as well.

Here is the code of the copy constructor which we have to define explicitly.

**//Copy constructor required to create deep copy in case of dynamic memory**

```
    MyList( const MyList &list )
{
    size = list.size;
    x = new int[ size ];
    for ( int i = 0; i < size; i++ )
        x[ i ] = list.x[ i ];
}
```

**Complete Code :**

```

public:
    MyList(){
        size = rand()%5+5;
        x = new int[size];
        for (int i=0;i<size;i++)
            x[i] = rand() % 100;
    }
    //Copy constructor required to create deep copy in case of dynamic memo
    MyList(const MyList &list){
        size = list.size;
        x = new int[size];
        for (int i=0;i<size;i++)
            x[i] = list.x[i];
    }
    void show() const{
        for (int i=0;i<size;i++)
            cout << x[i] << ' ';
        cout << '\n';
    }
    void set(int value, int index){
        if (index<size)
            x[index]=value;
    }
};

int main(){
    srand(time(0));
    MyList list1;
    list1.show();
    MyList list2=list1;
    list2.show();
    list1.set(-30, 2);
    list2.set(-40, 1);
    list1.show();
    list2.show();
    return 0;
}

```

```

E:\program.exe
58 71 44 22 53
58 71 44 22 53
58 71 -30 22 53
58 -40 44 22 53
-----
Process exited after 0.1686 sec
Press any key to continue . . .

```

**Explanation :** First an object list1 is created and assigned a dynamic array of random size and displayed. Just like the previous example. When the statement ( MyList list2 = list1; ) is executed The compiler called the copy constructor which we have defined explicitly. In the copy constructor we can see that we first assigned the value of size to the new list then we created new memory of the same size and then we copied every element of the previous list to the new array of list2. It means that list1.x[0] is copied to list2.x[0] and list1.x[1] is copied to list2.x[1] and so on. Except the address of the memory location ( Contents of \*x ) which is different in this case because now we are pointing to the new memory.

Now, we can see that when we change the contents of one list it does not affect the contents of the other. Both of the lists are the same initially but independent of each other.

## **Assignment Operator :**

We know that a constructor is called only when a new object is created. Whether it is a default constructor or a copy constructor, it can only be called once when the object is created. It cannot be called again with the same object. The problem arises when we need to copy the contents of an object into an existing object. How can we do this? Let's Find out... 😊

There is an operator called "Assignment Operator" ( = ) which is used to assign values of one variable to the other variable. Same "Assignment operator" can be used to copy an object. However, in classes, the concepts of shallow and deep copy still exist while using assignment operator.

The same concept repeats that there exists a default assignment operator defined in every class which can do shallow copying of objects. However, if we need to do Deep copy we have to define our own assignment operator explicitly.

The Syntax of the Assignment operator will remain the same. It does not depend on whether the operator is built in or defined explicitly.

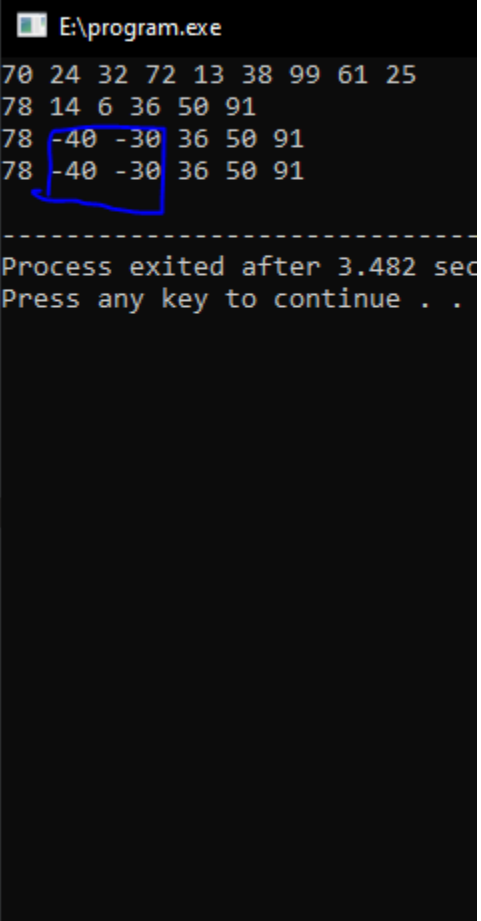
### **Syntax :**

```
Object_2 = Object_1;
```

## Default Assignment Operator for Shallow Copy

```
//Another code to show shallow copy but not using copy constructor
//rather it is using assignment operator
class MyList{
    int *x;
    int size;
public:
    MyList(){
        size = rand()%5+5;
        x = new int[size];
        for (int i=0;i<size;i++){
            x[i] = rand() % 100;
        }
    }
    void show() const{
        for (int i=0;i<size;i++){
            cout << x[i] << ' ';
        }
        cout << '\n';
    }
    void set(int value, int index){
        if (index<size)
            x[index]=value;
    }
    ~MyList(){ delete []x; }
};

int main(){
    srand(time(0));
    MyList list1;
    list1.show();
    MyList list2;
    list2.show();
    list1 = list2;
    list1.set(-30, 2);
    list2.set(-40, 1);
    list1.show();
    list2.show();
    return 0;
}
```



```
E:\program.exe
70 24 32 72 13 38 99 61 25
78 14 6 36 50 91
78 -40 -30 36 50 91
78 -40 -30 36 50 91
-----
Process exited after 3.482 sec
Press any key to continue . .
```

## Deep Copy Using Assignment Operator

//Assignment operator required to create deep copy in case of dynamic memory

```
void operator = ( const MyList &list ){
    delete[ ] x;
    size = list.size;
    x = new int[size];
    for (int i = 0; i < size; i++)
        x[i] = list.x[i];
}
```

```

1  #include <iostream>
2  #include <cstdlib>
3  #include <ctime>
4  using namespace std;
5
6  class MyList{
7      int *x;
8      int size;
9  public:
10     MyList(){
11         size = rand()%5+5;
12         x = new int[size];
13         for (int i=0;i<size;i++)
14             x[i] = rand() % 100;
15     }
16     //Copy constructor required to create deep copy in case of dynamic memory
17     MyList(const MyList &list){
18         size = list.size;
19         x = new int[size];
20         for (int i=0;i<size;i++)
21             x[i] = list.x[i];
22     }
23     //Assignment operator required to create deep copy in case of dynamic memory
24     void operator = (const MyList &list){
25         delete []x;
26         size = list.size;
27         x = new int[size];
28         for (int i=0;i<size;i++)
29             x[i] = list.x[i];
30     }
31     void show() const{
32         for (int i=0;i<size;i++)
33             cout << x[i] << ' ';
34         cout << '\n';
35     }
36     void set(int value, int index){
37         if (index<size)
38             x[index]=value;
39     }
40     ~MyList(){ delete []x; }
41 };
42 int main(){

```

Select E:\program.exe

```

24 77 31 71 27 5
89 18 39 35 56 31
89 18 -30 35 56 31
89 -40 39 35 56 31

```

-----  
Process exited after 0.147  
Press any key to continue

**Important Note:** We deleted the previous memory in the assignment operator function because the size of the new array may be different from the previous array so we should delete the previous memory and allocate new memory in order to avoid any unexpected results.

*--THE END--*