Object Oriented Programming - Lab

LAB 05

Topic: Static Key word, This Pointer, Deep Copy, Shallow copy constructor in C++

Course Code: CL1004

Semester: Spring 2025

Instructor: Engr. Muhammad Qasim

Email: Muhammad.qasim@nu.edu.pk

Department of Computer Science, National University of Computer and Emerging Sciences FAST Peshawar

GETTER AND SETTER METHODS

The meaning of Encapsulation, is to make sure that "sensitive" data is hidden from users. To achieve this, you must declare class variables/attributes as private (cannot be accessed from outside the class). If you want others to read or modify the value of a private member, you can provide public getter and setter methods.

Example

```
#include <iostream>

using namespace std;

class Employee {

private:

// Private attribute

int salary;

public:

// Setter

void setSalary(int s) {

salary = s;

}

// Getter

int getSalary() {

return salary;

}

};

int main() {

  Employee myObj;

  myObj.setSalary(50000);

  cout << myObj.getSalary();

  return 0;

}
```

Output:

50000

Constructor member initializer list

Instead of initializing the private data members inside the body of the constructor, as follows:

Circle(double r = 1.0, string c = "red") {

radius = r;

color = c;

}

We can use an alternate syntax called member initializer list as follows:

Circle(double r = 1.0, string c = "red") : radius(r), color(c) { }

Member initializer list is placed after the constructor's header, separated by a colon (:). Each initializer is in the form of data_member_name(parameter_name). For fundamental type, it is equivalent to data_member_name = parameter_name. For object, the constructor will be invoked to construct the object. The constructor's body (empty in this case) will be run after the completion of member initializer list. It is recommended to use member initializer list to initialize all the data members, as it is often more efficient than doing assignment inside the constructor's body. Also member initializer list syntax is mandatory for initiating the const data members in the classes.

Copy Constructor

Although C++ provides you with a basic copy constructor, but still there are occasions when you need to design your own copy constructor. Given below are some of the reasons why you might want to create a copy constructor.

You need to copy only some of the data members to a new object.

Your objects contain pointers.

Your objects contain dynamic data members.

There may be other numerous reasons why you might want to create a customized copy constructor. Before you begin you must familiarize yourself with the syntax of a copy constructor. A copy constructor always receives an object as a parameter and hence we can extract the data from the parameterized object and place the data in the newly created object. Presented below are the two syntax for copy constructors:

MyClass (MyClass& other); // A copy constructor prototype for a class called MyClass

MyClass (const MyClass& other); //const copy constructor prototype for class called Myclass

In the above prototypes the object which will be copied is called “other”. By writing the const keyword a copy of an object can be created without any change to the inherent data members. Although only some of the data members can be copied.

Copy Constructor is of two types:

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

A user defined copy constructor has the following general function prototype:

ClassName (const ClassName &old_obj);

Copy constructor is initiated by the following line of code written outside the class definition.

ClassName NewObj(OldObj);

OR

ClassName NewObj = OldObj;

Example

Following is a simple example of user defined copy constructor.

```
#include<iostream>

using namespace std;

class Point

{

private:

int x, y;

public:

Point(int x1, int y1) { x = x1; y = y1; }

// Copy constructor

Point(const Point &p2) {x = p2.x; y = p2.y; }

int getX()      {  return x; }

int getY()       {  return y; }


};
```

```
int main()
{
Point p1(10, 15); // Normal constructor is called here
Point p2 = p1; // Copy constructor is called here
// Let us access values assigned by constructors
cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
return 0;
}
```

Default Copy Constructor

If user doesn't provide any version of copy constructor, compiler itself performs the same operation by providing implicit version of copy constructors. But this version only provides shallow copy of the object. User defined copy constructor always make deep copy of the objects.

Shallow copy: both object data members are pointing to same memory location therefore has same values.

Deep copy: both object data members have their own copies of the values in the memory.

Shallow Copy Constructor

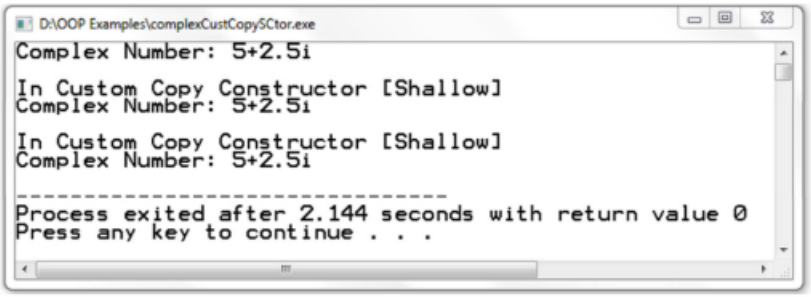
A shallow copy constructor is a copying function that does a member by member copy of one object to another. The copy constructor provided by default is a shallow copy constructor. If a class does not have any dynamic members then only a shallow copy constructor is needed. Consider another case in which you want to create a partial copy of an object i.e. you only want to copy some of the static data members then we need the help of a shallow copy constructor.

```
#include <iostream>
using namespace std;
class complex{
private:
double re, im;
public:
```

```
complex(): re(0),im(0) { }
complex(double r, double i): re(r), im(i) { }
complex(const complex &c){
 // custom copy constructor - Shallow
cout << "\nIn Custom Copy Constructor [Shallow]" << endl;
re = c.re; im = c.im;
}
void show(){
cout<<"Complex Number: "<<re<<"+"<<im<<"i"<<endl;
}
};
int main(){
complex c1(5,2.5);
c1.show();
// use of default copy constructor
// call by two ways
complex c2(c1); // 1. function notation
c2.show();
complex c3 = c2; // 2. assignment operator
c3.show();
return 0;
}
```

Output:

Figure 4.2: Output complexCustCopySCtor.cpp



Deep Copy Constructor

1. It is designed to handle pointers and dynamic data members.
2. Consider a class that has a pointer data member. When a shallow copy constructor is called, it copies the pointer data member to the new object. It might be thought that this is what being wanted but in fact it is wrong because copying a pointer means that the data and the address to which the pointer is pointing is copied. Thus, resulting into two objects that are pointing to the same memory location as shown in Figure 4.3a. It must be noted that two objects should have their own distinct identity and distinct memory as shown I Figure 4.3b.

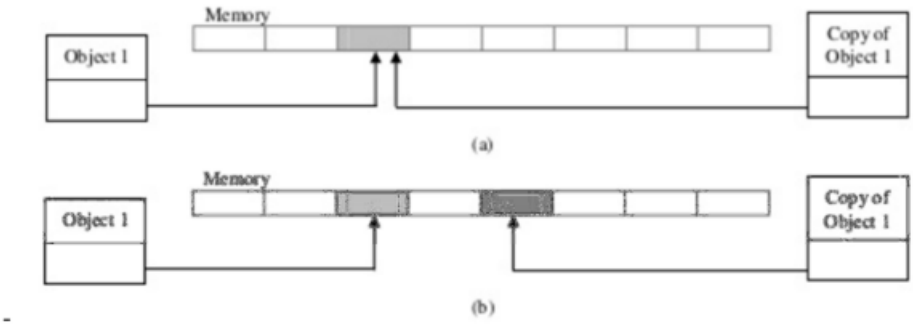


Figure 4.3 – The effect of copy constructors on a pointer data member a) using shallow copy, b) using deep copy

Example: Custom Deep Copy Constructor

```
#include <iostream>
#include <string>
#include <cstring>
using namespace std;
class Student{
private:-
char* name; // dynamic character array
int age;
float gpa;
public:
Student(): name(""),age(0),gpa(0.0) { }
```

```
Student(char n[], int a, float g): name(n),age(a),gpa(g) {}
Student(const Student &s){ // custom copy constructor - Deep
cout << "\nIn Custom Copy Constructor [Deep]" << endl;
int len = strlen(s.name); // step 1: find length of input array
name = new char[len+1]; // step 2: create name of length n + 1 ('/0')
strcpy(name,s.name); // step 3: copy using strcpy
age = s.age;
gpa = s.gpa;
}
~Student(){
cout << "Terminating object." << endl;
delete[] name;
}
void show(){
cout<<"Student Data"<<endl;
cout<<"Name: "<<name<<"\tAddress: "<<(void *)name<<endl;
cout<<"Age: "<<age<<endl;
cout<<"GPA: "<<gpa<<endl<<endl;
}
};
int main(){
Student s1("Ali Ahmad Khan",21,3.5);
s1.show();
Student s2(s1); s2.show();
Student s3 = s1; s3.show();
return 0;
}
```

Output:

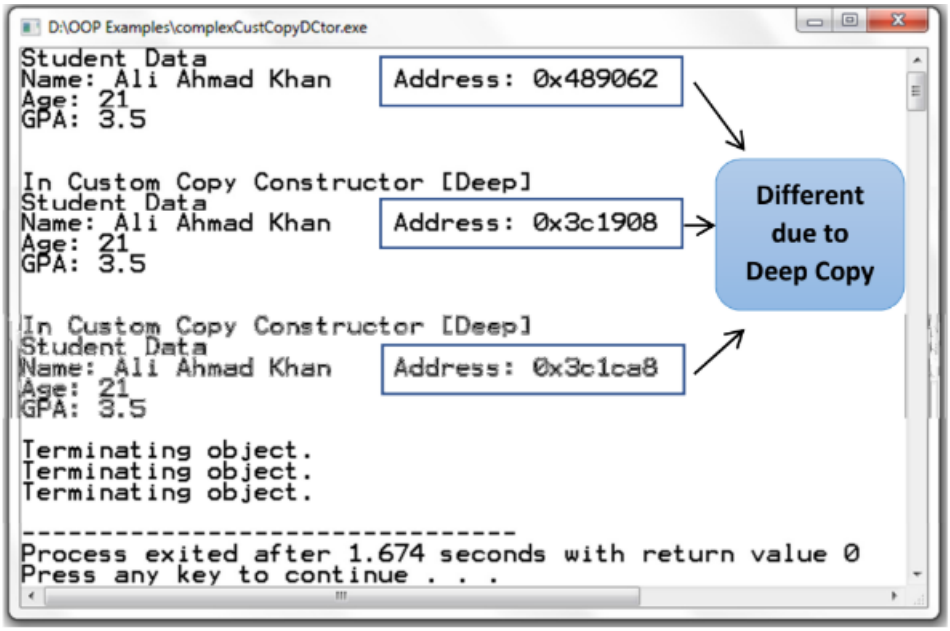


Figure 4.4: Output complexCustCopyDCtor.cpp

In complexCustCopyDCtor.cpp, deep copy of objects s2 and s3 is performed. This is evident from distinct memory locations for each object i.e. 0x3c1908 is address of s2 while 0x3c1ca8 is address of s3.

Now, consider the shallow copy of objects s2 and s3. To do so, copy constructor code must be replaced with the one given below. Output of this change is shown in Figure 4.5.

```
Student(const Student &s){ // custom copy constructor – Shallow
cout << "\nIn Custom Copy Constructor [Shallow]" << endl;
name = s.name;
age = s.age;
gpa = s.gpa;
}
```

Output:

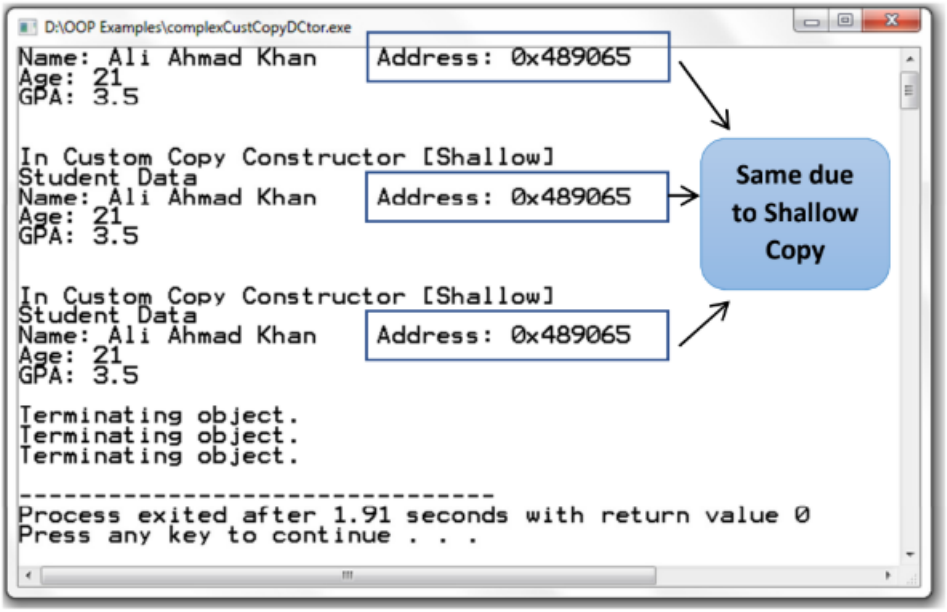


Figure 4.5: Output of changing Deep to Shallow Copy Constructor

It is evident in Figure 4.5 that by performing the shallow copy of s2 and s3, the two objects are pointing to the same memory location i.e. 0x489065. This is invalid as copying a pointer means that the data and the address to which the pointer is pointing is copied. Thus, for pointers deep copy is used as demonstrated in Figure 4.4 while shallow copy otherwise.

'this' Pointer

The this pointer holds the address of current object, in simple words you can say that this pointer points to the current object of the class. Let's take an example to understand this concept.

```
#include <iostream>

using namespace std;

class Demo {

private:

int num;

char ch;

public:

void setMyValues(int num, char ch){

this->num =num;

this->ch=ch;

}
```

```
void displayMyValues(){
cout<<num<<endl;
cout<<ch;
}
};
int main(){
Demo obj;
obj.setMyValues(100, 'A');
obj.displayMyValues();
return 0;
}
```

Static Members in the Class

When we declare a normal variable (data member) in a class, different copies of those data members create with the associated objects. In some cases, when we need a common data member that should be same for all objects, we cannot do this using normal data members. To fulfill such cases, we need static data members. Now we are going to learn about the static data members and static member functions, how they declare, how they access with and without member functions?

C++ static data member

It is a variable which is declared with the static keyword, it is also known as class member, thus only single copy of the variable creates for all objects. A static member is shared by all objects of the class. All static data is initialized to zero when the first object is created, if no other initialization is present. We can't put it in the class definition but it can be initialized outside the class as done in the following example by redeclaring the static variable, using the scope resolution operator :: to identify which class it belongs to. Any changes in the static data member through one member function will reflect in all other object's member functions.

If you are calling a static data member within a member function, member function should be declared as static (i.e. a static member function can access the static data members)

Example

```
#include <iostream>
```

```
using namespace std;
class Box {
public:
// Constructor definition
Box(double l = 2.0, double b = 2.0, double h = 2.0) {
cout <<"Constructor called." << endl;
length = l;
breadth = b;
height = h;
   // Increase every time object is created
      objectCount++;
    }
    double Volume() {
      return length * breadth * height;
    }
    static int getCount() {
      return objectCount;
    }

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

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

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

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

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

  return 0;
}
```

Output

```
Inital Stage Count: 0
Constructor called.
Constructor called.
Final Stage Count: 2
```

Static Member Functions

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

A static member function can only access static data member, other static member functions and any other functions from outside the class. Static member functions have a class scope and they do not have access to the this pointer of the class. You could use a static member function to determine whether some objects of the class have been created or not. Let us try the following example to understand the concept of static function members –

Example

```
#include <iostream>


using namespace std;


class Box {
  public:
    static int objectCount;


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


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

```
    }
    static int getCount() {
      return objectCount;
    }

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

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

int main(void) {
// Print total number of objects before creating object.
cout << "Inital Stage Count: " << Box::getCount() << endl;
Box Box1(3.3, 1.2, 1.5);    // Declare box1
Box Box2(8.5, 6.0, 2.0);    // Declare box2
// Print total number of objects after creating object.
cout << "Final Stage Count: " << Box::getCount() << endl;
return 0;
}
```

Output

```
Inital Stage Count: 0
Constructor called.
```

Constructor called.

Final Stage Count: 2

TASKS:

Task 01: Write a program that creates a class named “english”. The class has a string data member called sentence and another called size that shows the number of characters of the string. Create a constructor that initializes the class objects. Also create a copy constructor that copies the data of one object to the other.

Task 02: Create a class that includes a data member that holds a “serial number” for each object created from the class. That is, the first object created will be numbered 1, the second 2, and so on.

To do this, you’ll need another data member that records a count of how many objects have been created so far. (This member should apply to the class as a whole; not to individual objects. What keyword specifies this?) Then, as each object is created, its constructor can examine this count member variable to determine the appropriate serial number for the new object. Add a member function that permits an object to report its own serial number.

Then write a main() program that creates three objects and queries each one about its serial number. They should respond I am object number 2,and so on.

Task 03

Create a class called time that has separate int member data for hours, minutes, and seconds. One constructor should initialize this data to 0, and another should initialize it to fixed values. Another member function should display it, in 11:59:59 format. The final member function should add two objects of type time passed as arguments. Amain() program should create two initialized time objects (should they be const?) and one that isn’t initialized. Then it should add the two initialized values together, leaving the result in the third time variable. Finally it should display the value of this third variable. Make appropriate member functions const.