

---

## Lab No.4 Deep & Shallow Copy

---

### 4.1 Introduction

Lab 1 introduced basic components of a class such as access specifiers and constructors. In this lab, constructor will be covered in more detail. Constructor is a special function that is automatically called when an object is created. Different types of constructors such as parameterless and parameterized constructors are presented. An object can also be initialized with another object of the same type. This can be done in different ways leading to concept of shallow copy and deep copy in a class. This lab covers both in detail. Also, destructor is introduced. They are called whenever an object is terminated.

### 4.2 Objectives of the lab:

- 1 Understand and implement parameterless and parameterized constructor in a class.
- 2 Write a class (C++/Java) with overloaded constructors.
- 3 Write a test program to use default copy constructor (C++).
- 4 Understand the difference between a Shallow Copy and a Deep Copy.
- 5 Understand the concept of dynamic memory allocation.
- 6 Implement deep and shallow copy in a class (C++/Java).
- 7 Use and test deep and shallow copy in a class.
- 8 Understand and implement destructor in a class (C++/Python).

### 4.3 Pre-Lab

#### 4.3.1 Overloaded Constructors – Parameterized and Nullary Constructors

- 1 Constructor is a special function that is called automatically when an object is created.
- 2 Two types of constructor: Nullary/Parameterless and Parameterized.
  - Nullary/Parameterless Constructors – These do not need parameter for their calling
  - Parameterized Constructors – These need parameters for their calling
- 3 Constructors can be overloaded.
  - Same class can have a nullary constructor alongside a parameterized constructor

#### 4.3.2 Copy Constructor

- 1 It is a function through which a new object can be created from an existing, created object.

- 2 C++ provides a default copy constructor to assist in the copying of simple objects. For example, objects that do not use pointers or arrays.
- 3 Even though, default copy constructor will copy any type of object but it is strongly recommended that the copy constructor be used only for objects that has non-pointer data member.
- 4 Default copy constructor creates an exact copy where data members are copied one by one to the new object. The original object is maintained in its original state and the copy changes are only exhibited in the newly created object.
- 5 Two ways to call default copy constructor:
  - `class obj2(obj1);` // Function calling notation
  - `class obj2=obj1;` // Assignment statement notation

### 4.3.3 Example: Overloaded Constructor and Default Copy Constructor

#### Example code in C++: complexOverCopyCtor.cpp

```
#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) { } // parameterized, overloaded ctor
        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:

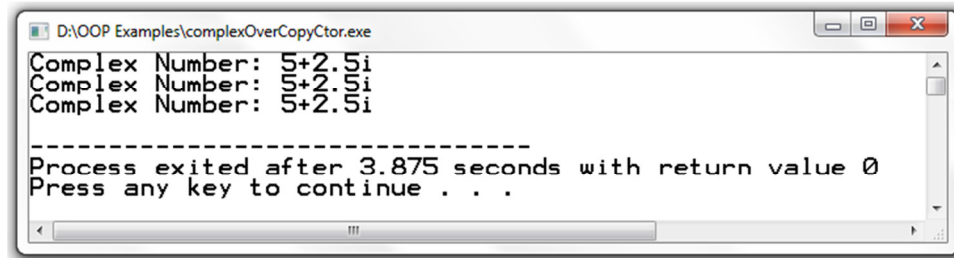


Figure 4.1: Output complexOverCopyCtor.cpp

### 4.3.4 Custom Copy Constructor

- 1 Reasons to create a custom copy constructor:
  - Class objects contain pointers.
  - Class objects contain dynamic data members.
  - Copying of only some of the data member is needed to a new object.
- 2 A copy constructor always receives an object as a parameter, extracts data from it, and stores data in the newly created object.
- 3 Two syntax of copy constructor are:
  - `class (class& obj);` // copy constructor prototype
  - `class (const class& obj);` // const copy constructor prototype
- 4 Note that the object copied is provided by obj. Using const keyword means that a copy of an object can be created without any change to the inherent data members even though only some of the data members can be copied.

### 4.3.5 Shallow Copy Constructor

- 1 It performs member by member copy of one object to another.
- 2 Default copy constructor is shallow.
- 3 When a class does not have any dynamic data members then only a shallow copy constructor is needed. Also, when a partial copy of an object is needed i.e. to copy some of the static data members.

### 4.3.6 Example: Custom Shallow Copy Constructor

#### Example code in C++: complexCustCopySCtor.cpp

```
#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**

```

D:\OOP Examples\complexCustCopySctor.exe
Complex Number: 5+2.5i
In Custom Copy Constructor [Shallow]
Complex Number: 5+2.5i
In Custom Copy Constructor [Shallow]
Complex Number: 5+2.5i
-----
Process exited after 2.144 seconds with return value 0
Press any key to continue . . .

```

### 4.3.7 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.

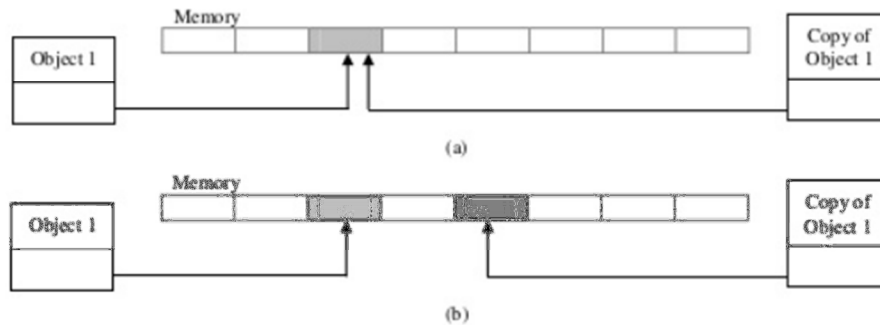


Figure 4.3 – The effect of copy constructors on a pointer data member a) using shallow copy, b) using deep copy

### 4.3.8 Example: Custom Deep Copy Constructor

#### Example code in C++: complexCustCopyDCtor.cpp

```
#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 << "\n\n 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:**

```

D:\OOP Examples\complexCustCopyDctor.exe
Student Data
Name: Ali Ahmad Khan   Address: 0x489062
Age: 21
GPA: 3.5

In Custom Copy Constructor [Deep]
Student Data
Name: Ali Ahmad Khan   Address: 0x3c1908
Age: 21
GPA: 3.5

In Custom Copy Constructor [Deep]
Student Data
Name: Ali Ahmad Khan   Address: 0x3c1ca8
Age: 21
GPA: 3.5

Terminating object.
Terminating object.
Terminating object.

-----
Process exited after 1.674 seconds with return value 0
Press any key to continue . . .

```

**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

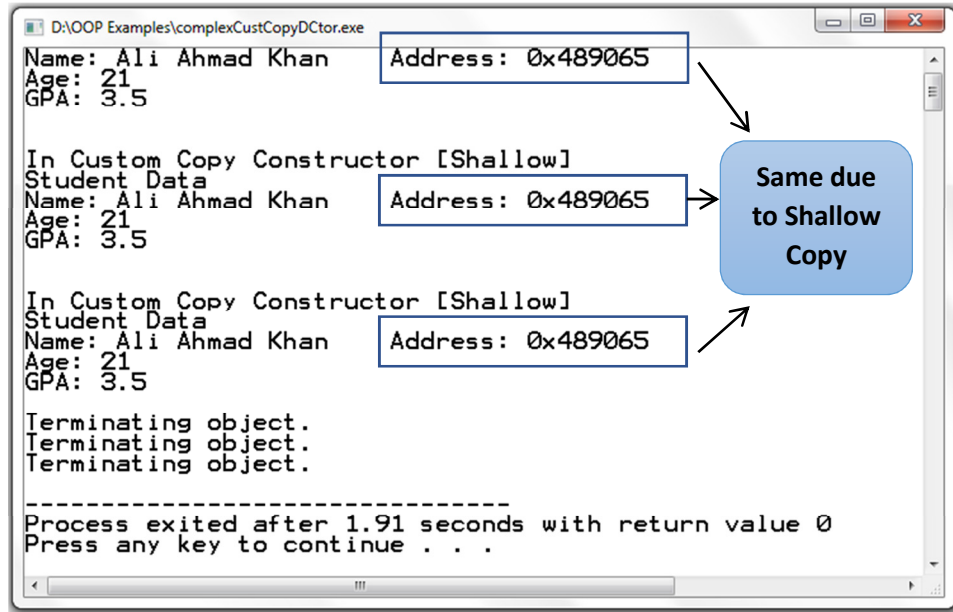
```

```

        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.

### 4.3.9 Example in Java

- 1 There is no copy constructor in java. But, an object can be copied into another one like copy constructor in C++.
- 2 There are three ways to do so:
  - By parameterized constructor taking an object.
  - By assigning the values of one object to another
  - By clone() method of Object class – this way is used for shallow and deep copy.
- 3 These concepts are discussed in the following example:

#### Address.java

```

package lab4s1;

```

```

public class Address {
    public String City;                // made public to access in main() function
    private String State, PC;
    public Address(String c, String s, String p){    // parameterized constructor
        this.City = c;
        this.State = s;
        this.PC = p;
    }
    public void setCity(String c) { this.City = c; }    // sets city
    public String getCity() { return City; }            // returns city
    public void setState(String s) { this.State = s; }  // sets state
    public String getState() { return State; }          // returns state
    public void setPC(String p) { this.PC = p; }        // sets postal code
    public String getPC() { return PC; }               // returns postal code
}

```

## Student.java

```

package lab4s1;

public class Student implements Cloneable{
    private int id;
    private String name;
    public Address adr;

    public Student() {}    // Null/Parameterless Constructor

    public Student(int i, String n, Address a){    // Parameterized, Overloaded Constructor
        System.out.println("In Parametrized, Overloaded Constructor");
        this.id = i;
        this.name = n;
        this.adr = a;
    }

    public Student(Student s) {
        System.out.println("In Custom Copy Constructor");
        this.id = s.id;
        this.name = s.name;
        this.adr = s.adr;
    }

    public void setstdAddress(Address a){

```



```
    this.adr = a;
}
```

```
// Deep Cloning in Java
```

```
@Override
```

```
public Object clone() throws CloneNotSupportedException{
    Student stdClone = (Student) super.clone();
    Address stdAddressClone = new Address(this.adr.getCity(),
                                           this.adr.getState(),
                                           this.adr.getPC());

    stdClone.setstdAddress(stdAddressClone);
    return stdClone;
}
```

Step 2

```
public void show() {
    System.out.println("ID: "+this.id);
    System.out.println("Name: "+this.name);
    System.out.println("Address:"+this.adr.getCity()+" "+this.adr.getState()+" "+this.adr.getPC());
}
}
```

## Test.java

```
package lab4s1;
```

```
public class Test {
```

```
    public static void main(String[] args){
```

```
        Address a1 = new Address("Peshawar", "KPK", "25000");
```

```
        System.out.println("s1");
```

```
        Student s1 = new Student(1011, "Ali Ahmad", a1);
```

```
        s1.show();
```

```
        // no copy constructor in java
```

```
        // but copy object contents in three ways
```

```
        // 1. assigning values of one object to another
```

```
        System.out.println("ns2");
```

```
        Student s2 = s1;
```

```
        s2.show();
```

```
        // 2. by constructor
```

```
        System.out.println("ns3");
```

```

Student s3 = new Student(s1);
s3.show();

// 3. by clone method of Object class
// deep cloning and shallow cloning
System.out.println("\ns4");

Student s4 = null;
try {
    s4 = (Student) s1.clone();
    s4.show();
} catch (CloneNotSupportedException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}

System.out.println("\ns1 after modifying City: ");
s1.adr.City = "Islamabad";
s1.show();

System.out.println("\ns4 after modifying s1's City: ");
s4.show();
}
}

```

Step 3

In this example, three classes are formed: Student, Address, and Test. Student is the key class storing student id, name and address. As address is combination of City, State, and Zip Code, it is represented by a separate class. Test class is used to test the two classes via main ( ) function.

Address class consists of three data members i.e. City, State, and Zip Code. It consists of three-argument, parameterized constructor to initialize the three data members. Also, set and get function is provided for each data member.

Student class consists of three data members i.e. id, name, and address. It contains null/Parameterless constructor, parameterized/overloaded constructor, and a copy constructor. It also includes setstdAddress() and show() functions. To perform deep copy, java uses the clone ( ) function of Object class. In this example, deep cloning is performed and tested as depicted by step 1, step2, and step 3.

Address and Student objects are created inside main() in Test class. Output of example is shown in Figure 4.6. s1 object is created using parameterized constructor. Two more objects s2 and s3 are created using object s1. s2 is formed using assignment operator

while s3 is created using copy constructor. s4 is created using deep cloning of s1. Each object contents are showed using show() function. Lastly, the City of s1 is altered. It can be seen that since s4 is deep clone and has its own memory thus its content will stay same after modifications in s1 as highlighted in Figure 4.6.

## Output:

```
<terminated> Test (6) [Java Application] C:\Program Files\Java\jdk1.8.0_141\bin\javaw.exe (Oct 16, 2018, 9:18:29 PM)

s1
In Parametrized, Overloaded Constructor
ID: 1011
Name: Ali Ahmad
Address: Peshawar KPK 25000

s2
ID: 1011
Name: Ali Ahmad
Address: Peshawar KPK 25000

s3
In Custom Copy Constructor
ID: 1011
Name: Ali Ahmad
Address: Peshawar KPK 25000

s4
ID: 1011
Name: Ali Ahmad
Address: Peshawar KPK 25000

s1 after modifying City:
ID: 1011
Name: Ali Ahmad
Address: Islamabad KPK 25000
s4 after modifying s1's City:
ID: 1011
Name: Ali Ahmad
Address: Peshawar KPK 25000

Different City due to Deep Cloning
```

Figure 4.6: Output Address.java, Student.java & Test.java

Now, consider the shallow cloning of objects s4. To do so, clone() function code must be replaced with the one given below. Output of this change is shown in Figure 4.7.

```
@Override
public Object clone() throws CloneNotSupportedException{ //Shallow Cloning in Java
    return super.clone();
}
```

It is evident in Figure 4.7 that by performing the shallow cloning of s4, the contents changes once s1 is modified. City of s1 is altered and correspondingly the city of s4 also changed.

## Output:

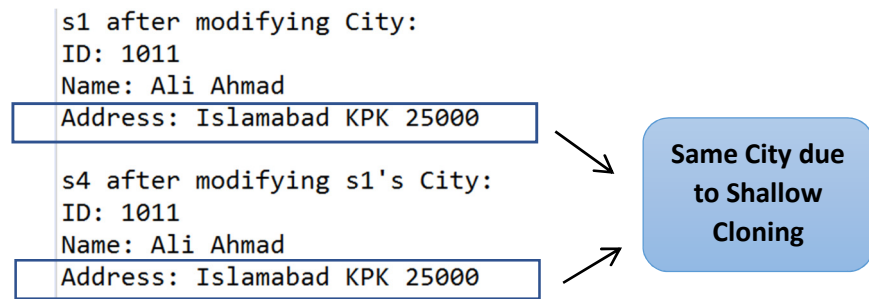


Figure 4.7: Output of changing Deep to Shallow Cloning

### 4.3.10 Example in Python

- 1 There are two types of constructors in Python: default constructor or parameterized constructor.
- 2 A single class will have either default constructor or parameterized constructor; not both. If multiple constructors are part of a class, then only the last one will be considered and used by objects.
- 3 There is no copy constructor in java. But, an object can be copied into another one like copy constructor in C++. To do so, copy class is imported and its member functions are used. copy class provides member function for both shallow copy and deep copy of an object.
- 4 Each of these concepts are discussed in the following example:

```
import copy

class Student:
    def __init__(self,i=0,n="",g=0.0):
        print "In Parametrized Constructor"
        self.id = i
        self.name = n
        self.gpa = g
    def __del__(self):
        print ("destructor")
    def show(self):
        print self.id," ",self.name," ",self.gpa

s1 = Student(1011,"Ali Ahmad",3.4)
s1.show()

s2 = copy.copy(s1)           # shallow copy
s2.show()
```

```

s3 = copy.deepcopy(s1)          # deep copy
s3.show()

del s1
del s2
del s3

```

## Output:

```

Python Interpreter
*** Python 2.7.10 (default, May 23 2015, 09:44:00) [MSC v.1500 64 bit (AMD64)] on win32. ***
*** Remote Python engine is active ***
>>>
*** Remote Interpreter Reinitialized ***
>>>
In Parametrized Constructor
1011  Ali Ahmad  3.4
1011  Ali Ahmad  3.4
1011  Ali Ahmad  3.4
destructor
destructor
destructor

```

**Figure 4.8: StdCopy.py**

In this example, Student class is written in python. It consists of a parameterized constructor, a destructor, and a show() functions. copy module is imported and two of its functions are used to perform shallow and deep copy. To demonstrate, three objects s1, s2, and s3 are created. s1 is created using parameterized constructor, s2 is created using s1 by shallow copy, and s3 is created using s1 by deep copy. Output of each object is shown. In the end, destructor is used to remove the objects.

Note that in this example, built-in copy.copy() and copy.deepcopy() functions are just used. These can also be tailor made and adjusted according to one's need. A custom shallow and deep copy in Python can also be implemented.

## 4.5 Activities

**Perform these activities in C++, Java and Python.**

### 4.5.1 Activity

- a) **C++:** Create a class called employee. This class maintains information about name (char\*), department (char\*), salary (double), and period of service in years (double).
  1. Provide a parameterless constructor to initialize the data members to some fixed values.
  2. Provide a 4-argument parameterized constructor to initialize the members to values sent from calling function.
    - a. (You have to make dynamic allocation for both name and department data members in constructor.)

3. Provide a copy-constructor that performs the deep copy of the data members.
4. Provide an input function that takes all the values from user during run-time.
5. Provide a show function that shows all the information about a specific employee to user.
6. Provide a destructor to free the memory allocated to name and department in constructor.

Write all the member function outside a C++ class. Write a driver program to test the functionality of the above-mentioned class.

b) Java: Create a class called employee. This class maintains information about name (String), department (String), salary (double), and period of service in years (double).

1. Provide a parameterless constructor to initialize the data members to some fixed values.
2. Provide a 4-argument parameterized constructor to initialize the members to values sent from calling function.
3. Provide an input function that takes all the values from user during run-time using Scanner.
4. Provide a deep clone() that performs the deep copy of the data members.
5. Provide a show function that shows all the information about a specific employee to user.

Write a Test class to test the functionality of the above-mentioned class using Test Case given in 4.6. Next, change the department of e1 to EE and view contents of e1 and e3. Change the clone() function to shallow and observe the output of e1 and e3 after changing the department. Write your observation for both shallow & deep clone and compare it with Java Example in Section 4.3.9. If the Address class is to be used in employee, how shallow & deep clone function will behave? And why?

c) Python: Import copy module. Create a class called employee.

1. Provide a 4-argument parameterized constructor to initialize the members to values sent from calling function.
2. Provide an input function dataIn() that takes all the values from user during run-time.
3. Provide a show function that shows all the information about a specific employee to user.

Test the functionality of the above-mentioned class using Test Case given in 4.6.

## 4.6 Testing

### Test Cases for Activity 4.5.1a)

Sample Inputs	Sample Outputs
Declare Employee Object e1	

Take e1 values by calling input()	Enter Employee Name: Hasan Waqar Enter Employee Dept.: DCSE Enter Employee Salary: 25000 Enter Employee Service Years: 3
Display the content of e1 using show()	Employee Name: Hasan Waqar Employee Dept.: DCSE Employee Salary: 25000 Employee Service Years: 3
Declare Employee Object e2 and copy e1 in it using assignment operator Display the content of e2 using show()	Employee Name: Hasan Waqar Employee Dept.: DCSE Employee Salary: 25000 Employee Service Years: 3
Declare Employee Object e3 and copy e1 in it using function notation Display the content of e3 using show()	Employee Name: Hasan Waqar Employee Dept.: DCSE Employee Salary: 25000 Employee Service Years: 3

#### Test Cases for Activity 4.5.1b)

Sample Inputs	Sample Outputs
Declare Employee Object e1 and e2 same as given in 4.5.1a)  Declare Employee Object e3 Clone contents of e1 into e3 Display the content of e3 using show()	Employee Name: Hasan Waqar Employee Dept.: DCSE Employee Salary: 25000 Employee Service Years: 3

#### Test Cases for Activity 4.5.1c)

Sample Inputs	Sample Outputs
Declare Employee Object e1 Assign data to e1 using dataIn() Display the content of e1 using show()	Hasan Waqar DCSE 25000 3
Declare and assign shallow copy of e1 to e2 Display the content of e2 using show()	Hasan Waqar DCSE 25000 3
Declare and assign deep copy of e1 to e3 Display the content of e3 using show()	Hasan Waqar DCSE 25000 3

## 4.7 References:

1. Class notes
2. Object-Oriented Programming in C++ by *Robert Lafore* (Chapter 6)
3. How to Program C++ by *Deitel & Deitel* (Chapter 7)
4. Programming and Problem Solving with Java by *Nell Dale & Chip Weems*
5. Murach's Python Programming by *Micheal Urban & Joel Murach*