SRM UNIVERSITY AP
Amaravati

Object Oriented Programming using C++

CSE-III

CSE206-OOP

Dr. Priyanka Singh

UNIT II: FEATURES OF OBJECT-ORIENTED PROGRAMMING

Introduction to Classes and Objects, Making sense of core object concepts (Encapsulation, Abstraction, Polymorphism, Classes, Messages Association, Interfaces). Constructors and its types, Destructors - Passing Objects as Function arguments and Returning Objects from Functions.

Unitization Plan

FEATURES OF OBJECT-ORIENTED PROGRAMMING	9 hrs.
Concept of classes and objects with real world examples	1
Encapsulation, data hiding using storage classifier	1
Polymorphism, Types of polymorphism, Use-cases	1
Method overloading, Method overriding	1
Virtual functions	1
Interfaces	1
Constructors and destructors	1
Methods, Method calling, Method with object parameters	1
Summary, Putting it all together with hands-on	1

Thinking OOP Way: Real World Examples

Old Saying in Carpentry

“Measure twice cut once”

Quote of the Day for Coders

“Think twice, implement once”

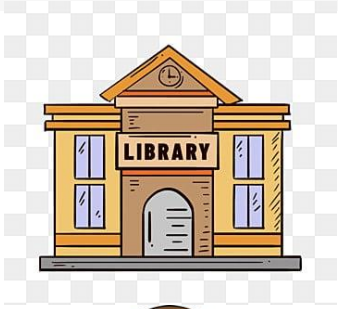
“Plan before you act”
Design comes before implementation.

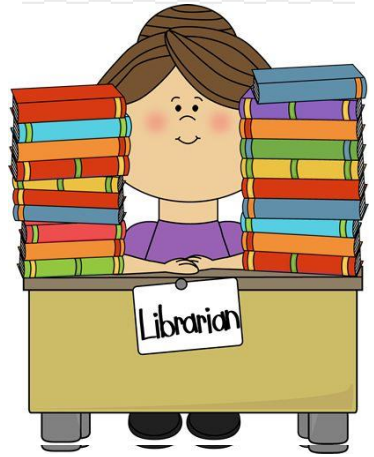
Real World Systems

Library: (Simple?)
- **Books**
- **Librarian**
- **Users**

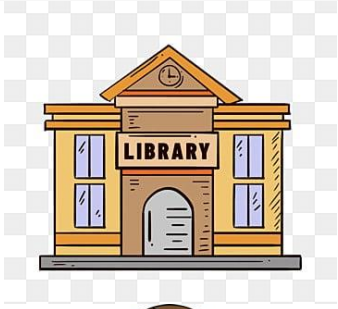
Library: (Slightly Complicated)
- **Different types of books**
- **Different types of users**

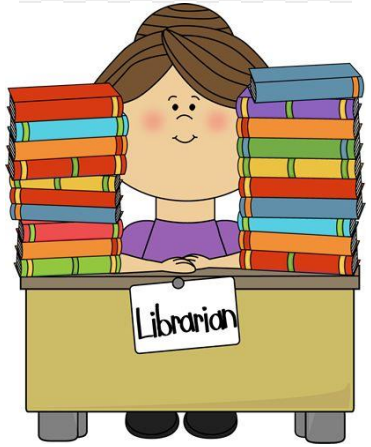
Library: (Moderately Complicated)
- **Book search**
- **Genre, Edition, Author etc.**
- **Account Management**
- **Issue dates, Due dates, Fines, Limit etc.**





Real World Systems





Procedural way of thinking

- **Identify modules**
- **Write pseudo-code for each module**

Draw flow charts

Connect/integrate flow charts together

Resolve inconsistencies during integration

Real World Systems

- **Real world is very complicated.**
- **Visualizing the end-to-end system is difficult**

- **Solutions:**
  - **Identify the group of entities/elements**
  - **Identify the properties of these entities**
  - **Identify actions**

  Represent them pictorially (How??)

- **Let’s take one step at a time.**

Real World Systems

Identify the group of entities/elements

- **Users**
- **Librarians**
- **Book**
- **Account**

Identify the properties of these entities

Users:

- **Role: {Student, Staff, Faculties}**
- **Borrow Limit: {no. of Books}**
- **Borrow Duration Limit: {Days}**
- **Fine:{Rs}**
- **Categories:{Types of books permitted to borrow}**

Librarians:

- **Role: {Admin, Assistant}**
- **Working Hours: {hrs}**

Book:

- **Genre:{Science, Fiction, Text}**
- **Title:{}**
- **Authors:{}**

Account

Identify actions

Users can:

Borrow books, Return books, Pay fines, Search books

Librarians:

Issue books, Deposit, Enter new user, Delete old user, Enter book, Delete lost books

Book:

Show status, Show due date

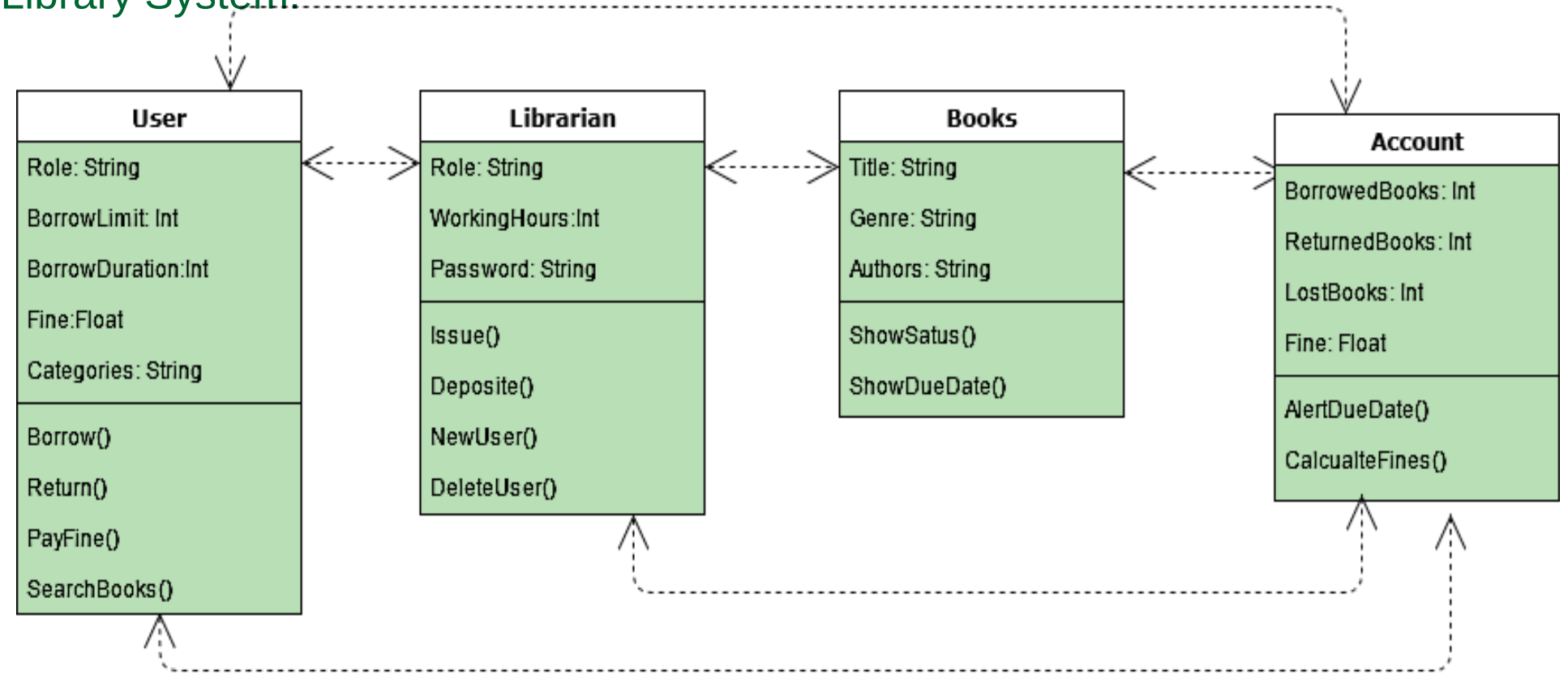
Account:

Alert due date, Calculate fines

Represent them pictorially (How??)

Unified Modelling Language (UML): Class Diagram (Briefly) & Object Diagram (May be Later)
Note: https://app.diagrams.net/ → “Create Diagram” → Select “Software” → Select “Class”

Class Diagram for Library System:



Real World Classes & Objects

Class:
Properties:

Characteristics of Objects

Identity: makes an object different from other objects created from the same class.

State:

Defined by the contents of an object's attributes.

Objects state vary during the execution of program.

Behavior: Defined by the messages (functions/methods) an object provides.

Object Life Cycle

- Object Creation
- Object Usage
- Object Disposal

Object Creation

Objects are created by instantiating a class.

```
ClassName object = new ClassName();
```

Object creation involves three actions:

Declaration: Declaring the type of data

Instantiation: Creating a memory space to store the object

Initialization: Initializing the attributes in the object.

Class

```
class className
{
 // some data
// some functions
 };
```

```
class Room {
    public:
        double length;
        double breadth;
        double height;

        double calculateArea(){
            return length * breadth;
        }

        double calculateVolume(){
            return length * breadth * height;
        }

};
```

Example

```
#include <iostream>
using namespace std;
// create a class
class Room {
   public:
    double length;
    double breadth;
    double height;
    double calculateArea() {
        return length * breadth;
    }
    double calculateVolume() {
        return length * breadth * height;
    }
};
```

```
int main() {
    // create object of Room class
    Room room1;
    // assign values to data members
    room1.length = 42.5;
    room1.breadth = 30.8;
    room1.height = 19.2;

    // calculate and display the area and volume of the room
    cout << "Area of Room =  " << room1.calculateArea() << endl;
    cout << "Volume of Room =  " << room1.calculateVolume() << endl;

    return 0;
}
```

Classes and Objects

- The fundamental ideas behind classes are data abstraction and encapsulation.
- Data abstraction lets us separate the implementation of an object from the operations that that object can perform (i.e. separation of interface and implementation).
- The interface of a class consists of the operations that users of the class can execute. The implementation includes the class' data members, the bodies of the functions that constitute the interface, and any functions needed to define the class that are not intended for general use.
- **Encapsulation** enforces the separation of a class' interface and implementation. A class that is encapsulated hides its implementation—users of the class can use the interface but have no access to the implementation.

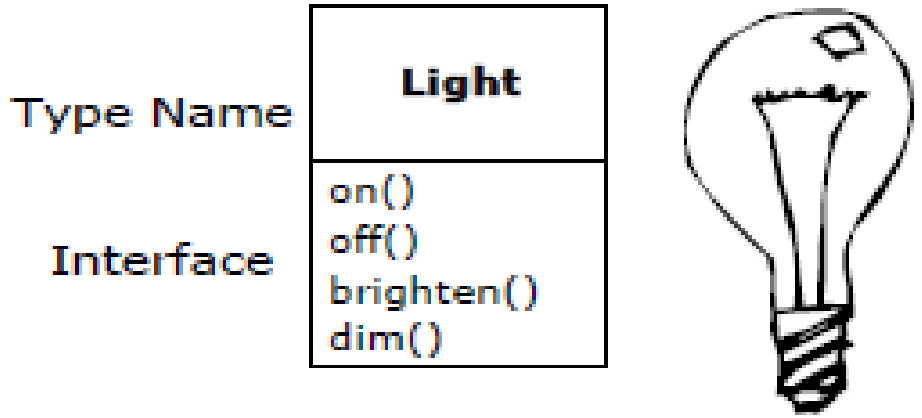
Classes and Objects

- In C++ we use classes to define our own data types. By defining types that mirror concepts in the problems we are trying to solve, we can make our programs easier to write, debug, and modify.

- A class that uses data abstraction and encapsulation defines an abstract data type. In an abstract data type, the class designer worries about how the class is implemented. Programmers who use the class need not know how the type works. They can instead think abstractly about what the type does.

- We'll learn how to control what happens when objects are copied, moved, assigned, or destroyed.

Interface and implementation




```
Light lt;
lt.on();
```

- The interface establishes what requests you can make for a particular object.
- However, there must be code somewhere to satisfy that request. This, along with the hidden data, comprises the implementation.

Interface and implementation

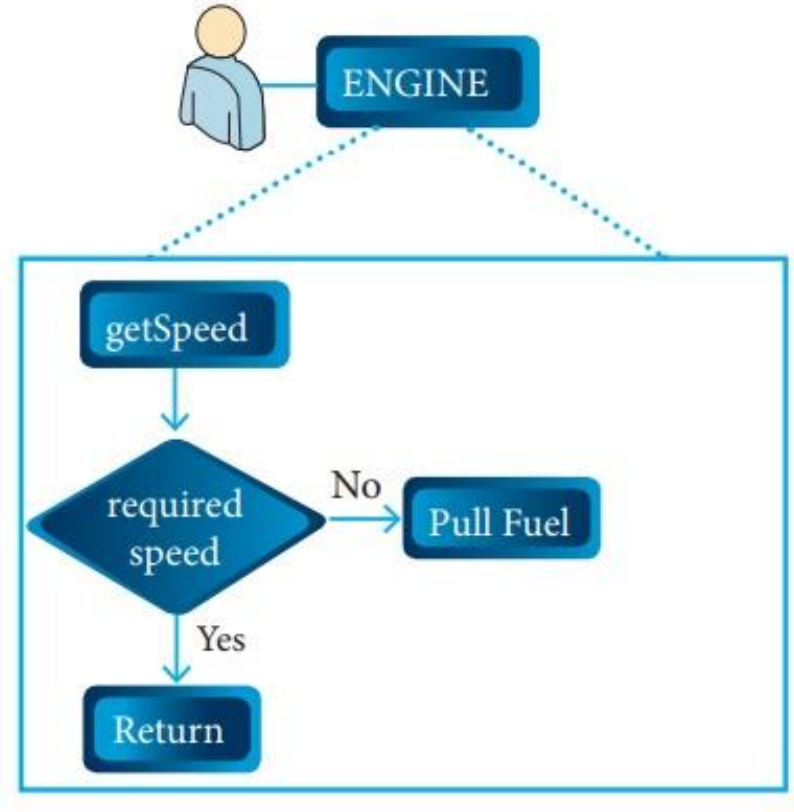
- An interface is a set of action that an object can do. For example, when you press a light switch, the light goes on, you may not have cared how it splashed the light.
- In OOPs language, an Interface is a description of all functions that a class must have in order to be a new interface. In our example, anything that "ACTS LIKE" a light, should have function definitions like turn_on () and a turn_off (). The purpose of interfaces is to allow the computer to enforce the properties of the class of TYPE T (whatever the interface is) must have functions called X, Y, Z, etc.

Interface	Implementation
Interface just defines what an object can do, but won't actually do it	Implementation carries out the instructions defined in the interface

Interface and implementation

- The person who drives the car doesn't care about the internal working. To increase the speed of the car he just presses the accelerator to get the desired behavior. Here the accelerator is the interface between the driver (the calling / invoking object) and the engine (the called object).

- In this case, the function call would be Speed (70): This is the interface.



Example Interface

Array based stack	List based stack

Although the internal structure of the two Stack classes is the same, the external interface is identical. So, a program needing a Stack could use either.

```
class Stack{
private:
  int values[MAXSTACK];
  int top;

public:
  Stack();
  ~Stack();

  void clear();

  bool empty() const;

  void push(int newvalue);

  int pop();
};
```

```
class Stack{
private:
  Entry *top;

public:
  Stack();
  ~Stack();

  void clear();

  bool empty() const;

  void push(int newvalue);

  int pop();
};
```

Example Implementation

Array based stack

List based stack

Although the interfaces of the two classes are identical, the implementation is quite different. A user of the class would not see any difference.

```
// push a new value onto the stack
void Stack::push(int newvalue){
  if(top < MAXSTACK)
    values[top++] = newvalue;
  else{
    cerr << "stack overflow" << endl;
    exit(1);
  }
}

// return a value from the top of the stack
int Stack::pop(){
  if(!empty())
    return values[--top];
  else{
    cerr << "stack underflow" << endl;
    exit(2);
  }
}
```

```
// push a new value onto the stack
void Stack::push(int newvalue){
  Entry *newentry = new Entry;

  newentry->value = newvalue;
  newentry->next = top;

  top = newentry;
}

// return a value from the top of the stack
int Stack::pop(){
  Entry *topentry;
  int topvalue;

  if(!empty()){
    topentry = top;
    topvalue = topentry->value;

    top = topentry->next;
    delete topentry;

    return topvalue;
  }
  else{
    cerr << "stack underflow" << endl;
    exit(2);
  }
}
```

Example Implementation

Array based stack **List based stack**

Because the interfaces are identical, this code will run without any modifications using either Stack implementation.

```
#include <iostream>
#include "Stack.h"
using namespace std;

int main(){
  Stack mystack;
  int num;

  cin >> num;
  while(!cin.eof()){
    mystack.push(num);
    cin >> num;
  }

  while(!mystack.empty()){
    num = mystack.pop();
    cout << num << ' ';
  }
  cout << endl;

  return 0;
}
```

empty() function is **used to check whether the set container is empty or not**.

Constructors

- A constructor is a special type of member function that is called automatically when an object is created.
- In C++, a constructor has the same name as that of the class and it does not have a return type. For example,

```
class  Wall {
  public:
    // create a constructor
    Wall() {
      // code
    }
};
```

- **Note:** If we have not defined a constructor in our class, then the C++ compiler will automatically create a default constructor with an empty code and no parameters.

Initialization

Constructors

```
class Book{
    public:
    String Name;
    // constructor
    Book(){
     // called during object
       initialization
    }
};
```

Constructors

```
class Book
{
    public:
    String Name;
    Book();// constructor declared
};
// constructor definition
Book::Book(){
    i = 1;
}
```

Initialization

Default Constructors

```
class Book{
    public:
    String Name;
    // constructor
    Book()
    {
     Name=”Default”;
    }
};
```

Parameterized Constructors

```
class Book{
    public:
    String Name;
    Book(String n)
    {
      Name=n;
     }
};
```

Example

```
#include <iostream>
using namespace std;
// declare a class
class  Wall {
  private:
    double length;
  public:    // default constructor to initialize variable
    Wall() {
      length = 5.5;
      cout << "Creating a wall." << endl;
      cout << "Length = " << length << endl;
    }
};
int main() {
  Wall wall1;
  return 0;
}
```

Example

```
#include <iostream>
using namespace std;
// declare a class
class Wall {
  private:
    double length;
    double height;
  public:
  // parameterized constructor to initialize variables
    Wall(double len, double hgt) {
      length = len;
      height = hgt;
    }
    double calculateArea() {
      return length * height;
    }
};
```

```
int main() {
  // create object and initialize data members
  Wall wall1(10.5, 8.6);
  Wall wall2(8.5, 6.3);

  cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
  cout << "Area of Wall 2: " << wall2.calculateArea();

  return 0;
}
```

Copy constructor

The copy constructor in C++ is used to copy data of one object to another. A copy constructor is a member function that initializes an object using another object of the same class. A copy constructor has the following general function prototype:

ClassName (const ClassName &old_obj);

```
Wall(Wall &obj)
{
length = obj.length;
height = obj.height;
}
```

Copy Constructors

```
class Book{
 public:
      String Name;
      Book(){
          Name=”Default”;
      }
      Book(const Book& b)
      {
          Name=b.Name;
      }
};…
Book b2 = b1;// Copy Constr.
Book t1, t2;
t1=t2;      // Assignment
```

Example

```
#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 &p1) {x = p1.x; y = p1.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

             cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
             cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
             return 0;
}
```

Example

```
#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 &p) {y = p.y;}
           int getX()                    { return x; }
           int getY()                    { return y; }
};
```

```
int main()
{
Point p1(10, 15); // Normal constructor is called here
Point p2 (20, 30);
Point p3 = p1; // Copy constructor is called here
cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
cout << "\np2.x = " << p2.getX() << ", p2.y = " << p2.getY();
cout << "\np3.x = " << p3.getX() << ", p3.y = " << p3.getY();
return 0;
}
```

When is copy constructor called?

In C++, a Copy Constructor may be called in the following cases:

1. When an object of the class is returned by value.

2. When an object of the class is passed (to a function) by value as an argument.

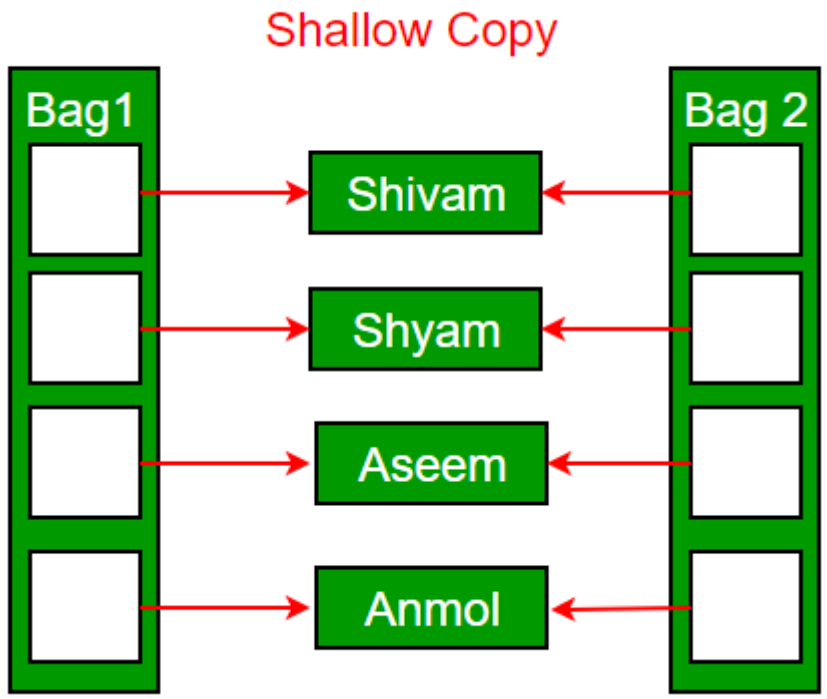
3. When an object is constructed based on another object of the same class.

4. When the compiler generates a temporary object.

When is a user-defined copy constructor needed?

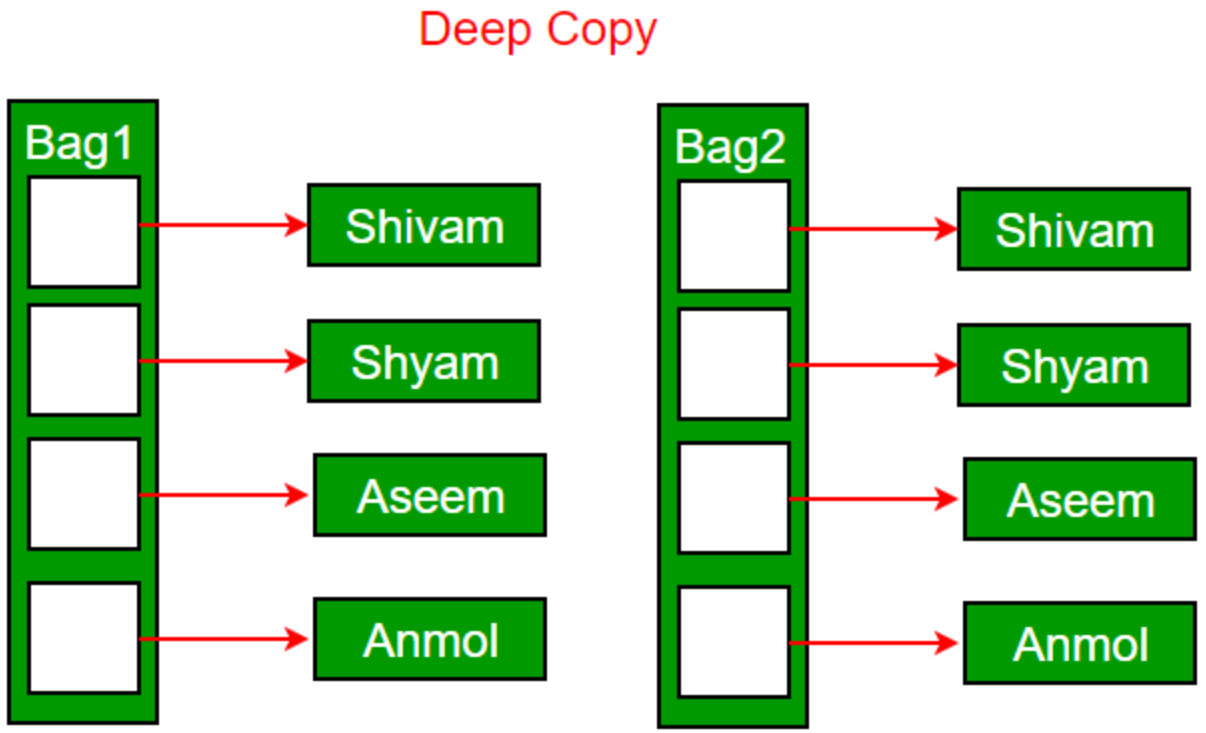
If we don’t define our own copy constructor, the C++ compiler creates a default copy constructor for each class which does a member-wise copy between objects.

- The default ***constructor does only shallow copy.***



Cont…

- ***Deep copy is possible only with user defined copy constructor.*** In user defined copy constructor, we make sure that pointers (or references) of copied object point to new memory locations.



Copy constructor vs Assignment Operator

- Which of the following two statements call copy constructor and which one calls assignment operator?

```
MyClass t1, t2;
MyClass t3 = t1;  // ----> (1)
t2 = t1;          // -----> (2)
```

- Copy constructor is called when a new object is created from an existing object, as a copy of the existing object. Assignment operator is called when an already initialized object is assigned a new value from another existing object. In the above example (1) calls copy constructor and (2) calls assignment operator.

Self learning

- Can we make copy constructor private?
- Why argument to a copy constructor must be passed as a reference?
- Why argument to a copy constructor should be const?