Object Oriented Programming Lab

Assignment 5

Submitted by:

Navdeep Singh

26th August 2025

Roll No: 24124073
Group: 3
Branch: Information Technology
Year: 2nd Year

Practice Question to parctice about copy constructer shallow copy and deep copy

Code

```cpp
#include <bits/stdc++.h>
using namespace std;

// ------------------- SHALLOW COPY -------------------
class Points1 {
public:
    int *x;

    // Constructor
    Points1(int a = 0) {
        x = new int(a);
        cout << "Constructor called, *x = " << *x << endl;
    }

    // Default copy constructor (SHALLOW COPY)
    // Compiler-generated: just copies the pointer address (not the value itself)

    void setValue(int val) {
        *x = val;
    }

    void display() {
        cout << "Value = " << *x << " | Address = " << x << endl;
    }

    // Destructor
    ~Points1() {
        cout << "Destructor called for *x = " << *x << endl;
        delete x;
    }
};

// ------------------- DEEP COPY -------------------
class Points2 {
public:
    int *x;

    // Constructor
    Points2(int a = 0) {
        x = new int(a);
        cout << "Constructor called, *x = " << *x << endl;
    }

    // User-defined Copy Constructor (DEEP COPY)
    Points2(const Points2 &p) {
        x = new int(*p.x);  // allocate new memory and copy value
        cout << "Deep Copy constructor called, *x = " << *x << endl;
    }

    void setValue(int val) {
        *x = val;
    }

    void display() {
        cout << "Value = " << *x << " | Address = " << x << endl;
    }

    // Destructor
    ~Points2() {
        cout << "Destructor called for *x = " << *x << endl;
        delete x;
    }
};

int main() {
    cout << "===== Shallow Copy Example =====" << endl;
    Points1 p1(10);
    Points1 p2 = p1;  // shallow copy (pointer copied, not value)

    cout << "Before change:" << endl;
    p1.display();
    p2.display();

    p1.setValue(20);  // changing p1 also changes p2 because both share same memory

    cout << "After change in p1:" << endl;
    p1.display();
    p2.display();

    cout << "\n===== Deep Copy Example =====" << endl;
    Points2 q1(30);
    Points2 q2 = q1;  // deep copy (separate memory)

    cout << "Before change:" << endl;
    q1.display();
    q2.display();

    q1.setValue(40);  // changing q1 does NOT affect q2

    cout << "After change in q1:" << endl;
    q1.display();
    q2.display();

    return 0;
}
```

Sample Output

```
===== Shallow Copy Example =====
Constructor called, *x = 10
Before change:
Value = 10 | Address = 0x600003e40
Value = 10 | Address = 0x600003e40   <-- same address (shared)
After change in p1:
Value = 20 | Address = 0x600003e40
Value = 20 | Address = 0x600003e40   <-- both changed (shallow copy problem)
```

```

===== Deep Copy Example =====
Constructor called, *x = 30
Deep Copy constructor called, *x = 30
Before change:
Value = 30 | Address = 0x600004120
Value = 30 | Address = 0x600004140   <-- different address
After change in q1:
Value = 40 | Address = 0x600004120
Value = 30 | Address = 0x600004140   <-- q2 unaffected
```

1 Assignment 5 V1

Q1. Write a C++ program to define a class named Tracker that performs the following operations:

- All data members should be public, except for `count` and `nextId`, which should be private static members.
- Define a parameterized constructor that:
  - Assigns a unique id to each object.
  - Increments the active object count.
  - Prints: `"Constructor called.  ID = X, Count = Y"`, where `X` is the object's id and `Y` is the current count.
- Define a copy constructor that:
  - Assigns a new unique id to the copied object.
  - Increments the active object count.
  - Prints: `"Copy constructor called for id = X (copied from id = Y)"`, where `X` is the new object's id and `Y` is the original object's id.
- Define a destructor that:
  - Decrements the active object count.
  - Prints: `"Destructor called for id = X, Count before destruction = Y"`, where `X` is the object's id and `Y` is the count before destruction.
- Define a function `createTracker()` that creates and returns a `Tracker` object by value.
- Define a function `takeTracker()` that accepts a `Tracker` object by value.

In the `main()` function perform the following:

1. Create two `Tracker` objects using the constructor.
2. Copy one of the objects using the copy constructor.
3. Call `createTracker()` and assign the returned object to a variable.
4. Pass one of the objects to `takeTracker()` (copy constructor should be called).

5. Create a block scope {} and define two `Tracker` objects inside it.
6. After each major step, print the current active instance count.

Code

```
#include <iostream>
using namespace std;

class Tracker {
public:
    int id;
    string name;

private:
    static int count;     // number of active objects
    static int nextId;    // gives a unique id to each new object

public:
    // constructor
    Tracker(string n) {
        id = nextId++;
        name = n;
        count++;
        cout << "Object created: " << name << " (id = " << id << ")\n";
    }

    // copy constructor
    Tracker(const Tracker &other) {
        id = nextId++;
        name = other.name;
        count++;
        cout << "Copy made from id " << other.id << " to new id " << id
            << "\n";
    }

    // destructor
    ~Tracker() {
        cout << "Object destroyed: id = " << id << "\n";
        count--;
    }

    static int getActiveCount() {
        return count;
    }
};

// static members
int Tracker::count = 0;
int Tracker::nextId = 1;

// returns an object
Tracker makeOne() {
    Tracker t("TempObj");
    return t;
}

// takes object by value
```

```
void useOne(Tracker t) {
    cout << "Using tracker with id = " << t.id << "\n";
}

int main() {
    cout << "Creating objects...\n";
    Tracker a("First");
    Tracker b("Second");
    cout << "Active = " << Tracker::getActiveCount() << "\n\n";

    cout << "Copying object...\n";
    Tracker c = a;
    cout << "Active = " << Tracker::getActiveCount() << "\n\n";

    cout << "Object returned from function...\n";
    Tracker d = makeOne();
    cout << "Active = " << Tracker::getActiveCount() << "\n\n";

    cout << "Passing object to function...\n";
    useOne(b);
    cout << "Active = " << Tracker::getActiveCount() << "\n\n";

    cout << "Block scope demo...\n";
    {
        Tracker e("Block1");
        Tracker f("Block2");
        cout << "Active inside block = " << Tracker::getActiveCount() << "\n";
    } // e and f destroyed here
    cout << "Active after block = " << Tracker::getActiveCount() << "\n";

    return 0;
}
```

Sample Output

```
Creating objects...
Object created: First (id = 1)
Object created: Second (id = 2)
Active = 2

Copying object...
Copy made from id 1 to new id 3
Active = 3

Object returned from function...
Object created: TempObj (id = 4)
Active = 4

Passing object to function...
Copy made from id 2 to new id 5
Using tracker with id = 5
Object destroyed: id = 5
Active = 4

Block scope demo...
```

```
Object created: Block1 (id = 6)
Object created: Block2 (id = 7)
Active inside block = 6
Object destroyed: id = 7
Object destroyed: id = 6
Active after block = 4
Object destroyed: id = 4
Object destroyed: id = 3
Object destroyed: id = 2
Object destroyed: id = 1
```

Q2. Write a C++ program to define a class named SessionManager that models user sessions with the following specifications:

- All data members should be public.
- The data members are:
  - `int sessionId` — a unique ID assigned to each session object.
  - `static int activeSessions` — counts how many sessions currently exist (shared across all instances).
  - `static int nextSessionId` — used to assign unique session IDs (shared across all instances).
- Define the following:
  - **Default constructor:**
    - Automatically assigns a unique `sessionId`.
    - Increments `activeSessions`.
    - Prints: `"Session started. ID = X, Active sessions = Y"`.
  - **Copy constructor:**
    - Creates a new session with a new unique `sessionId`.
    - Increments `activeSessions`.
    - Prints: `"Session duplicated. New ID = X (copied from ID = Y)"`.
  - **Destructor:**
    - Decrements `activeSessions`.
    - Prints: `"Session ended. ID = X, Active sessions before ending = Y"`.
- Define a function `SessionManager startNewSession()` that creates and returns a new session object by value.
- Define a function `void processSession(SessionManager s)` that accepts a session object by value.

In the `main()` function perform the following:

1. Create three `SessionManager` objects.
2. Duplicate one session using the copy constructor.
3. Call `startNewSession()` and assign the returned session to a variable.
4. Pass a session object to `processSession()` (triggering the copy constructor).
5. Create a nested block {} where two more sessions are started.
6. After every step, print the number of active sessions by accessing `activeSessions`.

Code

```cpp
#include <iostream>
using namespace std;

class SessionManager {
public:
    int sessionId;
    static int activeSessions;
    static int nextSessionId;

    // constructor
    SessionManager() {
        sessionId = nextSessionId++;
        activeSessions++;
        cout << "New session started (id = " << sessionId << ")\n";
    }

    // copy constructor
    SessionManager(const SessionManager &other) {
        sessionId = nextSessionId++;
        activeSessions++;
        cout << "Session copied from id " << other.sessionId
             << " to new id " << sessionId << "\n";
    }

    // destructor
    ~SessionManager() {
        cout << "Session ended (id = " << sessionId << ")\n";
        activeSessions--;
    }

    // return new session
    SessionManager makeSession() {
        SessionManager s;
        return s;
    }

    // take session by value
    void handle(SessionManager s) {
        cout << "Handling session id = " << s.sessionId << "\n";
    }
};

// initialize static members
int SessionManager::activeSessions = 0;
int SessionManager::nextSessionId = 1;

int main() {
    cout << "Creating a few sessions...\n";
    SessionManager s1, s2, s3;
    cout << "Currently active: " << SessionManager::activeSessions << "\n\n";

    cout << "Copying a session...\n";
    SessionManager s4 = s2;
    cout << "Currently active: " << SessionManager::activeSessions << "\n\n";

    cout << "Starting a session from inside function...\n";
    SessionManager s5 = s1.makeSession();
    cout << "Currently active: " << SessionManager::activeSessions << "\n\n";

    cout << "Passing session to a function...\n";
    s3.handle(s1);
    cout << "Currently active: " << SessionManager::activeSessions << "\n\n";

    cout << "Block scope example...\n";
    {
        SessionManager s6, s7;
        cout << "Active inside block: " << SessionManager::activeSessions << "\n";
    } // s6, s7 destroyed here
    cout << "Active after block: " << SessionManager::activeSessions << "\n";

    return 0;
}
```

Sample Output

```
Creating a few sessions...
New session started (id = 1)
New session started (id = 2)
New session started (id = 3)
Currently active: 3

Copying a session...
Session copied from id 2 to new id 4
Currently active: 4

Starting a session from inside function...
New session started (id = 5)
Currently active: 5

Passing session to a function...
Session copied from id 1 to new id 6
Handling session id = 6
Session ended (id = 6)
Currently active: 5

```

```
Block scope example...
New session started (id = 7)
New session started (id = 8)
Active inside block: 7
Session ended (id = 8)
Session ended (id = 7)
Active after block: 5
Session ended (id = 5)
Session ended (id = 4)
Session ended (id = 3)
Session ended (id = 2)
Session ended (id = 1)
```

2 Assignment 5 V2

Q3. Define a class Person with attributes name (string) and age (int), and a method display_person() that prints this information. Then create a class Student that inherits from Person and introduces a new attribute student_id (string) along with its own method display_student_id() to print the ID. In your main program, instantiate a Student object, assign values to all attributes, and call both display_person() and display_student_id() to show the student's details.

Code

```cpp
#include <iostream>
using namespace std;

// Base class
class Person {
public:
    string name;
    int age;

    void display_person() {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
    }
};

// Derived class
class Student : public Person {
public:
    string student_id;

    void display_student_id() {
        cout << "Student ID: " << student_id << endl;
    }
};

int main() {
    // Create Student object
    Student s1;

    // Assign values
    s1.name = "Navdeep";
    s1.age = 19;
    s1.student_id = "ST12345";

    // Call functions
    cout << "Student Details:" << endl;
    s1.display_person();
    s1.display_student_id();

    return 0;
}
```

Sample Output

```
Student Details:
Name: Navdeep
Age: 19
Student ID: ST12345
```

Q4. Create a base class called Vehicle that contains an attribute brand (string) and a method show_brand() that prints the brand of the vehicle. Then, define a derived class Car that inherits from Vehicle and adds a new attribute model (string) along with its own method show_model() to print the model. In the main part of the program, create an object of the Car class, set both brand and model values, and call both show_brand() and show_model() methods to display the complete car information.

Code

```
#include <iostream>
using namespace std;

// Base class
class Vehicle {
public:
    string brand;

    void show_brand() {
        cout << "Brand: " << brand << endl;
    }
};

// Derived class
class Car : public Vehicle {
public:
    string model;

    void show_model() {
        cout << "Model: " << model << endl;
    }
};

int main() {
    // Create Car object
    Car c1;

    // Set values
    c1.brand = "Toyota";
    c1.model = "Corolla";

    // Display details
    cout << "Car Information:" << endl;
    c1.show_brand();
    c1.show_model();

    return 0;
}
```

Sample Output

```
Car Information:
Brand: Toyota
Model: Corolla
```

Q5. Create a class Employee with attributes name (string) and salary (float), and a method show_employee() that displays this data. Then, create a subclass Developer that inherits from Employee and adds a new attribute programming_language (string), with its own method show_language() that prints the language the developer uses. In the main section, create a Developer object, assign all attribute values, and call both show_employee() and show_language() to display the complete information.

Code

```
#include <iostream>
using namespace std;

// Base class
class Employee {
public:
    string name;
    float salary;

    void show_employee() {
        cout << "Name: " << name << endl;
        cout << "Salary: " << salary << endl;
    }
};
```

```cpp

// Derived class
class Developer : public Employee {
public:
    string programming_language;

    void show_language() {
        cout << "Programming Language: " << programming_language << endl;
    }
};

int main() {
    // Create Developer object
    Developer d1;

    // Assign values
    d1.name = "Aman";
    d1.salary = 55000.50;
    d1.programming_language = "C++";

    // Show details
    cout << "Developer Information:" << endl;
    d1.show_employee();
    d1.show_language();

    return 0;
}
```

Sample Output

```
Developer Information:
Name: Aman
Salary: 55000.5
Programming Language: C++
```

******* END OF ASSIGNMENT *******