Object Oriented Programming

Chapter 2: Objects & Classes

Coding Problems and Solutions — Set 5

This is a fifth, independent set of programming problems on Objects & Classes, using scenarios different from Sets 1 to 4, while still covering class specification, member functions, dynamic memory within objects, static members, constructors and destructors working with static counters, arrays of objects, objects as function arguments, and access specifiers. Each problem includes the full C++ program, its output, and a short explanation.

Problem 1: Private access is at the class level, not the object level

Statement: Write a program for a class Time (hours and minutes as private members) with a member function isLater() that directly compares the private data of the calling object with the private data of another Time object passed as an argument.

Solution:

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

class Time
{
    private:
        int hours, minutes;
    public:
        void setTime(int h, int m) { hours = h; minutes = m; }
        bool isLater(Time t)
        {
            // directly accessing t.hours and t.minutes, both private
            if (hours != t.hours)
                return hours > t.hours;
            return minutes > t.minutes;
        }
};

int main()
{
    Time t1, t2;
    t1.setTime(10, 45);
    t2.setTime(9, 50);

    if (t1.isLater(t2))
        cout << "t1 is later than t2";
    else
        cout << "t1 is not later than t2";
    return 0;
}
```

Output: t1 is later than t2

Explanation: Although hours and minutes are private, isLater() is a member function of the same class Time, so it is allowed to access t.hours and t.minutes directly, even though t is a different object from the one that called the function. This shows that the private access specifier restricts access based on which class a piece of code belongs to, not which particular object it belongs to.

Problem 2: Dynamic array as a data member, allocated in the constructor and freed in the destructor

Statement: Write a program for a class Marksheet that allocates an array of a given size dynamically inside its constructor (using new), fills and displays it, and releases the memory in its destructor (using delete[]).

Solution:

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

class Marksheet
{
    int *marks;
    int size;
    public:
        Marksheet(int n)            // constructor allocates memory
        {
            size = n;
            marks = new int[size];
            for (int i = 0; i < size; i++)
                marks[i] = (i + 1) * 10;
        }
        void show()
        {
            for (int i = 0; i < size; i++)
                cout << marks[i] << " ";
            cout << endl;
        }
        ~Marksheet()                // destructor releases memory
        {
            delete[] marks;
            cout << "Marksheet memory released." << endl;
        }
};

int main()
{
    Marksheet m(4);
    m.show();
    return 0;
}
```

Output: 10 20 30 40, Marksheet memory released.

Explanation: Unlike a fixed-size array data member, marks here is a pointer, and the actual array it points to is created on the heap with new int[size] inside the constructor, so its size can even be decided at run time. The destructor ~Marksheet() automatically runs when m goes out of scope at the end of main(), freeing the dynamically allocated array with delete[] so no memory is leaked.

Problem 3: An array of pointers to dynamically created objects

Statement: Write a program to create an array of pointers to Book objects, where each Book is created individually on the heap, and then display all of them.

Solution:

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

class Book
{
    public:
        string title;
        Book(string t) { title = t; }
        void show() { cout << title << endl; }
```

```
};

int main()
{
    Book *shelf[3];              // array of pointers, not array of objects

    shelf[0] = new Book("C++ Fundamentals");
    shelf[1] = new Book("Data Structures");
    shelf[2] = new Book("Operating Systems");

    for (int i = 0; i < 3; i++)
        shelf[i]->show();

    for (int i = 0; i < 3; i++)
        delete shelf[i];         // each object released individually
    return 0;
}
```

Output: C++ Fundamentals, Data Structures, Operating Systems

Explanation: shelf is an array of Book pointers rather than an array of Book objects, so each element must be individually pointed at a separately created object using new Book(...). Members are accessed through each pointer using the arrow operator, and since each object was created independently with new, each one must also be released independently with its own delete call.

Problem 4: Static data member automatically tracking the highest value across all objects

Statement: Write a program for a class Score where a static data member automatically keeps track of the highest score recorded among all Score objects created so far.

Solution:

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

class Score
{
    int value;
    static int highest;
    public:
        void setScore(int v)
        {
            value = v;
            if (value > highest)
                highest = value;
        }
        static void showHighest()
        {
            cout << "Highest score so far: " << highest << endl;
        }
};
int Score::highest = 0;

int main()
{
    Score s1, s2, s3;
    s1.setScore(72);
    Score::showHighest();

    s2.setScore(95);
    Score::showHighest();

    s3.setScore(60);
```

```
    Score::showHighest();      // stays 95, since 60 < 95
    return 0;
}
```

Output: Highest score so far: 72, Highest score so far: 95, Highest score so far: 95

Explanation: highest is static, so it is one single value shared by s1, s2 and s3. Every call to setScore() compares the new value against this shared value and updates it only if the new score is larger, meaning the static member effectively remembers information across every object of the class, not just the one currently being modified.

Problem 5: Constructor and destructor together maintaining a live object count

Statement: Write a program for a class Session that increments a static counter in its constructor and decrements the same counter in its destructor, so that the counter always reflects how many Session objects currently exist.

Solution:

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

class Session
{
    static int activeCount;
    public:
        Session() { activeCount++; }     // one more object now exists
        ~Session() { activeCount--; }    // one fewer object now exists
        static void showActive()
        {
            cout << "Active sessions: " << activeCount << endl;
        }
};
int Session::activeCount = 0;

int main()
{
    Session::showActive();          // 0
    {
        Session s1, s2;
        Session::showActive();      // 2
        {
            Session s3;
            Session::showActive();  // 3
        }                            // s3 destroyed here
        Session::showActive();      // back to 2
    }                                // s1, s2 destroyed here
    Session::showActive();          // back to 0
    return 0;
}
```

Output: Active sessions: 0, Active sessions: 2, Active sessions: 3, Active sessions: 2, Active sessions: 0

Explanation: Every time a Session object is created, its constructor increases the single shared counter activeCount, and every time one is destroyed (automatically, when it goes out of scope), its destructor decreases the same counter. Because activeCount is static, this gives an always-accurate, class-wide count of how many objects currently exist at any point in the program, rather than just how many have ever been created.

Problem 6: Array of objects — computing an aggregate (average) over all elements

Statement: Write a program using an array of 4 Employee objects, each storing a salary, and compute the average salary across the whole department.

Solution:

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

class Employee
{
    public:
        float salary;
        void set(float s) { salary = s; }
};

int main()
{
    Employee dept[4];
    dept[0].set(32000);
    dept[1].set(41000);
    dept[2].set(28000);
    dept[3].set(35000);

    float total = 0;
    for (int i = 0; i < 4; i++)
        total += dept[i].salary;

    cout << "Average salary = " << total / 4;
    return 0;
}
```

Output: Average salary = 34000

Explanation: dept[4] is an array of independent Employee objects, and the loop accumulates the salary member of every element into a single running total, exactly as it would for an ordinary array of numbers. This illustrates that aggregate calculations (sum, average, minimum, maximum) over an array of objects follow the same pattern as over arrays of plain data types, just reached through a member of each object.

Problem 7: A function that returns a reference to an object in an array

Statement: Write a program with a function getItem() that returns a reference to a particular Item object inside an array, so that the caller can modify that array element directly through the returned reference.

Solution:

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

class Item
{
    public:
        int quantity;
};

Item& getItem(Item arr[], int index)    // returns a reference, not a copy
{
    return arr[index];
}

int main()
{
    Item stock[3];
    stock[0].quantity = 10;
    stock[1].quantity = 20;
    stock[2].quantity = 30;

    getItem(stock, 1).quantity += 5;    // modifies stock[1] directly
```

```
    for (int i = 0; i < 3; i++)
        cout << "Item " << i << ": " << stock[i].quantity << endl;
    return 0;
}
```

Output: Item 0: 10, Item 1: 25, Item 2: 30

Explanation: Because getItem() returns Item& (a reference) rather than Item (a copy), the expression getItem(stock, 1) refers to the actual stock[1] object itself. Modifying quantity through this returned reference changes stock[1] directly, which is why the second item's quantity becomes 25 instead of remaining unchanged, unlike a function that returned a plain copy of the object.

Problem 8: Using a struct as a simple function return type, contrasted with a class-based version

Statement: Write a program where a function findMinMax() returns both the minimum and maximum of an array of integers together, packaged in a struct, and compare briefly with how a class-based Result object would do the same thing.

Solution:

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

struct Range                // public by default - fine for a simple data bundle
{
    int minVal;
    int maxVal;
};

Range findMinMax(int arr[], int n)
{
    Range r;
    r.minVal = arr[0];
    r.maxVal = arr[0];
    for (int i = 1; i < n; i++)
    {
        if (arr[i] < r.minVal) r.minVal = arr[i];
        if (arr[i] > r.maxVal) r.maxVal = arr[i];
    }
    return r;
}

int main()
{
    int data[5] = {23, 9, 47, 15, 31};
    Range result = findMinMax(data, 5);

    cout << "Min = " << result.minVal << ", Max = " << result.maxVal;
    return 0;
}
```

Output: Min = 9, Max = 47

Explanation: Range is declared as a struct rather than a class because it is used purely to bundle two related values together with no need for data hiding, and its members being public by default makes result.minVal and result.maxVal directly accessible without writing getter functions. Had this been written as a class instead, minVal and maxVal would default to private and would need explicit public getter functions or a public: label to be read in main() — an unnecessary complication for such a simple data-carrying purpose.

Problem 9: A single class using all three access specifiers together

Statement: Write a program for a class EmployeeRecord that uses private for sensitive data, protected for data meant to be shared with a future derived class, and public for the general interface, all in one class.

Solution:

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

class EmployeeRecord
{
    private:
        float basicSalary;          // sensitive - private
    protected:
        string department;          // shared with derived classes - protected
    public:
        string name;                 // general info - public

        void setDetails(string n, string dept, float sal)
        {
            name = n;
            department = dept;
            basicSalary = sal;
        }
        float getSalary()            // controlled access to private data
        {
            return basicSalary;
        }
        void show()
        {
            cout << name << " (" << department << ") - Salary: " << basicSalary <<
endl;
        }
};

int main()
{
    EmployeeRecord e;
    e.setDetails("Priya", "Finance", 48000);
    e.name = "Priya Sharma";        // OK: name is public
    // e.basicSalary = 99999;       // ERROR: basicSalary is private
    // e.department = "HR";         // ERROR: department is protected
    e.show();
    return 0;
}
```

Output: Priya Sharma (Finance) - Salary: 48000

Explanation: All three access specifiers coexist in this one class: name can be read and written directly from main() because it is public; department cannot be touched directly from main() because it is protected (it would only be directly accessible from a class derived from EmployeeRecord); and basicSalary cannot be touched directly at all from outside, being private, so it can only be read through the public getSalary() function or set through setDetails().

Problem 10: An object as a function argument used to transact with the calling object

Statement: Write a program for a class Wallet with a member function transferTo() that takes another Wallet object by reference and transfers a given amount from the calling object's balance into the other object's balance.

Solution:

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

class Wallet
{
    float balance;
    public:
        void setBalance(float b) { balance = b; }
        void transferTo(Wallet &other, float amount)
        {
            if (amount <= balance)
            {
                balance -= amount;
                other.balance += amount;   // modifying the other object directly
            }
            else
            {
                cout << "Insufficient balance for transfer." << endl;
            }
        }
        void show() { cout << "Balance: " << balance << endl; }
};

int main()
{
    Wallet w1, w2;
    w1.setBalance(1000);
    w2.setBalance(200);

    w1.transferTo(w2, 300);

    w1.show();
    w2.show();
    return 0;
}
```

Output: Balance: 700, Balance: 500

Explanation: w1.transferTo(w2, 300) calls transferTo() on w1, so inside the function balance (without a prefix) refers to w1's own balance, while other.balance refers to w2's balance via the reference parameter. Because other is a reference and not a copy, the change made to other.balance is reflected in the actual w2 object back in main(), completing the transfer between two distinct objects of the same class.

Problem 11: Combined program — dynamic array of objects with a static, running class average

Statement: Write a complete program for a class Student used to build a dynamically sized classroom: an array of Student objects created with new, a constructor/destructor pair that maintains a static count of currently active students, and a static function reporting the class average mark.

Solution:

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

class Student
{
    int mark;
    static int activeCount;
    static int markTotal;
    public:
        Student() { mark = 0; activeCount++; }    // markTotal unaffected by a 0
mark
        ~Student()
```

```
        {
            activeCount--;
            markTotal -= mark;
        }
        void setMark(int m)
        {
            markTotal += (m - mark);   // adjust shared total by the change only
            mark = m;
        }
        void show() { cout << "Mark: " << mark << endl; }
        static void showClassAverage()
        {
            if (activeCount == 0)
                cout << "No active students." << endl;
            else
                cout << "Class average: " << (float)markTotal / activeCount << endl;
        }
};
int Student::activeCount = 0;
int Student::markTotal = 0;

int main()
{
    int n = 3;
    Student *classroom = new Student[n];   // dynamic array of objects

    classroom[0].setMark(80);
    classroom[1].setMark(70);
    classroom[2].setMark(90);

    for (int i = 0; i < n; i++)
        classroom[i].show();

    Student::showClassAverage();

    delete[] classroom;                    // destructors run for all 3 elements
    Student::showClassAverage();           // back to 'No active students'
    return 0;
}
```

Output: Mark: 80, Mark: 70, Mark: 90, Class average: 80, No active students.

Explanation: new Student[n] creates 3 default-constructed Student objects on the heap, each incrementing the shared activeCount. Marks are then assigned through setMark(), which carefully adjusts the shared markTotal by only the change in that one object's mark, keeping the class-wide total accurate no matter which object is updated. showClassAverage() then reports the correct running average, and once delete[] classroom; runs, each element's destructor fires, subtracting its own mark from markTotal and bringing activeCount back down to zero.