Object Oriented Programming

Chapter 2: Objects & Classes

Coding Problems and Solutions — Set 4

This is a fourth, independent set of programming problems on Objects & Classes, using scenarios different from Sets 1, 2 and 3, while still covering class specification, member function overloading, arrays within a class, object creation and memory allocation, static members, arrays of objects, objects as function arguments, and access/inheritance-related specifiers. Each problem includes the full C++ program, its output, and a short explanation.

Problem 1: Member function overloading within a class

Statement: Write a program for a class Calculator with two overloaded member functions named add() — one that adds two integers and one that adds two floating-point numbers.

Solution:

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

class Calculator
{
    public:
        int add(int a, int b)
        {
            return a + b;
        }
        float add(float a, float b)     // same name, different parameter types
        {
            return a + b;
        }
};

int main()
{
    Calculator calc;
    cout << "Int sum = " << calc.add(5, 7) << endl;
    cout << "Float sum = " << calc.add(2.5f, 3.2f);
    return 0;
}
```

Output: Int sum = 12, Float sum = 5.7

Explanation: Both functions are named add() and belong to the same class, but they differ in the data types of their parameters, so the compiler treats them as two separate overloaded functions. When calc.add() is called, the compiler chooses the correct version based on the argument types supplied at the call site.

Problem 2: Arrays within a class — a character array data member

Statement: Write a program for a class Quiz that stores 5 answers (as characters 'A'-'D') in a char array data member and compares them against a correct-answer key to compute the score.

Solution:

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

class Quiz
{
```

```
    char answers[5];
    public:
        void submitAnswers()
        {
            char given[5] = {'A', 'C', 'B', 'D', 'A'};
            for (int i = 0; i < 5; i++)
                answers[i] = given[i];
        }
        int evaluate(char key[])
        {
            int score = 0;
            for (int i = 0; i < 5; i++)
                if (answers[i] == key[i])
                    score++;
            return score;
        }
};

int main()
{
    char answerKey[5] = {'A', 'B', 'B', 'D', 'C'};
    Quiz q;
    q.submitAnswers();
    cout << "Score = " << q.evaluate(answerKey) << " out of 5";
    return 0;
}
```

Output: Score = 3 out of 5

Explanation: answers[5] is a char array data member private to the Quiz class, holding the submitted answers. The evaluate() member function compares this internal array element-by-element with an external key array passed in as an ordinary array parameter, and counts the matches.

Problem 3: Creating objects using overloaded constructors (default and parameterized)

Statement: Write a program for a class Product that can create an object either with default values or with values supplied at the time of creation, using constructor overloading.

Solution:

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

class Product
{
    int code;
    float price;
    public:
        Product()                       // default constructor
        {
            code = 0;
            price = 0;
        }
        Product(int c, float p)         // parameterized constructor
        {
            code = c;
            price = p;
        }
        void show()
        {
            cout << "Code: " << code << "  Price: " << price << endl;
        }
};
```

```
int main()
{
    Product p1;                 // uses default constructor
    Product p2(305, 899.50);    // uses parameterized constructor

    p1.show();
    p2.show();
    return 0;
}
```

Output: Code: 0 Price: 0, Code: 305 Price: 899.5

Explanation: Product has two constructors with the same name but different parameter lists, so the compiler decides which one to call based on how the object is declared: Product p1; matches the no-argument constructor, while Product p2(305, 899.50); matches the two-argument constructor. This shows that objects can be created in more than one way depending on the constructors a class provides.

Problem 4: Memory allocation for objects — verifying with the sizeof operator

Statement: Write a program that uses the sizeof operator to show that the memory occupied by an object depends only on its non-static data members, and is unaffected by how many member functions the class has.

Solution:

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

class Small
{
    int a;          // 4 bytes (typical)
    public:
        void show() { cout << a; }
};

class Big
{
    int a;          // 4 bytes
    double b;       // 8 bytes
    char c;         // 1 byte
    public:
        void f1() {}
        void f2() {}
        void f3() {}   // extra functions do NOT add to object size
};

int main()
{
    cout << "sizeof(Small) = " << sizeof(Small) << " bytes" << endl;
    cout << "sizeof(Big)   = " << sizeof(Big)   << " bytes" << endl;
    return 0;
}
```

Output: sizeof(Small) = 4 bytes, sizeof(Big) = 24 bytes (exact value may vary slightly due to compiler padding/alignment)

Explanation: The size of an object reported by sizeof is governed entirely by its non-static data members (with possible extra bytes added by the compiler for alignment/padding) — the number of member functions in Big (f1, f2, f3) has no effect on its size at all, because, as explained in the chapter, function code is stored once and shared by all objects rather than being stored inside each object.

Problem 5: Static data member modelling a shared, limited resource pool

Statement: Write a program for a class ParkingSlot where a fixed number of slots (say 2) are shared across all vehicles trying to park, using a static data member to track availability.

Solution:

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

class ParkingSlot
{
    static int slotsAvailable;
    bool parked;
    public:
        void tryPark()
        {
            if (slotsAvailable > 0)
            {
                parked = true;
                slotsAvailable--;
            }
            else
            {
                parked = false;
            }
        }
        void status()
        {
            cout << (parked ? "Parked. " : "Parking full. ")
                 << "Slots left: " << slotsAvailable << endl;
        }
};
int ParkingSlot::slotsAvailable = 2;

int main()
{
    ParkingSlot car1, car2, car3;
    car1.tryPark(); car1.status();
    car2.tryPark(); car2.status();
    car3.tryPark(); car3.status();   // no slots left
    return 0;
}
```

Output: Parked. Slots left: 1, Parked. Slots left: 0, Parking full. Slots left: 0

Explanation: slotsAvailable is static, so it is shared by car1, car2 and car3 rather than each object tracking its own copy. As each car successfully parks, the single shared value is decremented, and once it reaches zero the third car correctly fails to park, exactly reflecting the class-wide, rather than object-wide, nature of the resource.

Problem 6: Static member function used as a class-wide utility (no object state needed)

Statement: Write a program with a class MathUtils containing static member functions square() and cube(), called directly through the class name without ever creating an object.

Solution:

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

class MathUtils
{
    public:
        static int square(int n) { return n * n; }
        static int cube(int n) { return n * n * n; }
};
```

```
int main()
{
    int num = 4;
    cout << "Square of " << num << " = " << MathUtils::square(num) << endl;
    cout << "Cube of "   << num << " = " << MathUtils::cube(num);
    return 0;
}
```

Output: Square of 4 = 16, Cube of 4 = 64

Explanation: square() and cube() are static member functions, meaning they belong to the class as a whole and do not operate on any particular object's data. Since MathUtils has no data members at all here, it never needs an object to be created — both functions are used purely as organised, class-scoped utility functions.

Problem 7: Arrays of objects representing a fixed set of related items

Statement: Write a program using an array of 4 objects of a class CardSuit to represent the four suits of a deck of playing cards, and display them along with their colour.

Solution:

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

class CardSuit
{
    public:
        string name;
        string color;
        void set(string n, string c) { name = n; color = c; }
        void show() { cout << name << " (" << color << ")" << endl; }
};

int main()
{
    CardSuit suits[4];          // fixed-size array of objects
    suits[0].set("Hearts", "Red");
    suits[1].set("Diamonds", "Red");
    suits[2].set("Clubs", "Black");
    suits[3].set("Spades", "Black");

    for (int i = 0; i < 4; i++)
        suits[i].show();
    return 0;
}
```

Output: Hearts (Red), Diamonds (Red), Clubs (Black), Spades (Black)

Explanation: suits[4] is an array of exactly four CardSuit objects, a natural fit when the number of items is fixed and known in advance. Each element is set and displayed independently through the same pair of member functions, showing how an array of objects can neatly model a small, fixed collection of related real-world entities.

Problem 8: Objects as function arguments — swapping two objects using pointers

Statement: Write a program with a function swapPlayers() that swaps the data of two Player objects using pointers to those objects.

Solution:

```
#include <iostream>
#include <string>
```

```
using namespace std;

class Player
{
    public:
        string name;
        int rank;
};

void swapPlayers(Player *p1, Player *p2)   // objects accessed via pointers
{
    Player temp = *p1;
    *p1 = *p2;
    *p2 = temp;
}

int main()
{
    Player a, b;
    a.name = "Team X"; a.rank = 1;
    b.name = "Team Y"; b.rank = 2;

    swapPlayers(&a, &b);

    cout << a.name << " is now rank " << a.rank << endl;
    cout << b.name << " is now rank " << b.rank;
    return 0;
}
```

Output: Team X is now rank 2, Team Y is now rank 1

Explanation: The addresses of a and b are passed to swapPlayers(), which dereferences the pointers (*p1, *p2) to work with the actual objects. Since whole objects can be assigned to each other just like built-in variables (copying every data member at once), a simple temp-based swap works correctly on entire objects, and the change is visible on the original objects back in main().

Problem 9: An object created and returned by a global (non-member) function

Statement: Write a program with a global function createEmployee() that takes plain values, builds and returns a fully-formed Employee object, without that function being a member of the class.

Solution:

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

class Employee
{
    public:
        string name;
        float salary;
        void show() { cout << name << " - Rs. " << salary << endl; }
};

Employee createEmployee(string n, float s)   // ordinary function, not a member
{
    Employee e;
    e.name = n;
    e.salary = s;
    return e;             // object built and returned
}
```

```
int main()
{
    Employee emp = createEmployee("Farhan", 42000);
    emp.show();
    return 0;
}
```

Output: Farhan - Rs. 42000

Explanation: createEmployee() is an ordinary global function, not a member function of Employee, yet it can freely create a local Employee object because name and salary are public, set its fields, and return the completed object by value — showing that objects behave like any other value type when returned from a function, whether that function is a member of the class or not.

Problem 10: Default inheritance access: struct vs class when used as a base

Statement: Write a program to show that when no inheritance access specifier is written, a struct inherits publicly by default while a class inherits privately by default.

Solution:

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

struct BaseA { int x = 10; };
struct DerivedA : BaseA { };       // no specifier -> public inheritance (struct
default)

class BaseB { public: int y = 20; };
class DerivedB : BaseB { };        // no specifier -> private inheritance (class
default)

int main()
{
    DerivedA da;
    cout << "da.x = " << da.x << endl;   // OK: publicly inherited

    DerivedB db;
    // cout << db.y;   // ERROR: y is privately inherited, not accessible here
    cout << "DerivedB cannot expose y outside without a public: specifier.";
    return 0;
}
```

Output: da.x = 10, DerivedB cannot expose y outside without a public: specifier.

Explanation: This program highlights a lesser-known difference between struct and class: besides the default member-access specifier, their default inheritance mode also differs. DerivedA : BaseA with no specifier inherits publicly (because struct defaults to public), so da.x remains accessible from main(); DerivedB : BaseB with no specifier inherits privately (because class defaults to private), so db.y, though public in BaseB, becomes inaccessible from outside DerivedB — the commented line would fail to compile.

Problem 11: Combined program — overloaded constructors, static tracking, and an array of objects together

Statement: Write a complete program for a class Room in a hostel management system, with overloaded constructors, a static data member tracking how many rooms are occupied, and an array of 3 Room objects.

Solution:

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

class Room
```

```
{
    int roomNo;
    bool occupied;
    static int occupiedCount;
    public:
        Room()                          // default constructor: empty room
        {
            roomNo = 0;
            occupied = false;
        }
        Room(int no)                    // parameterized constructor: occupied room
        {
            roomNo = no;
            occupied = true;
            occupiedCount++;
        }
        void show()
        {
            if (occupied)
                cout << "Room " << roomNo << ": Occupied" << endl;
            else
                cout << "Room (unassigned): Vacant" << endl;
        }
        static void showOccupiedCount()
        {
            cout << "Total occupied rooms: " << occupiedCount << endl;
        }
};
int Room::occupiedCount = 0;

int main()
{
    Room rooms[3] = { Room(101), Room(102), Room() };   // array of objects, mixed
constructors

    for (int i = 0; i < 3; i++)
        rooms[i].show();

    Room::showOccupiedCount();
    return 0;
}
```

Output: Room 101: Occupied, Room 102: Occupied, Room (unassigned): Vacant, Total occupied rooms: 2

Explanation: This program combines constructor overloading (two different constructors for occupied and vacant rooms), a static data member occupiedCount shared across all Room objects and incremented only by the parameterized constructor, an array of objects rooms[3] initialised with a mix of both constructors, and a static member function showOccupiedCount() called through the class name to report the shared total.