Object Oriented Programming

Chapter 2: Objects & Classes

Coding Problems and Solutions — Set 3

This is a third, independent set of programming problems on Objects & Classes, using scenarios different from Set 1 and Set 2, while still covering class specification, member functions, arrays within a class, object creation and memory, static members, 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: Class with member functions defined both inside and outside, using default arguments

Statement: Write a program for a class Loan that calculates simple interest, with the rate having a default value if not supplied by the caller.

Solution:

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

class Loan
{
    private:
        float principal, time, rate;
    public:
        void setLoan(float p, float t, float r = 8.5)   // default argument
        {
            principal = p;
            time = t;
            rate = r;
        }
        float simpleInterest();     // declared here, defined outside
};

float Loan::simpleInterest()
{
    return (principal * time * rate) / 100;
}

int main()
{
    Loan l1, l2;
    l1.setLoan(10000, 2);          // uses default rate 8.5
    l2.setLoan(10000, 2, 10);      // rate explicitly supplied

    cout << "l1 interest = " << l1.simpleInterest() << endl;
    cout << "l2 interest = " << l2.simpleInterest();
    return 0;
}
```

Output: l1 interest = 1700, l2 interest = 2000

Explanation: setLoan() is defined inside the class and uses a default argument for rate, so a call that omits the third argument automatically uses 8.5. simpleInterest() is only declared inside the class and defined outside using Loan::, demonstrating that default arguments and out-of-class definitions can be combined freely in the same class.

Problem 2: Arrays within a class — a 2D array (matrix) as a data member

Statement: Write a program for a class Matrix that stores a 2x2 matrix as an array data member and provides a member function to display it and compute the sum of its diagonal elements.

Solution:

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

class Matrix
{
    int m[2][2];
    public:
        void readMatrix()
        {
            int sample[2][2] = {{4, 7}, {2, 9}};
            for (int i = 0; i < 2; i++)
                for (int j = 0; j < 2; j++)
                    m[i][j] = sample[i][j];
        }
        void display()
        {
            for (int i = 0; i < 2; i++)
            {
                for (int j = 0; j < 2; j++)
                    cout << m[i][j] << " ";
                cout << endl;
            }
        }
        int diagonalSum()
        {
            return m[0][0] + m[1][1];
        }
};

int main()
{
    Matrix mat;
    mat.readMatrix();
    mat.display();
    cout << "Diagonal Sum = " << mat.diagonalSum();
    return 0;
}
```

Output: 4 7 / 2 9 (on separate lines), Diagonal Sum = 13

Explanation: m[2][2] is a two-dimensional array data member, showing that arrays within a class are not limited to one dimension. Every Matrix object gets its own independent 2x2 block of memory for m, and the member functions operate on it exactly as ordinary functions would operate on any 2D array.

Problem 3: Creating objects on the stack vs the heap, and comparing their data

Statement: Write a program that creates one object of a class Coupon on the stack and one on the heap, and show that both behave identically from the point of view of member access.

Solution:

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

class Coupon
{
    public:
        int code;
        float discount;
```

```
        void set(int c, float d) { code = c; discount = d; }
        void show() { cout << "Code: " << code << "  Discount: " << discount << "%"
<< endl; }
};

int main()
{
    Coupon stackObj;             // created on the stack
    stackObj.set(101, 10);
    stackObj.show();

    Coupon *heapObj = new Coupon();   // created on the heap
    heapObj->set(202, 20);
    heapObj->show();

    delete heapObj;
    return 0;
}
```

Output: Code: 101 Discount: 10%, Code: 202 Discount: 20%

Explanation: stackObj is created directly as a named variable, and its memory is automatically managed and released when main() ends. heapObj is created with new, so its memory lives on the heap until explicitly released with delete; the dot operator is used for the stack object and the arrow operator for the heap object accessed through a pointer, but both objects otherwise work the same way.

Problem 4: Static data member shared across objects to enforce a global limit

Statement: Write a program for a class SeatBooking that allows only 3 seats to be booked in total across all objects, using a static data member to track seats remaining.

Solution:

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

class SeatBooking
{
    static int seatsLeft;
    int seatNo;
    bool confirmed;
    public:
        void bookSeat()
        {
            if (seatsLeft > 0)
            {
                seatNo = 4 - seatsLeft;
                seatsLeft--;
                confirmed = true;
            }
            else
            {
                confirmed = false;
            }
        }
        void show()
        {
            if (confirmed)
                cout << "Seat " << seatNo << " booked." << endl;
            else
                cout << "Booking failed - no seats left." << endl;
        }
};
```

```
int SeatBooking::seatsLeft = 3;

int main()
{
    SeatBooking s1, s2, s3, s4;
    s1.bookSeat(); s1.show();
    s2.bookSeat(); s2.show();
    s3.bookSeat(); s3.show();
    s4.bookSeat(); s4.show();      // no seats left by now
    return 0;
}
```

Output: Seat 1 booked., Seat 2 booked., Seat 3 booked., Booking failed - no seats left.

Explanation: seatsLeft is static, so every SeatBooking object checks and updates the same shared counter. Since only one copy of seatsLeft exists, once the first three objects reduce it to zero, the fourth object's call to bookSeat() correctly fails, demonstrating a practical use of static data members to enforce a class-wide limit.

Problem 5: Static member function that validates input before any object is created

Statement: Write a program with a class Age containing a static member function isValid() that checks whether a given age is acceptable (0-120) before an object is actually constructed with it.

Solution:

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

class Age
{
    int years;
    public:
        static bool isValid(int a)
        {
            return (a >= 0 && a <= 120);
        }
        void setAge(int a) { years = a; }
        void show() { cout << "Age: " << years << endl; }
};

int main()
{
    int input = 150;

    if (Age::isValid(input))        // checked before creating/using object
    {
        Age a;
        a.setAge(input);
        a.show();
    }
    else
    {
        cout << "Invalid age: " << input;
    }
    return 0;
}
```

Output: Invalid age: 150

Explanation: isValid() is static, so it can be called using Age:: even before any Age object has been created, which is exactly the point — it is used as a pre-check on raw input, and only if the check passes does the program go on to create and use an actual object.

Problem 6: Arrays of objects — linear search by a data member

Statement: Write a program to declare an array of 5 objects of a class Contact storing name and phone number, and search the array for a contact by name.

Solution:

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

class Contact
{
    public:
        string name;
        string phone;
        void set(string n, string p) { name = n; phone = p; }
};

int main()
{
    Contact list[5];
    list[0].set("Anita", "9001");
    list[1].set("Rohit", "9002");
    list[2].set("Kavya", "9003");
    list[3].set("Suresh", "9004");
    list[4].set("Meena", "9005");

    string search = "Kavya";
    bool found = false;

    for (int i = 0; i < 5; i++)
    {
        if (list[i].name == search)
        {
            cout << search << "'s number: " << list[i].phone;
            found = true;
            break;
        }
    }
    if (!found)
        cout << "Contact not found.";
    return 0;
}
```

Output: Kavya's number: 9003

Explanation: list[5] is an array of Contact objects, and the loop performs a simple linear search by comparing the name member of each object with the target string, illustrating how ordinary array-searching techniques apply directly to arrays of objects.

Problem 7: Objects as function arguments — a function that modifies one of several objects passed to it

Statement: Write a program with a function that receives three Account objects by reference and adds a fixed bonus only to the one with the lowest balance.

Solution:

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

class Account
{
```

```
    public:
        float balance;
        void show() { cout << "Balance: " << balance << endl; }
};

void giveBonusToLowest(Account &a, Account &b, Account &c, float bonus)
{
    Account *lowest = &a;
    if (b.balance < lowest->balance) lowest = &b;
    if (c.balance < lowest->balance) lowest = &c;
    lowest->balance += bonus;
}

int main()
{
    Account a1, a2, a3;
    a1.balance = 5000;
    a2.balance = 2000;
    a3.balance = 3500;

    giveBonusToLowest(a1, a2, a3, 500);

    a1.show();
    a2.show();
    a3.show();
    return 0;
}
```

Output: Balance: 5000, Balance: 2500, Balance: 3500

Explanation: All three objects are passed by reference, so the function can identify and modify the original object with the lowest balance (a2) directly, using an internal pointer lowest that is updated to point to whichever reference-parameter currently holds the smallest value. Only a2's balance changes in main(), confirming the modification reached the correct original object.

Problem 8: Object as a data member with its own setup performed through the outer class

Statement: Write a program where a class Address is used as a data member inside a class Person, and the Person class provides a single function to set both the person's and the address's details together.

Solution:

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

class Address
{
    public:
        string city;
        string pinCode;
        void set(string c, string p) { city = c; pinCode = p; }
};

class Person
{
    string name;
    Address addr;              // Address object as a data member
    public:
        void setPerson(string n, string city, string pin)
        {
            name = n;
            addr.set(city, pin);   // delegating to the inner object's function
```

```
        }
        void show()
        {
            cout << name << ", " << addr.city << " - " << addr.pinCode << endl;
        }
};

int main()
{
    Person p;
    p.setPerson("Ritika", "Siliguri", "734001");
    p.show();
    return 0;
}
```

Output: Ritika, Siliguri - 734001

Explanation: addr is a data member of Person whose type is itself the class Address, so every Person object contains a full Address object inside it. Person::setPerson() does not set city and pinCode directly; instead it calls addr.set(), delegating that part of the work to the embedded object's own member function, which is a natural way to organise related data using object composition.

Problem 9: private vs protected — showing what a derived class can and cannot access

Statement: Write a program with a base class Vehicle having one private and one protected data member, and a derived class Bike that shows which one it can access directly.

Solution:

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

class Vehicle
{
    private:
        int engineNo;          // not accessible to derived class directly
    protected:
        int topSpeed;          // accessible to derived class directly
    public:
        void setEngine(int e) { engineNo = e; }
        int getEngine() { return engineNo; }
};

class Bike : public Vehicle
{
    public:
        void setSpeed(int s) { topSpeed = s; }      // OK - protected
        // void setEngineDirect(int e) { engineNo = e; }   // ERROR - private
        void show()
        {
            cout << "Top Speed: " << topSpeed << endl;
            cout << "Engine No (via base function): " << getEngine() << endl;
        }
};

int main()
{
    Bike b;
    b.setSpeed(120);
    b.setEngine(4521);       // public function used to set private member
    b.show();
    return 0;
}
```

Output: Top Speed: 120, Engine No (via base function): 4521

Explanation: topSpeed is protected, so Bike's own member function setSpeed() can assign to it directly. engineNo is private to Vehicle, so Bike cannot touch it directly at all (the commented line would fail to compile); it can only be set or read through Vehicle's own public functions setEngine() and getEngine(), which remain accessible to everyone, including derived classes and outside code.

Problem 10: Struct with a member function vs class — proving both can have functions in C++

Statement: Write a program using a struct (not a class) that contains a member function, to show that in C++, unlike in C, a struct is allowed to have functions — with public-by-default access being the only real difference from a class.

Solution:

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

struct Timer
{
    int minutes, seconds;          // public by default

    void setTime(int m, int s)     // struct CAN have member functions in C++
    {
        minutes = m;
        seconds = s;
    }
    int totalSeconds()
    {
        return minutes * 60 + seconds;
    }
};

int main()
{
    Timer t;
    t.setTime(2, 30);
    t.minutes = 3;              // OK: direct access, since struct members are
public
    cout << "Total seconds = " << t.totalSeconds();
    return 0;
}
```

Output: Total seconds = 210

Explanation: Timer is declared as a struct but still has member functions, exactly like a class would, because in C++ a struct is essentially a class with public default access rather than private. t.minutes = 3; works directly because struct members are public unless stated otherwise, which would have caused a compile error had Timer been written as a class without an explicit public: label.

Problem 11: Array of objects combined with a static counter that resets

Statement: Write a program for a class Token where an array of 4 Token objects is created, each getting a serial number from a static counter, and provide a static function to reset the counter.

Solution:

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

class Token
{
    static int counter;
```

```
    int serial;
    public:
        Token() { serial = ++counter; }
        void show() { cout << "Token #" << serial << endl; }
        static void resetCounter() { counter = 0; }
};
int Token::counter = 0;

int main()
{
    Token batch1[4];             // 4 objects created, counter runs 1..4
    for (int i = 0; i < 4; i++)
        batch1[i].show();

    Token::resetCounter();       // reset before next batch

    Token batch2[2];             // counter restarts from 1
    for (int i = 0; i < 2; i++)
        batch2[i].show();
    return 0;
}
```

Output: Token #1, Token #2, Token #3, Token #4, Token #1, Token #2

Explanation: counter is static and shared across all Token objects, so declaring the array Token batch1[4]; calls the default constructor four times in a row, each time incrementing the same shared counter. resetCounter(), also static, is then called using the class name to bring the shared counter back to zero, so the next array, batch2, starts numbering from 1 again.