Object Oriented Programming

Chapter 2: Objects & Classes

Coding Problems and Solutions — Set 2

This is a second, independent set of programming problems on Objects & Classes, using different scenarios from Set 1 while still covering class specification, member functions, arrays within a class, object creation and memory, static members, arrays of objects, objects as function arguments, objects as data members of another class, and access specifiers. Each problem includes the full C++ program, its output, and a short explanation.

Problem 1: Specifying a class with a member function that returns a value

Statement: Write a program to define a class Complex representing a complex number (real and imaginary parts) with a member function add() that adds two Complex objects and returns the result as a new Complex object.

Solution:

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

class Complex
{
    private:
        float real, imag;
    public:
        void setValue(float r, float i) { real = r; imag = i; }
        Complex add(Complex c2)
        {
            Complex temp;
            temp.real = real + c2.real;
            temp.imag = imag + c2.imag;
            return temp;            // an object is returned
        }
        void show()
        {
            cout << real << " + " << imag << "i" << endl;
        }
};

int main()
{
    Complex c1, c2, c3;
    c1.setValue(3, 4);
    c2.setValue(1, 2);

    c3 = c1.add(c2);       // function returns an object
    c3.show();
    return 0;
}
```

Output: 4 + 6i

Explanation: The member function add() creates a local object temp, fills it with the sum of the two complex numbers, and returns it by value. Since add() is a member function of Complex, it can directly access the private members of the object that calls it (c1) as well as of the argument object (c2), because private access is granted at the class level, not the object level.

Problem 2: Arrays within a class — searching within the array

Statement: Write a program for a class Inventory that stores stock quantities of 5 items in an array data member, and provides a member function to search for and display the quantity of a given item by index.

Solution:

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

class Inventory
{
    int stock[5];
    public:
        void loadStock()
        {
            int sample[5] = {120, 45, 78, 200, 15};
            for (int i = 0; i < 5; i++)
                stock[i] = sample[i];
        }
        void lowStockAlert(int threshold)
        {
            for (int i = 0; i < 5; i++)
                if (stock[i] < threshold)
                    cout << "Item " << i << " is LOW (qty " << stock[i] << ")" <<
endl;
        }
};

int main()
{
    Inventory inv;
    inv.loadStock();
    inv.lowStockAlert(50);
    return 0;
}
```

Output: Item 1 is LOW (qty 45), Item 4 is LOW (qty 15)

Explanation: stock[5] is an array data member private to the class. The member function lowStockAlert() loops through this internal array and compares each value against a threshold passed as an argument, printing only the items whose quantity falls below it.

Problem 3: Dynamic object creation and memory allocation using new and delete

Statement: Write a program to create an object of a class Car dynamically using the new operator, use it through a pointer, and release its memory using delete.

Solution:

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

class Car
{
    string model;
    int year;
    public:
        void setData(string m, int y) { model = m; year = y; }
        void showData() { cout << model << " (" << year << ")" << endl; }
};

int main()
```

```
{
    Car *ptr = new Car();     // object created on the heap
    ptr->setData("Sedan X", 2023);
    ptr->showData();

    delete ptr;               // memory released
    ptr = nullptr;
    return 0;
}
```

Output: Sedan X (2023)

Explanation: new Car() allocates memory for a Car object on the heap and returns its address, which is stored in the pointer ptr. Members are accessed through the pointer using the arrow operator (->). Since the object was created with new, it must be explicitly released with delete once it is no longer needed, otherwise the memory remains occupied for the life of the program (a memory leak).

Problem 4: Static data member used to auto-generate unique IDs

Statement: Write a program for a class Ticket that automatically assigns a unique, increasing ticket number to every object created, using a static data member.

Solution:

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

class Ticket
{
    static int nextNumber;   // shared across all objects
    int ticketNo;
    public:
        Ticket()
        {
            ticketNo = nextNumber;
            nextNumber++;
        }
        void show() { cout << "Ticket No: " << ticketNo << endl; }
};
int Ticket::nextNumber = 1001;   // starting ticket number

int main()
{
    Ticket t1, t2, t3;
    t1.show();
    t2.show();
    t3.show();
    return 0;
}
```

Output: Ticket No: 1001, Ticket No: 1002, Ticket No: 1003

Explanation: nextNumber is static, so only one copy of it exists no matter how many Ticket objects are created. Each time the constructor runs, the current value of nextNumber is copied into that object's own ticketNo (a non-static member, unique per object), and then the shared counter is incremented for the next object.

Problem 5: Static member function as a utility (no object required)

Statement: Write a program with a class TempConverter containing a static member function that converts Celsius to Fahrenheit, called without creating any object.

Solution:

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

class TempConverter
{
    public:
        static float celsiusToFahrenheit(float c)
        {
            return (c * 9 / 5) + 32;
        }
};

int main()
{
    cout << "25C = " << TempConverter::celsiusToFahrenheit(25) << "F" << endl;
    cout << "100C = " << TempConverter::celsiusToFahrenheit(100) << "F";
    return 0;
}
```

Output: 25C = 77F, 100C = 212F

Explanation: celsiusToFahrenheit() is declared static, so it belongs to the class as a whole rather than to any specific object, and it is called directly using the class name and the scope resolution operator, TempConverter::, without ever creating a TempConverter object — appropriate here since the conversion does not depend on any object's state.

Problem 6: Arrays of objects — sorting objects by a data member

Statement: Write a program to declare an array of 5 objects of a class Player storing name and score, and sort the array in descending order of score using a simple bubble sort.

Solution:

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

class Player
{
    public:
        string name;
        int score;
        void set(string n, int s) { name = n; score = s; }
};

int main()
{
    Player p[5];
    p[0].set("Amit", 45);
    p[1].set("Sonal", 78);
    p[2].set("Rakesh", 62);
    p[3].set("Neha", 90);
    p[4].set("Vikas", 55);

    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 4 - i; j++)
            if (p[j].score < p[j + 1].score)
            {
                Player t = p[j];
                p[j] = p[j + 1];
                p[j + 1] = t;
            }
```

```
    for (int i = 0; i < 5; i++)
        cout << p[i].name << " - " << p[i].score << endl;
    return 0;
}
```

Output: Neha - 90, Sonal - 78, Rakesh - 62, Vikas - 55, Amit - 45

Explanation: p[5] is an array of Player objects. Since Player has only public members here, whole objects can be swapped directly with a simple assignment (Player t = p[j];), which copies every data member of one object into another — the same bubble-sort logic used for ordinary arrays works equally well on an array of objects.

Problem 7: Objects as function arguments — a function that compares two objects

Statement: Write a program with a function that takes two Box objects (passed by reference, for efficiency) and returns a reference-free copy of whichever one has the larger volume.

Solution:

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

class Box
{
    public:
        float length, width, height;
        void set(float l, float w, float h) { length = l; width = w; height = h; }
        float volume() { return length * width * height; }
};

Box larger(Box &b1, Box &b2)   // objects passed by reference
{
    if (b1.volume() >= b2.volume())
        return b1;              // object returned by value
    else
        return b2;
}

int main()
{
    Box a, b, result;
    a.set(2, 3, 4);     // volume = 24
    b.set(5, 2, 3);     // volume = 30

    result = larger(a, b);
    cout << "Larger box volume = " << result.volume();
    return 0;
}
```

Output: Larger box volume = 30

Explanation: a and b are passed to larger() by reference, avoiding the cost of copying them just to compare their volumes. The function itself returns a Box object by value, so the object assigned to result in main() is a fresh copy of whichever box had the greater volume.

Problem 8: Object as a data member of another class (object composition)

Statement: Write a program where a class Engine is used as a data member inside a class Car, and show how the Car object's member function uses the embedded Engine object.

Solution:

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

```
class Engine
{
    public:
        int horsepower;
        void setPower(int hp) { horsepower = hp; }
};

class Car
{
    string model;
    Engine engine;         // object of Engine used as a data member
    public:
        void setCar(string m, int hp)
        {
            model = m;
            engine.setPower(hp);   // calling Engine's function on the embedded
object
        }
        void show()
        {
            cout << model << " - " << engine.horsepower << " HP" << endl;
        }
};

int main()
{
    Car c;
    c.setCar("Falcon", 180);
    c.show();
    return 0;
}
```

Output: Falcon - 180 HP

Explanation: Here engine is a private data member of Car, but its type is itself a class (Engine) rather than a built-in type. Each Car object therefore contains a complete Engine object inside it. Car's member functions can call Engine's public member functions on this embedded object using the ordinary dot operator, just as they would on any other data member.

Problem 9: Protected access specifier — accessing base class data safely from a derived class

Statement: Write a program with a base class Shape having a protected data member color, and a derived class Circle that sets and displays it directly.

Solution:

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

class Shape
{
    protected:
        string color;
};

class Circle : public Shape
{
    float radius;
    public:
        void setCircle(string c, float r)
        {
            color = c;   // direct access to protected member
```

```
            radius = r;
        }
        void show()
        {
            cout << "Color: " << color << ", Area: " << 3.14 * radius * radius <<
endl;
        }
};

int main()
{
    Circle c1;
    c1.setCircle("Red", 4);
    c1.show();
    // c1.color = "Blue";   // ERROR: color is protected, not accessible outside
    return 0;
}
```

Output: Color: Red, Area: 50.24

Explanation: color is declared protected in Shape, so it cannot be accessed directly through an object in main(), as the commented line shows. However, it can be accessed directly inside setCircle(), a member function of the derived class Circle, without needing any getter/setter from the base class — this is exactly what the protected specifier is designed for.

Problem 10: Struct vs class — a program that will not compile if access rules are ignored

Statement: Write a short program showing a compile-time error caused by forgetting that a class defaults to private access (unlike a struct), and then show the corrected version.

Solution:

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

// Incorrect version (will NOT compile):
// class Wallet
// {
//     float balance;         // private by default
// };
// int main()
// {
//     Wallet w;
//     w.balance = 500;       // ERROR: balance is private
// }

// Corrected version:
class Wallet
{
    public:                   // explicitly made public
        float balance;
};

int main()
{
    Wallet w;
    w.balance = 500;          // OK now
    cout << "Balance = " << w.balance;
    return 0;
}
```

Output: Balance = 500

Explanation: If Wallet were written as a struct, balance would have been public automatically and w.balance = 500; would compile without any changes. Because it is a class, balance is private by default, so direct outside access fails until the public: specifier is added explicitly — this program isolates exactly the default-access difference between struct and class.

Problem 11: Array of dynamically created objects (new with an array)

Statement: Write a program to dynamically create an array of 3 Student objects using new, assign roll numbers, display them, and free the memory.

Solution:

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

class Student
{
    public:
        int roll;
        void setRoll(int r) { roll = r; }
        void show() { cout << "Roll: " << roll << endl; }
};

int main()
{
    Student *arr = new Student[3];   // array of objects on the heap

    for (int i = 0; i < 3; i++)
        arr[i].setRoll(100 + i);

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

    delete[] arr;                    // array delete for array new
    return 0;
}
```

Output: Roll: 100, Roll: 101, Roll: 102

Explanation: new Student[3] allocates memory for 3 Student objects on the heap and returns a pointer to the first one, so individual elements can be accessed with the usual array indexing (arr[i]). Because the memory was allocated as an array with new[], it must be released using delete[] rather than a plain delete, so that all three objects are correctly destroyed.

Problem 12: Combined program — class specification, static counter, array of objects, and object argument together

Statement: Write a complete program for a class LibraryBook with title and price, a static member that tracks the total value of all books created, an array of 3 book objects, and a function that takes a book object by reference to apply a discount.

Solution:

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

class LibraryBook
{
    string title;
    float price;
    static float totalValue;    // static data member
```

```
    public:
        void setBook(string t, float p)
        {
            title = t;
            price = p;
            totalValue += p;     // shared total updated on every book
        }
        float getPrice() { return price; }
        void setPrice(float p) { price = p; }
        void show() { cout << title << " - Rs. " << price << endl; }
        static void showTotal()
        {
            cout << "Total value of all books: Rs. " << totalValue << endl;
        }
};
float LibraryBook::totalValue = 0;

void applyDiscount(LibraryBook &b, float percent)   // object passed by reference
{
    b.setPrice(b.getPrice() - (b.getPrice() * percent / 100));
}

int main()
{
    LibraryBook books[3];        // array of objects
    books[0].setBook("C++ Primer", 600);
    books[1].setBook("Let Us C", 350);
    books[2].setBook("OOP Made Easy", 450);

    applyDiscount(books[0], 10); // discount applied on original object

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

    LibraryBook::showTotal();
    return 0;
}
```

Output: C++ Primer - Rs. 540, Let Us C - Rs. 350, OOP Made Easy - Rs. 450, Total value of all books: Rs. 1400

Explanation: This program combines several ideas: totalValue is a static data member shared by all LibraryBook objects and updated inside setBook(); books[3] is an array of objects; applyDiscount() takes an object by reference so that the change made through setPrice() affects the original object in the array (note that totalValue itself is not reduced, since it was updated only once, at creation time, reflecting the original prices); and showTotal() is a static member function called using the class name.