

Object Oriented Programming

Chapter 3: Constructors and Destructors

Coding Problems and Solutions

This set contains hands-on C++ programming problems built around constructors and destructors — covering default, parameterized, copy, overloaded and explicit constructors, along with destructor behaviour and resource management. Each problem includes the problem statement, a complete working solution, and its expected output.

Problem 1: Default Constructor — Object Counter

Problem Statement:

Write a C++ class Counter that uses a static data member to keep track of how many objects of the class currently exist. The default constructor should increment this count every time a new object is created. Display the total count after creating three objects.

Solution:

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

class Counter {
    static int count;
public:
    Counter() {          // default constructor
        count++;
    }
    static int getCount() { return count; }
};
int Counter::count = 0;

int main() {
    Counter c1, c2, c3;
    cout << "Total objects created: " << Counter::getCount();
    return 0;
}
```

Output:

```
Total objects created: 3
```

Problem 2: Parameterized Constructor — Rectangle Area

Problem Statement:

Write a C++ program that defines a class Rectangle with a parameterized constructor accepting length and breadth. Create two objects with different dimensions and print the area of each.

Solution:

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

```

class Rectangle {
    double length, breadth;
public:
    Rectangle(double l, double b) {    // parameterized constructor
        length = l;
        breadth = b;
    }
    double area() { return length * breadth; }
};

int main() {
    Rectangle r1(12.0, 5.0);
    Rectangle r2(7.5, 4.0);
    cout << "Area of r1: " << r1.area() << endl;
    cout << "Area of r2: " << r2.area() << endl;
    return 0;
}

```

Output:

```

Area of r1: 60
Area of r2: 30

```

Problem 3: Copy Constructor — Shallow Copy Problem

Problem Statement:

Write a C++ program with a class `DynamicArray` that allocates memory dynamically in its constructor. Demonstrate, using the compiler's default (shallow) copy constructor, what problem occurs when one object is destroyed while another still references the same memory.

Solution:

```

#include <iostream>
using namespace std;

class DynamicArray {
    int *data;
public:
    DynamicArray(int val) {
        data = new int(val);
    }
    // no user-defined copy constructor -> compiler uses shallow copy
    void show() { cout << *data << endl; }
    ~DynamicArray() { delete data; }
};

int main() {
    DynamicArray a(10);
    {
        DynamicArray b = a;    // shallow copy: b.data == a.data
    }                          // b's destructor deletes the shared memory
    // a.show();               // UNDEFINED BEHAVIOUR: a.data now dangles
    cout << "b went out of scope; a's pointer is now dangling." << endl;
    return 0;
}

```

Explanation:

Because no copy constructor is written, the compiler performs a member-wise copy, so `a.data` and `b.data` point to the same memory block. When `b` is destroyed at the end of the inner block, its destructor frees that memory. The object `a` is left holding a dangling pointer, and using `a.show()` afterwards leads to undefined behaviour. This demonstrates exactly why a deep copy constructor is needed (see Problem 4).

Problem 4: Copy Constructor — Deep Copy Solution**Problem Statement:**

Rewrite the `DynamicArray` class from Problem 3 with a user-defined deep copy constructor so that each object owns independent memory, and show that modifying one object's data does not affect the other.

Solution:

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

class DynamicArray {
    int *data;
public:
    DynamicArray(int val) { data = new int(val); }
    DynamicArray(const DynamicArray &src) { // deep copy constructor
        data = new int(*src.data);
    }
    void set(int val) { *data = val; }
    void show() { cout << *data << endl; }
    ~DynamicArray() { delete data; }
};

int main() {
    DynamicArray a(10);
    DynamicArray b = a; // deep copy: b.data is a separate block
    b.set(99);
    cout << "a: "; a.show();
    cout << "b: "; b.show();
    return 0;
}
```

Output:

```
a: 10
b: 99
```

Problem 5: Constructor Overloading — Complex Number Class**Problem Statement:**

Write a C++ class `Complex` representing a complex number (real and imaginary parts). Overload the constructor so that an object can be created with no arguments ($0 + 0i$), with one argument (real part only, imaginary part 0), or with two arguments (real and imaginary parts). Overload the `+` operator to add two complex numbers and display the result.

Solution:

```
#include <iostream>
```

```

using namespace std;

class Complex {
    double real, imag;
public:
    Complex() { real = 0; imag = 0; }           // zero-argument
    Complex(double r) { real = r; imag = 0; }   // one argument
    Complex(double r, double i) { real = r; imag = i; } // two arguments

    Complex operator+(const Complex &c) {
        return Complex(real + c.real, imag + c.imag);
    }
    void show() {
        cout << real << " + " << imag << "i" << endl;
    }
};

int main() {
    Complex c1;
    Complex c2(5);
    Complex c3(3, 4);
    Complex c4 = c2 + c3;
    c4.show();
    return 0;
}

```

Output:

```
8 + 4i
```

Problem 6: Explicit Constructor — Preventing Implicit Conversion

Problem Statement:

Write a C++ class Temperature with a single-argument constructor that should not allow implicit conversion from a plain double. Demonstrate the compiler error that occurs without explicit, and show the corrected version that compiles safely.

Solution:

```

#include <iostream>
using namespace std;

class Temperature {
    double celsius;
public:
    explicit Temperature(double c) { celsius = c; }
    double get() { return celsius; }
};

void printTemp(Temperature t) {
    cout << t.get() << " C" << endl;
}

int main() {
    Temperature t1(36.6);    // OK: direct initialisation
    printTemp(t1);           // OK: already a Temperature object
}

```

```
// printTemp(37.0);    // ERROR: implicit double->Temperature blocked
return 0;
}
```

Explanation:

Without the explicit keyword, the line `printTemp(37.0)` would compile silently by implicitly converting the double `37.0` into a temporary `Temperature` object. Declaring the constructor explicit forces the programmer to write `printTemp(Temperature(37.0))` instead, making the conversion visible and intentional.

Problem 7: Destructor — Resource Release (File Handling)

Problem Statement:

Write a C++ class `LogFile` that opens a file in its constructor and closes it in its destructor, following the RAII idiom, so the file is guaranteed to be closed even if the object goes out of scope early.

Solution:

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

class LogFile {
    ofstream file;
public:
    LogFile(const string &filename) {
        file.open(filename);
        cout << "File opened" << endl;
    }
    void write(const string &msg) { file << msg << endl; }
    ~LogFile() {                          // destructor releases the resource
        file.close();
        cout << "File closed" << endl;
    }
};

int main() {
    {
        LogFile log("activity.log");
        log.write("Program started");
    } // destructor runs automatically here
    cout << "End of main" << endl;
    return 0;
}
```

Output:

```
File opened
File closed
End of main
```

Problem 8: Destructor Order — Array of Objects

Problem Statement:

Write a C++ program that creates an array of three objects of a class Item, each printing a message in its constructor and destructor. Observe and explain the order in which the destructors are called when the array goes out of scope.

Solution:

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

class Item {
    int id;
public:
    Item(int i) : id(i) {
        cout << "Constructing Item " << id << endl;
    }
    ~Item() {
        cout << "Destroying Item " << id << endl;
    }
};

int main() {
    Item arr[3] = { Item(1), Item(2), Item(3) };
    cout << "End of main reached" << endl;
    return 0;
}
```

Output:

```
Constructing Item 1
Constructing Item 2
Constructing Item 3
End of main reached
Destroying Item 3
Destroying Item 2
Destroying Item 1
```

Explanation:

Objects in an array are constructed in order from the first element to the last. When the array goes out of scope, however, destructors are called in the reverse order — from the last element to the first — which mirrors the general rule that destruction always undoes construction in reverse.

Problem 9: Virtual Destructor — Safe Polymorphic Deletion

Problem Statement:

Write a C++ program with a base class Shape and a derived class Circle, where Circle allocates memory dynamically. Show what goes wrong when the base class destructor is not virtual and a Circle object is deleted through a Shape pointer, then fix it using a virtual destructor.

Solution:

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

class Shape {
public:
```

```

    virtual ~Shape() {                // virtual destructor
        cout << "Shape destructor" << endl;
    }
};

class Circle : public Shape {
    int *radiusData;
public:
    Circle() { radiusData = new int(5); }
    ~Circle() {
        delete radiusData;
        cout << "Circle destructor" << endl;
    }
};

int main() {
    Shape *s = new Circle();
    delete s;    // with virtual ~Shape(), both destructors run correctly
    return 0;
}

```

Output:

```

Circle destructor
Shape destructor

```

Explanation:

If `~Shape()` were not declared virtual, `delete s` would call only the base class destructor, so `Circle`'s destructor — and the `delete radiusData` inside it — would never run, causing a memory leak. Marking the base class destructor virtual ensures the most-derived destructor is always invoked first, followed by the base class destructor, releasing every resource correctly.

Problem 10: Combined Program — Bank Account with Constructor Overloading, Deep Copy and Destructor

Problem Statement:

Write a complete C++ program for a class `BankAccount` that: (a) supports a default constructor, a parameterized constructor and a deep copy constructor; (b) dynamically stores the account holder's name; and (c) uses a destructor to release that memory, printing a message whenever an account object is destroyed.

Solution:

```

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

class BankAccount {
    char *name;
    double balance;
public:
    BankAccount() {                // default constructor
        name = new char[8];
        strcpy(name, "Unknown");
    }
};

```

```

        balance = 0.0;
    }
    BankAccount(const char *n, double b) {           // parameterized constructor
        name = new char[strlen(n) + 1];
        strcpy(name, n);
        balance = b;
    }
    BankAccount(const BankAccount &src) {           // deep copy constructor
        name = new char[strlen(src.name) + 1];
        strcpy(name, src.name);
        balance = src.balance;
    }
    void show() {
        cout << name << ": Rs. " << balance << endl;
    }
    ~BankAccount() {                               // destructor
        cout << "Closing account of " << name << endl;
        delete[] name;
    }
};

int main() {
    BankAccount a1;
    BankAccount a2("Arun Roy", 15000.0);
    BankAccount a3 = a2;    // deep copy constructor invoked
    a1.show();
    a2.show();
    a3.show();
    return 0;
}

```

Output:

```

Unknown: Rs. 0
Arun Roy: Rs. 15000
Arun Roy: Rs. 15000
Closing account of Arun Roy
Closing account of Arun Roy
Closing account of Unknown

```

Explanation:

Each object allocates its own block of memory for the name, so a3 (created via the deep copy constructor) does not share memory with a2. When main() ends, the three destructors run automatically in the reverse order of construction (a3, a2, then a1), each safely freeing only the memory it owns.

Additional Practice Problems (Try Yourself)

- Write a class Student with a parameterized constructor to initialise roll number and name, and a destructor that prints a farewell message when the object is destroyed.
- Write a class Matrix that allocates a 2D array dynamically in its constructor, provides a deep copy constructor, and releases the memory in its destructor.
- Write a class Employee with three overloaded constructors (zero, one and three arguments) and demonstrate all three in main().
- Write a class Stack using a dynamically allocated array, with a constructor to set its capacity and a destructor to free the array.
- Write a program with a base class Animal and a derived class Dog, each printing messages in their constructors and destructors, and verify the order of both construction and destruction.
- Write a class String that mimics a simple string type: store characters dynamically, provide a deep copy constructor, an explicit constructor from a C-string, and a destructor.