

Object Oriented Programming

Chapter 3: Constructors and Destructors

Coding Problems and Solutions — Part 2 (Advanced)

This is a second, more advanced set of C++ programming problems on constructors and destructors. It builds on Part 1 and covers member initializer lists, delegating constructors, constructors for classes with member (nested) objects, the copy constructor vs. the assignment operator, singleton-style private constructors, default arguments, dynamic arrays of objects, and exception safety during construction. Each problem includes the problem statement, a complete working solution, and its expected output.

Problem 1: Member Initializer List — Constructing const and Reference Members

Problem Statement:

Write a C++ class Circle that stores its radius as a const double and also holds a reference to an external int counter. Explain why these members cannot be set inside the constructor body, and write the constructor using a member initializer list instead.

Solution:

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

class Circle {
    const double radius;
    int &objectCounter;
public:
    Circle(double r, int &counter)
        : radius(r), objectCounter(counter) {    // initializer list
        objectCounter++;
    }
    double area() { return 3.1416 * radius * radius; }
};

int main() {
    int total = 0;
    Circle c1(4.0, total);
    Circle c2(2.5, total);
    cout << "Area of c1: " << c1.area() << endl;
    cout << "Circles created: " << total << endl;
    return 0;
}
```

Output:

```
Area of c1: 50.2656
Circles created: 2
```

Explanation:

A const data member and a reference member must both be given a value at the moment they are created, because neither can be reassigned afterwards. If radius or objectCounter were set inside the constructor's body with an ordinary assignment, the compiler would reject the code, since by that point the members

already exist uninitialised. The member initializer list initialises them directly as part of construction, which is the only place this is allowed.

Problem 2: Delegating Constructors — Avoiding Duplicate Initialization Code

Problem Statement:

Write a C++ class Box with three constructors (zero, one and three arguments) such that the simpler constructors delegate to the most general one instead of repeating the initialisation logic, using C++11 delegating constructor syntax.

Solution:

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

class Box {
    int length, breadth, height;
public:
    Box(int l, int b, int h) : length(l), breadth(b), height(h) {
        cout << "Main constructor called" << endl;
    }
    Box(int side) : Box(side, side, side) {}           // delegates
    Box() : Box(1, 1, 1) {}                           // delegates

    int volume() { return length * breadth * height; }
};

int main() {
    Box b1;
    Box b2(4);
    Box b3(2, 3, 5);
    cout << b1.volume() << " " << b2.volume() << " " << b3.volume() << endl;
    return 0;
}
```

Explanation:

Each delegating constructor's body runs after the constructor it delegates to has fully finished, so Box(int) and Box() both end up calling the three-argument constructor's initialisation logic exactly once, instead of duplicating the assignment code in every constructor.

Problem 3: Constructors for Classes Containing Member (Nested) Objects

Problem Statement:

Write a C++ program with a class Engine and a class Car, where Car contains an Engine object as a data member. Show the order in which the constructors and destructors of Engine and Car execute when a Car object is created and destroyed.

Solution:

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

class Engine {
public:
```

```

    Engine() { cout << "Engine constructed" << endl; }
    ~Engine() { cout << "Engine destroyed" << endl; }
};

class Car {
    Engine engine;      // member object
public:
    Car() { cout << "Car constructed" << endl; }
    ~Car() { cout << "Car destroyed" << endl; }
};

int main() {
    Car c;
    return 0;
}

```

Output:

```

Engine constructed
Car constructed
Car destroyed
Engine destroyed

```

Explanation:

Before a class's own constructor body runs, the constructors of its member objects execute first, in the order the members are declared. Destruction always happens in the exact reverse order: the enclosing object's own destructor body runs first, and only afterwards are its member objects destroyed.

Problem 4: Copy Constructor vs. Overloaded Assignment Operator

Problem Statement:

Write a C++ class Wallet with a dynamically allocated balance. Implement both a deep copy constructor and an overloaded assignment operator (operator=), and demonstrate the difference between when each one is called. Guard the assignment operator against self-assignment.

Solution:

```

#include <iostream>
using namespace std;

class Wallet {
    int *balance;
public:
    Wallet(int b) { balance = new int(b); }

    Wallet(const Wallet &src) {                // copy constructor
        balance = new int(*src.balance);
        cout << "Copy constructor called" << endl;
    }

    Wallet& operator=(const Wallet &src) {      // assignment operator
        if (this == &src) return *this;        // guard against self-assignment
        delete balance;                         // release old memory first
        balance = new int(*src.balance);
        cout << "Assignment operator called" << endl;
    }
}

```

```

        return *this;
    }

    void show() { cout << *balance << endl; }
    ~Wallet() { delete balance; }
};

int main() {
    Wallet w1(500);
    Wallet w2 = w1;      // copy constructor: w2 is being created
    Wallet w3(0);
    w3 = w1;             // assignment operator: w3 already exists
    w2.show();
    w3.show();
    return 0;
}

```

Explanation:

The copy constructor is invoked only when a new object is being created from an existing one (Wallet w2 = w1;). The assignment operator, on the other hand, is invoked when an already-existing object is assigned a new value (w3 = w1;), and it must first release whatever memory it already owns before copying, otherwise the old memory would leak.

Problem 5: Private Constructor — Singleton Pattern

Problem Statement:

Write a C++ class PrinterManager that must have only one instance throughout the program. Prevent objects from being created directly by making the constructor private, and provide a static method to access the single shared instance.

Solution:

```

#include <iostream>
using namespace std;

class PrinterManager {
    static PrinterManager *instance;
    PrinterManager() {                // private constructor
        cout << "PrinterManager created" << endl;
    }
public:
    static PrinterManager* getInstance() {
        if (instance == nullptr)
            instance = new PrinterManager();
        return instance;
    }
    void printJob() { cout << "Printing..." << endl; }
};
PrinterManager* PrinterManager::instance = nullptr;

int main() {
    PrinterManager *p1 = PrinterManager::getInstance();
    PrinterManager *p2 = PrinterManager::getInstance();
    p1->printJob();
}

```

```
cout << (p1 == p2 ? "Same instance" : "Different instances") << endl;
// PrinterManager p3;    // ERROR: constructor is private
return 0;
}
```

Output:

```
PrinterManager created
Printing...
Same instance
```

Explanation:

Because the constructor is private, no code outside the class can create a PrinterManager object directly with new or a plain declaration. The only way to obtain an object is through the static getInstance() method, which creates it once and returns the same pointer on every subsequent call — the essence of the Singleton design pattern.

Problem 6: Constructor with Default Arguments

Problem Statement:

Write a C++ class Cylinder whose constructor takes radius and height, but where height defaults to a fixed value if the caller does not supply it. Show that this lets one constructor behave like both a one-argument and a two-argument constructor.

Solution:

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

class Cylinder {
    double radius, height;
public:
    Cylinder(double r, double h = 10.0) {    // default argument
        radius = r;
        height = h;
    }
    double volume() { return 3.1416 * radius * radius * height; }
};

int main() {
    Cylinder c1(3.0);           // height defaults to 10.0
    Cylinder c2(3.0, 20.0);     // height explicitly given
    cout << c1.volume() << endl;
    cout << c2.volume() << endl;
    return 0;
}
```

Explanation:

A default argument lets a single constructor definition serve the role of multiple overloaded constructors, as long as the omitted arguments are always the trailing ones. This reduces code duplication compared to writing separate overloaded constructors that would otherwise contain nearly identical logic.

Problem 7: Dynamic Array of Objects — new[] and delete[] with Constructors and Destructors

Problem Statement:

Write a C++ program that dynamically allocates an array of five Product objects using new[], where each object's default constructor sets an initial ID, and correctly releases the whole array using delete[]. Explain what would go wrong if delete were used instead of delete[].

Solution:

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

class Product {
    static int nextId;
    int id;
public:
    Product() : id(nextId++) {
        cout << "Product " << id << " created" << endl;
    }
    ~Product() {
        cout << "Product " << id << " destroyed" << endl;
    }
};
int Product::nextId = 1;

int main() {
    Product *arr = new Product[5];    // calls default constructor 5 times
    delete[] arr;                     // calls destructor 5 times
    return 0;
}
```

Explanation:

new Product[5] calls the default constructor once for every element, and delete[] arr calls the destructor once for every element before freeing the underlying memory block. If plain delete arr were used instead of delete[] arr, the compiler would call the destructor for only the first element and the remaining four objects' destructors would never run, causing any resources they hold to leak, besides being undefined behaviour in general.

Problem 8: Exception Safety — Resource Leak from a Failing Constructor**Problem Statement:**

Write a C++ class DataBuffer whose constructor allocates two resources in sequence, where the second allocation can throw an exception. Explain why the first resource can leak if the constructor is not written carefully, and show a corrected version.

Solution:

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

class DataBuffer {
    int *bufferA;
    int *bufferB;
public:
    DataBuffer(bool forceFail) {
        bufferA = new int[100];    // resource 1 acquired
```

```

        try {
            if (forceFail)
                throw runtime_error("allocation failed");
            bufferB = new int[100];           // resource 2 acquired
        } catch (...) {
            delete[] bufferA;                 // clean up resource 1 before rethrowing
            throw;
        }
    }
    ~DataBuffer() {
        delete[] bufferA;
        delete[] bufferB;
    }
};

int main() {
    try {
        DataBuffer d(true);
    } catch (exception &e) {
        cout << "Caught: " << e.what() << endl;
        cout << "bufferA was released safely." << endl;
    }
    return 0;
}

```

Explanation:

If a constructor throws an exception partway through, the object is considered never to have been fully constructed, so its destructor will NOT be called — meaning any resource already acquired earlier in the same constructor (bufferA here) will leak unless it is explicitly released in a catch block before the exception is rethrown, as shown above. This is why constructors that acquire multiple resources need careful exception handling, or should rely on RAII wrapper types (such as smart pointers) that clean up automatically even when an exception occurs.

Additional Practice Problems (Try Yourself)

- Write a class Point3D with all three coordinates stored as const members, initialised only through a member initializer list.
- Write a class Config with four overloaded constructors that all delegate to one master constructor using delegating constructor syntax.
- Write a class Library containing a member object of class Book, and trace the constructor/destructor call order when an array of three Library objects is created and destroyed.
- Write a class Vector2D with both a deep copy constructor and an overloaded assignment operator, and demonstrate self-assignment (`v = v;`) being handled safely.
- Write a Singleton class DatabaseConnection that lazily creates its one instance on first use and reuses it afterwards.
- Write a class Ticket whose constructor allocates two dynamic resources, and show correct clean-up if the second allocation fails partway through construction.