

# Object Oriented Programming

## Chapter 3: Constructors and Destructors

*Exam-Oriented Questions and Answers — Part 2 (Short, Broad and Coding)*

---

This is a second, more advanced set of exam-oriented questions on Constructors and Destructors, covering member initializer lists, delegating constructors, composition/nested objects, the Singleton pattern, default arguments, object slicing and exception safety — topics not repeated from Part 1.

### Section A: Short Answer Type Questions

---

**Q1. What is a member initializer list and why is it used?**

**Ans.** A member initializer list is the syntax `ClassName(args) : member1(val1), member2(val2) {}` used to initialise data members directly as the object is constructed. It is required for const members, reference members, and members without a default constructor, and is generally more efficient than assigning values inside the constructor body.

**Q2. Why can a const or reference data member not be assigned inside the constructor body?**

**Ans.** A const member cannot be assigned to after it is created, and a reference must be bound to something at the moment it comes into existence — by the time the constructor body executes, both members already exist, so only a member initializer list can legally set them.

**Q3. What is a delegating constructor?**

**Ans.** A delegating constructor is a constructor that, instead of writing its own initialisation logic, calls another constructor of the same class in its initializer list (e.g., `Box(int s) : Box(s, s, s) {}`), avoiding duplicated code across overloaded constructors.

**Q4. In what order are the constructors of member (nested) objects called relative to the enclosing class's own constructor?**

**Ans.** All member objects are constructed first, in the order they are declared in the class, and only after that does the enclosing class's own constructor body execute.

**Q5. Differentiate between the copy constructor and the overloaded assignment operator in one line each.**

**Ans.** The copy constructor initialises a brand-new object from an existing one (`Wallet b = a;`); the assignment operator updates the value of an object that already exists (`b = a;`), and must first release any resource it currently owns.

**Q6. Why must an overloaded assignment operator guard against self-assignment?**

**Ans.** Without a check such as `if (this == &src) return *this;`, an object assigned to itself (`a = a;`) could have its existing resource deleted before it is copied from — since source and destination are the same memory — leaving the object holding a dangling or empty resource.

**Q7. Why would a class deliberately make its constructor private?**

**Ans.** To prevent objects from being created freely from outside the class, most commonly in the Singleton design pattern, where only a controlled static method is allowed to create the single permitted instance.

**Q8. What are default arguments in a constructor?**

**Ans.** Default arguments are values assigned to trailing parameters in a constructor's declaration, such as `Cylinder(double r, double h = 10.0)`, which let the same constructor be called with fewer arguments while the omitted ones take their default values.

**Q9. What is object slicing?**

**Ans.** Object slicing occurs when a derived class object is assigned or passed by value to a variable of the base class type; only the base part of the object is copied and the derived-specific members are 'sliced off' and lost.

**Q10. How can object slicing be avoided?**

**Ans.** By using pointers or references to the base class instead of passing or assigning objects by value, so the full derived object is accessed rather than a truncated base-only copy.

**Q11. Can a constructor throw an exception? What happens to the object if it does?**

**Ans.** Yes, a constructor can throw an exception. If it does, the object is considered never to have been fully constructed, so its destructor is never called, meaning any resource already acquired earlier in that same constructor must be released manually before the exception propagates.

**Q12. Can a destructor be declared private? Give one situation where this is useful.**

**Ans.** Yes. A private destructor prevents an object from being deleted or going out of scope normally from outside the class, which is useful for objects that must only ever be destroyed through a controlled class method, such as certain reference-counted or factory-managed objects.

**Q13. How does a constructor differ from an ordinary member function?**

**Ans.** A constructor has the same name as the class, has no return type, and is called automatically at object creation, whereas an ordinary member function has its own name, may have a return type, and must be called explicitly by the programmer.

**Q14. Why can static data members not be initialised inside the constructor?**

**Ans.** A static data member belongs to the class as a whole rather than to any single object, and it must exist before any object is even created, so it is defined and initialised once, outside the class (or inline in modern standards), not inside a constructor that runs per object.

**Q15. Is there a limit on the number of constructors a class can have?**

**Ans.** No fixed limit is imposed by the language; a class may have as many overloaded constructors as needed, provided each one has a distinct, unambiguous parameter list.

## Section B: Broad / Descriptive Questions

---

**Q1. Explain member initializer lists with an example. Why are they mandatory for const and reference members?**

---

**Answer:**

A member initializer list is placed between a constructor's parameter list and its body, and directly initialises data members as part of construction rather than assigning to them afterwards. It is written as `ClassName(args) : member1(value1), member2(value2) { ... }`.

It is mandatory for const members and reference members because both must be given a value at the exact moment they are created — a const member cannot be assigned to later, and a reference must be bound to an object when it is defined. Assignment inside the constructor body happens after the members already exist, which the compiler will reject for these two cases.

```
class Account {
    const int accountId;
    int &linkedCounter;
public:
    Account(int id, int &counter)
        : accountId(id), linkedCounter(counter) {}
};
```

**Q2. Explain the Singleton design pattern using a private constructor. Illustrate with a program.**

---

**Answer:**

The Singleton pattern ensures that a class has only one instance throughout the program's lifetime, and provides one global point of access to it. This is achieved by making the constructor private, so that no code outside the class can call it directly, and providing a public static method that creates the one allowed instance on first use and returns the same instance on every later call.

```
class Logger {
    static Logger *instance;
    Logger() {} // private constructor
public:
    static Logger* getInstance(){
        if (!instance) instance = new Logger();
        return instance;
    }
};
Logger* Logger::instance = nullptr;
```

Since instance creation is entirely controlled through `getInstance()`, every part of the program that calls it receives a pointer to the very same object, guaranteeing there is never more than one `Logger` in existence.

**Q3. Differentiate between the copy constructor and the overloaded assignment operator, with an example showing when each is called.**

---

**Answer:**

The copy constructor is called only when a new object is being created and initialised from an existing object of the same class — for example, `Wallet w2 = w1;` or passing/returning an object by value. The assignment

operator is called when an object that already exists is given the value of another object — for example, `w3 = w1;` where `w3` was declared earlier.

Because the assignment operator's target object already owns resources, a well-written `operator=` must first release what it currently holds before copying the new values, and should guard against self-assignment; the copy constructor, in contrast, is building a fresh object that owns nothing yet, so no such release step is needed.

```
Wallet w1(500);
Wallet w2 = w1;    // copy constructor: w2 did not exist before
Wallet w3(0);
w3 = w1;           // assignment operator: w3 already existed
```

---

**Q4. What is constructor delegation? Explain with a program how it reduces duplicated initialisation code.**

---

**Answer:**

Constructor delegation (introduced in C++11) allows one constructor of a class to call another constructor of the same class in its member initializer list, instead of repeating the same initialisation statements. The delegating constructor's own body executes only after the constructor it delegates to has completely finished running.

```
class Box {
    int l, b, h;
public:
    Box(int a, int c, int d) : l(a), b(c), h(d) {}
    Box(int side) : Box(side, side, side) {}    // delegates
    Box() : Box(1, 1, 1) {}                    // delegates
};
```

Here `Box(int)` and `Box()` do not repeat the assignment logic at all; they simply hand off to the three-argument constructor, which keeps the initialisation logic in exactly one place and easier to maintain.

---

**Q5. What is object slicing? Explain the problem with an example and describe how it can be avoided.**

---

**Answer:**

Object slicing happens when an object of a derived class is assigned to, or passed/returned by value as, a variable of the base class type. Only the base-class portion of the object is copied; the extra data members introduced by the derived class are 'sliced away' and lost, and any overridden virtual behaviour reverts to the base class version.

```
class Animal { public: string name = "Animal"; };
class Dog : public Animal { public: string breed = "Labrador"; };

void printAnimal(Animal a) {    // pass by value -> slicing occurs
    cout << a.name << endl;    // 'breed' is lost here
}

int main() {
    Dog d;
    printAnimal(d);    // only the Animal part of d is copied
}
```

```
    return 0;
}
```

Slicing can be avoided by passing and returning objects through pointers or references to the base class instead of by value, since a pointer or reference does not copy the object at all and so keeps access to its full derived type.

**Q6. Explain why exception safety matters inside a constructor that acquires more than one resource. Illustrate with an example.**

**Answer:**

If a constructor acquires several resources one after another and a later acquisition throws an exception, the object as a whole is considered never to have been successfully constructed. Because of this, its destructor will never run — so any resource acquired earlier in that same constructor, before the exception was thrown, will leak unless it is released explicitly before the exception is allowed to propagate.

```
class Buffers {
    int *a, *b;
public:
    Buffers() {
        a = new int[100];
        try {
            b = new int[100];
        } catch (...) {
            delete[] a;    // release the first resource before rethrowing
            throw;
        }
    }
    ~Buffers() { delete[] a; delete[] b; }
};
```

In modern C++ this risk is usually avoided altogether by wrapping each resource in an RAII type such as a smart pointer, so that each member cleans up after itself automatically even if a later member's initialisation throws.

## Section C: Coding (Programming) Questions

---

**Q1. Write a C++ program demonstrating a member initializer list used to initialise a const data member.**

---

**Answer:**

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

class Product {
    const int productId;
public:
    Product(int id) : productId(id) {}    // initializer list required
    void show() { cout << "ID: " << productId << endl; }
};

int main() {
    Product p(101);
    p.show();
    return 0;
}
```

**Q2. Write a C++ program implementing the Singleton pattern with a private constructor.**

---

**Answer:**

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

class Settings {
    static Settings *instance;
    Settings() { cout << "Settings created" << endl; }
public:
    static Settings* getInstance() {
        if (!instance) instance = new Settings();
        return instance;
    }
};

Settings* Settings::instance = nullptr;

int main() {
    Settings *s1 = Settings::getInstance();
    Settings *s2 = Settings::getInstance();
    cout << (s1 == s2 ? "Same object" : "Different objects") << endl;
    return 0;
}
```

**Q3. Write a C++ program using delegating constructors so that two simpler constructors reuse one master constructor.**

---

**Answer:**

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

```

class Timer {
    int hrs, mins, secs;
public:
    Timer(int h, int m, int s) : hrs(h), mins(m), secs(s) {}
    Timer(int m) : Timer(0, m, 0) {}    // delegates
    Timer() : Timer(0, 0, 0) {}         // delegates
    void show() { cout << hrs << ":" << mins << ":" << secs << endl; }
};

int main() {
    Timer t1;
    Timer t2(45);
    Timer t3(1, 30, 15);
    t1.show(); t2.show(); t3.show();
    return 0;
}

```

**Q4. Write a C++ program that clearly shows the difference between the copy constructor and the assignment operator being invoked.**

**Answer:**

```

#include <iostream>
using namespace std;

class Box {
public:
    Box() { cout << "Normal constructor" << endl; }
    Box(const Box &b) { cout << "Copy constructor" << endl; }
    Box& operator=(const Box &b) {
        cout << "Assignment operator" << endl;
        return *this;
    }
};

int main() {
    Box b1;
    Box b2 = b1;    // copy constructor (b2 is new)
    Box b3;         // normal constructor
    b3 = b1;        // assignment operator (b3 already existed)
    return 0;
}

```

**Q5. Write a C++ program that demonstrates the object slicing problem when a derived object is passed to a function by value.**

**Answer:**

```

#include <iostream>
using namespace std;

class Shape {
public:
    virtual void draw() { cout << "Drawing a generic shape" << endl; }
};

class Square : public Shape {

```

```

public:
    void draw() override { cout << "Drawing a square" << endl; }
};

void render(Shape s) { s.draw(); }           // pass by value -> slicing
void renderRef(Shape &s) { s.draw(); }       // pass by reference -> no slicing

int main() {
    Square sq;
    render(sq);           // prints "Drawing a generic shape" (sliced)
    renderRef(sq);        // prints "Drawing a square" (correct)
    return 0;
}

```

**Q6. Write a C++ program using a default argument in a constructor so one constructor can act as both a one-argument and two-argument constructor.**

**Answer:**

```

#include <iostream>
using namespace std;

class Discount {
    double price, percent;
public:
    Discount(double p, double pct = 10.0) {    // default argument
        price = p;
        percent = pct;
    }
    double finalPrice() { return price - (price * percent / 100); }
};

int main() {
    Discount d1(1000);           // uses default 10% discount
    Discount d2(1000, 25);       // explicit 25% discount
    cout << d1.finalPrice() << endl;
    cout << d2.finalPrice() << endl;
    return 0;
}

```