

Object Oriented Programming

Chapter 3: Constructors and Destructors

Exam-Oriented Questions and Answers (Short, Broad and Programming Type)

This set covers the most frequently asked short-answer, broad/descriptive, and programming-based questions from the chapter Constructors and Destructors, useful for semester examinations, viva and competitive/technical exams.

Section A: Short Answer Type Questions

Q1. What is a constructor?

Ans. A constructor is a special member function of a class, having the same name as the class and no return type, which is automatically invoked when an object of that class is created, mainly to initialise its data members.

Q2. Why does a constructor not have a return type?

Ans. A constructor is not meant to return a value to the caller; its only job is to build and initialise the object, so the language specification deliberately excludes a return type, not even void.

Q3. What is a default constructor?

Ans. A default constructor is a constructor that takes no arguments (or has all parameters defaulted), so it can be invoked without supplying any values, giving the object standard initial values.

Q4. What is a parameterized constructor?

Ans. A parameterized constructor is a constructor that accepts one or more arguments, allowing the calling code to supply specific initial values for an object's data members at the time of creation.

Q5. What is a copy constructor?

Ans. A copy constructor is a constructor that initialises a new object using the values of an already existing object of the same class; it takes a reference to an object of its own class as its argument, e.g. `ClassName(const ClassName &obj)`.

Q6. When is a copy constructor invoked?

Ans. It is invoked when an object is initialised from another object of the same type, when an object is passed to a function by value, and when an object is returned by value from a function.

Q7. What is the difference between a shallow copy and a deep copy?

Ans. A shallow copy copies data members as they are, so pointer members end up pointing to the same memory in both objects; a deep copy allocates separate memory and copies the actual pointed-to data, so each object owns its own independent copy.

Q8. What is a zero-argument constructor?

Ans. It is a constructor written with an empty parameter list, `ClassName(){...}`, which is called whenever an object is declared without any initialising values.

Q9. What does the explicit keyword do when used with a constructor?

Ans. It stops a single-argument constructor from being used by the compiler for implicit type conversions, so an object of that class can then be created only through a direct, explicit call.

Q10. What is constructor overloading?

Ans. Constructor overloading is defining more than one constructor in the same class, each with a different number or type of parameters, so the compiler picks the matching one based on the arguments supplied at object creation.

Q11. Can a constructor be declared virtual? Why or why not?

Ans. No. A constructor cannot be virtual because virtual dispatch depends on an object's vtable, which is only set up once the object already exists — and the constructor's job is to create that object in the first place.

Q12. What is a destructor?

Ans. A destructor is a special member function, named with a tilde followed by the class name (e.g., `~Student()`), that is automatically called when an object's lifetime ends, mainly to release resources held by the object.

Q13. Can a class have more than one destructor?

Ans. No. Unlike constructors, a destructor cannot be overloaded; a class can have only one destructor, and it always takes no arguments.

Q14. In what order are destructors called for objects in an inheritance hierarchy?

Ans. Destructors run in the reverse order of construction — the derived class's destructor executes first, followed by the destructors of its base classes.

Q15. Why should a base class destructor be declared virtual?

Ans. So that when a derived-class object is deleted through a base-class pointer, the derived class's destructor is also correctly invoked, ensuring all resources are properly released instead of only the base part being cleaned up.

Q16. Give two practical uses of a destructor.

Ans. (i) Releasing dynamically allocated memory acquired with `new`, preventing memory leaks. (ii) Closing open files or network connections that the object was using, freeing system resources.

Q17. What is the RAII idiom and how does it relate to destructors?

Ans. RAII (Resource Acquisition Is Initialisation) is a technique where a resource is acquired in a constructor and guaranteed to be released in the matching destructor, ensuring the resource is freed automatically no matter how the object's scope is exited.

Q18. What happens if a programmer does not write a destructor for a class?

Ans. The compiler automatically generates an implicit default destructor, which simply calls the destructors of the class's member objects and base classes; it does not release manually acquired resources like heap memory.

Section B: Broad / Descriptive Questions

Q1. What is a constructor? Explain its important properties with a suitable example.

Answer:

A constructor is a special member function that is automatically executed when an object of a class is created. Its chief purpose is to initialise the data members of the object and, if necessary, to acquire resources the object will need.

Important properties: it has the same name as the class; it has no return type; it is invoked automatically and cannot be called like a normal function; a class may have several overloaded constructors; it is usually declared public; and if none is defined, the compiler supplies an implicit one.

```
class Circle {
    double radius;
public:
    Circle(double r) {    // constructor
        radius = r;
    }
    double area() { return 3.1416 * radius * radius; }
};
```

Q2. Differentiate between a default constructor and a parameterized constructor.

Answer:

A default constructor takes no arguments and gives every object the same fixed initial state, whereas a parameterized constructor takes one or more arguments and lets each object be initialised with different values supplied by the caller.

A default constructor is either written explicitly with an empty parameter list or generated implicitly by the compiler when no constructor is defined; a parameterized constructor must always be written explicitly, and once it exists, the implicit default constructor is no longer generated automatically.

```
class Item {
    int qty;
public:
    Item() { qty = 1; }           // default constructor
    Item(int q) { qty = q; }     // parameterized constructor
};
```

Q3. Explain the copy constructor with an example. Why is a user-defined (deep) copy constructor sometimes necessary?

Answer:

A copy constructor builds a new object as a copy of an existing object of the same class, and takes a reference to that object as its single argument. If the class contains only simple data members, the compiler's default member-wise copy is enough.

However, if a class holds a pointer to dynamically allocated memory, the default shallow copy makes two objects share the same memory block. When one object is destroyed and its destructor frees that memory,

the other object is left with a dangling pointer, which leads to undefined behaviour if it is used or destroyed afterwards. A user-defined deep copy constructor avoids this by allocating separate memory for the new object and copying the actual values.

```
class Buffer {
    int *data;
public:
    Buffer(int v) { data = new int(v); }
    Buffer(const Buffer &b) {          // deep copy constructor
        data = new int(*b.data);
    }
    ~Buffer() { delete data; }
};
```

Q4. What is constructor overloading? Illustrate with a program having at least three overloaded constructors.

Answer:

Constructor overloading allows a class to have multiple constructors, each with a distinct parameter list, so objects can be created in more than one way depending on the information available at the time. The compiler chooses the appropriate constructor automatically by matching the number and type of arguments used at object creation.

```
class Employee {
    string name;
    int id;
public:
    Employee() {                // zero-argument
        name = "NA"; id = 0;
    }
    Employee(int i) {           // one argument
        name = "NA"; id = i;
    }
    Employee(string n, int i) {  // two arguments
        name = n; id = i;
    }
};

int main() {
    Employee e1;
    Employee e2(101);
    Employee e3("Arun", 102);
    return 0;
}
```

Q5. What is a destructor? Explain its properties and how it differs from a constructor.

Answer:

A destructor is a special member function invoked automatically when an object's lifetime ends, used mainly to release resources the object was holding. It is written as the class name preceded by a tilde, for example ~Employee().

Differences from a constructor: a destructor takes no arguments and cannot be overloaded, while a constructor may take arguments and may be overloaded; a destructor frees resources at the end of an object's life, while a constructor acquires/initialises them at the start; and a class can have only one destructor but many constructors.

```
class Logger {
public:
    Logger() { cout << "Object created"; }
    ~Logger() { cout << "Object destroyed"; }    // destructor
};
```

Q6. Explain, with examples, any four practical uses of destructors.

Answer:

- Releasing dynamically allocated memory obtained with new, so that the program does not suffer memory leaks.
- Closing files, sockets or database connections that were opened during the object's lifetime.
- Releasing locks or other synchronisation resources acquired by the object, preventing deadlocks in multi-threaded programs.
- Decrementing a static object-count variable, or logging that an object has gone out of scope, which helps in debugging object lifetimes.

Q7. Explain why a base class destructor should be declared virtual when a class hierarchy involves dynamic (polymorphic) deletion.

Answer:

When an object of a derived class is deleted through a pointer of the base class type, and the base class destructor is not virtual, only the base class's destructor executes; the derived class's own destructor is skipped, so any resources it was responsible for releasing are leaked. Declaring the base class destructor virtual ensures that the correct, most-derived destructor is called first, followed automatically by the destructors further up the hierarchy, guaranteeing complete and correct clean-up.

```
class Base {
public:
    virtual ~Base() { cout << "Base destroyed"; }
};
class Derived : public Base {
public:
    ~Derived() { cout << "Derived destroyed"; }
};

int main() {
    Base *p = new Derived();
    delete p;    // both destructors run because ~Base() is virtual
    return 0;
}
```

Section C: Programming / Coding Questions

Q1. Write a C++ program to demonstrate the use of a default and a parameterized constructor in the same class.

Answer:

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

class Student {
    int roll;
public:
    Student() { roll = 0; }           // default constructor
    Student(int r) { roll = r; }     // parameterized constructor
    void show() { cout << "Roll: " << roll << endl; }
};

int main() {
    Student s1;           // calls default constructor
    Student s2(25);       // calls parameterized constructor
    s1.show();
    s2.show();
    return 0;
}
```

Output: Roll: 0, followed by Roll: 25 — each object is initialised by whichever constructor matches the arguments used to create it.

Q2. Write a C++ program that demonstrates a deep copy constructor for a class containing a pointer data member.

Answer:

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

class Marks {
    int *score;
public:
    Marks(int s) { score = new int(s); }
    Marks(const Marks &m) {           // deep copy constructor
        score = new int(*m.score);
    }
    void set(int s) { *score = s; }
    void show() { cout << "Score: " << *score << endl; }
    ~Marks() { delete score; }
};

int main() {
    Marks m1(80);
    Marks m2 = m1; // deep copy constructor invoked
    m2.set(95);    // modifies only m2's own copy
    m1.show();     // Score: 80
    m2.show();     // Score: 95
    return 0;
}
```

```
}
```

Because the copy constructor allocates a fresh block of memory for m2, changing m2's score does not affect m1, which proves that a genuine deep copy has taken place.

Q3. Write a C++ program using constructor overloading to initialise objects of a class Rectangle in three different ways.

Answer:

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

class Rectangle {
    int length, breadth;
public:
    Rectangle() { length = breadth = 0; }
    Rectangle(int side) { length = breadth = side; }           // square
    Rectangle(int l, int b) { length = l; breadth = b; }
    int area() { return length * breadth; }
};

int main() {
    Rectangle r1;
    Rectangle r2(5);
    Rectangle r3(8, 4);
    cout << r1.area() << " " << r2.area() << " " << r3.area();
    return 0;
}
```

Q4. Write a C++ program to show that a destructor is called automatically and in reverse order for objects in a derived class hierarchy.

Answer:

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

class Base {
public:
    Base() { cout << "Base constructor" << endl; }
    ~Base() { cout << "Base destructor" << endl; }
};

class Derived : public Base {
public:
    Derived() { cout << "Derived constructor" << endl; }
    ~Derived() { cout << "Derived destructor" << endl; }
};

int main() {
    Derived d;
    return 0;
}
```

Output order: Base constructor, Derived constructor (construction runs base-to-derived), then, as the program ends, Derived destructor, Base destructor (destruction runs derived-to-base — the exact reverse).

Q5. Write a C++ program using the explicit keyword to prevent an unwanted implicit conversion in a single-argument constructor.

Answer:

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

class Distance {
    int metres;
public:
    explicit Distance(int m) { metres = m; }
    int get() { return metres; }
};

void printDistance(Distance d) {
    cout << d.get() << " m" << endl;
}

int main() {
    Distance d1(50);           // OK - direct initialisation
    printDistance(d1);         // OK - already a Distance object
    // printDistance(50);      // ERROR - implicit int-to-Distance blocked
    return 0;
}
```

Marking the constructor explicit forces the caller to create a Distance object directly, rather than allowing the compiler to silently convert a plain integer into one.

Q6. Write a C++ program that uses a destructor to close a dynamically opened resource and demonstrates the RAII idiom.

Answer:

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

class Resource {
    int *buffer;
public:
    Resource(int size) {
        buffer = new int[size];
        cout << "Resource acquired" << endl;
    }
    ~Resource() {                // RAII: release in destructor
        delete[] buffer;
        cout << "Resource released" << endl;
    }
};

int main() {
    {
        Resource r(10);          // acquired here
    }                             // automatically released here
    cout << "End of main" << endl;
    return 0;
}
```

The moment `r` goes out of scope at the closing brace, its destructor runs automatically and frees the memory, so the program never needs to remember to release the resource manually.