Object Oriented Programming

Chapter 3: Constructors and Destructors

3.1 Concept of Constructor (Default, Parameterized, Copy), Zero-Argument and Explicit / Overloaded Constructors, Destructors and Their Properties, Uses of Destructors

3.1.1 Introduction to Constructors

In object oriented programming, a class describes the structure and behaviour of its objects, but the data members of an object must be given valid initial values before the object can be used safely. A constructor is a special member function of a class that is automatically invoked whenever an object of that class is created. Its main purpose is to initialise the data members of the object and, if required, to allocate resources such as memory, files or network handles that the object will need during its lifetime.

A constructor is easily recognised because it always carries the same name as the class itself, and it never has a return type — not even void. Since it does not return a value, it cannot be called like an ordinary function; it is invoked implicitly by the compiler at the moment of object creation.

Key Properties of a Constructor

- The constructor has the same name as its class.
- It has no return type, not even void.
- It is invoked automatically when an object is created; it cannot be called explicitly like a normal member function.
- A class may have more than one constructor, provided each has a different signature (this is called constructor overloading).
- A constructor can be declared with any access specifier, but it is almost always declared public so that objects can be created from outside the class.
- Constructors cannot be declared virtual and cannot be inherited, although a derived class can call a base class constructor.
- If the programmer does not define any constructor, the compiler automatically supplies a default constructor.

3.1.2 Default Constructor

A default constructor is a constructor that either takes no arguments at all, or one in which every parameter has a default value so that it can be called with no arguments. If a class does not explicitly define any constructor, the compiler automatically generates an implicit default constructor. This compiler-generated constructor initialises data members to their default values (for example, zero for built-in numeric types) and calls the default constructors of any member objects or base classes.

The moment a programmer defines any constructor of their own, the compiler stops supplying the implicit default constructor. If a no-argument construction is still needed after other constructors have been added, the programmer must define it explicitly.

```
class Student {
    int roll;
    string name;
public:
    // user-defined default constructor
    Student() {
        roll = 0;
        name = "Unknown";
    }
};

int main() {
    Student s1;      // default constructor is called
    return 0;
}
```

A default constructor ensures that every object starts in a well-defined, predictable state, even when the caller supplies no initial values.

3.1.3 Parameterized Constructor

A parameterized constructor is a constructor that accepts one or more arguments. It allows different objects of the same class to be initialised with different values at the time of creation, instead of relying on fixed default values. This is one of the most common ways of initialising data members efficiently, because the required values are supplied directly at the point where the object is defined.

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

int main() {
    Rectangle r1(10, 5);   // values passed at creation
    Rectangle r2 = Rectangle(8, 3);
    return 0;
}
```

Once a parameterized constructor is defined, the compiler-generated default constructor is no longer available automatically; if objects are also to be created without arguments, a separate default constructor (or default values for the parameters) must be provided.

3.1.4 Copy Constructor

A copy constructor is a special constructor that creates a new object as a copy of an existing object of the same class. It takes a single argument, which is a reference to an object of the same class, conventionally written as ClassName(const ClassName &obj). It is invoked automatically in situations such as when an object

is initialised from another object, when an object is passed to a function by value, and when an object is returned by value from a function.

If the programmer does not define a copy constructor, the compiler provides an implicit one that performs a member-wise (shallow) copy of the data members. This default behaviour is adequate for simple classes, but it becomes dangerous for classes that contain pointers to dynamically allocated memory, because a shallow copy makes two objects point to the same memory block; when one object is destroyed, the other is left holding a dangling pointer. For such classes, a deep copy constructor must be written explicitly so that the copy allocates its own independent memory.

```
class Marks {
    int *score;
public:
    Marks(int s) {
        score = new int(s);
    }
    // deep copy constructor
    Marks(const Marks &m) {
        score = new int(*m.score);
    }
    ~Marks() { delete score; }
};

int main() {
    Marks m1(90);
    Marks m2 = m1;   // copy constructor invoked
    return 0;
}
```

In short, the copy constructor is essential for correct object duplication whenever a class manages a resource — such as heap memory, a file handle, or a network connection — that must not be shared blindly between two objects.

3.1.5 Zero-Argument Constructor and Explicit Constructors

Zero-Argument Constructor

A zero-argument constructor is simply a constructor that takes no parameters at all — it is the most direct form of a default constructor. It is written as ClassName() { ... } and is called whenever an object is declared without any initialising arguments, for example Student s;. It is commonly used to set up sensible starting values or to perform basic setup work such as opening a resource or initialising counters.

Explicit Constructors

When a constructor can be called with a single argument, the compiler is normally allowed to use it for implicit type conversion — that is, it may silently convert a value of the argument's type into an object of the class whenever such an object is expected. This is convenient in some situations, but it can also lead to unintended and confusing conversions being performed automatically.

The keyword explicit is placed before a single-argument constructor's declaration to switch this behaviour off. An explicit constructor can still be used to create an object directly, but it can no longer be used by the compiler for implicit conversions; the object must be created with a clear, direct call.

```
class Box {
    int side;
public:
    explicit Box(int s) { side = s; }
};

void printBox(Box b) { /* ... */ }

int main() {
    Box b1(5);        // OK - direct initialisation
    Box b2 = Box(5);  // OK - direct call
    // Box b3 = 5;    // ERROR - implicit conversion blocked
    // printBox(5);   // ERROR - implicit conversion blocked
    return 0;
}
```

Using explicit is considered good programming practice for single-argument constructors, since it prevents the compiler from creating objects through unexpected, hidden conversions and makes the programmer's intent visible in the code.

3.1.6 Overloaded Constructors (Constructor Overloading)

A class is not restricted to having only one constructor. Constructor overloading is the technique of defining multiple constructors within the same class, each with a different number or type of parameters. When an object is created, the compiler examines the arguments supplied and automatically selects the constructor whose signature matches, in the same way that ordinary overloaded functions are resolved.

Constructor overloading gives a class flexibility, since objects can then be initialised in several different ways depending on what information is available at the point of creation.

```
class Point {
    int x, y;
public:
    Point() { x = 0; y = 0; }              // zero-argument
    Point(int a) { x = a; y = a; }         // one argument
    Point(int a, int b) { x = a; y = b; }  // two arguments
    Point(const Point &p) { x = p.x; y = p.y; } // copy constructor
};

int main() {
    Point p1;          // calls Point()
    Point p2(4);       // calls Point(int)
    Point p3(4, 7);    // calls Point(int, int)
    Point p4 = p3;     // calls copy constructor
    return 0;
}
```

Overload resolution for constructors follows the normal function-overloading rules of the language: the number of arguments, their order, and their data types must together uniquely identify one constructor, otherwise the compiler reports an ambiguous call.

3.1.7 Destructors and Their Properties

A destructor is a special member function that is automatically invoked when an object's lifetime ends — for example, when a locally declared object goes out of scope, when an object created with new is explicitly deleted, or when the program terminates for a globally declared object. Its principal responsibility is to perform clean-up work, most importantly releasing any resources that the constructor, or any other member function, acquired on behalf of the object, such as dynamically allocated memory, open files, network sockets or database connections.

Properties of a Destructor

- A destructor has the same name as the class, preceded by a tilde (~) symbol, for example ~Student().
- It takes no arguments and cannot be overloaded — a class can have only one destructor.
- It has no return type, not even void.
- It cannot be declared static or const, since it always acts on the specific object being destroyed.
- It is called automatically by the compiler/runtime; it should not, in normal circumstances, be invoked explicitly by the programmer.
- If no destructor is defined by the programmer, the compiler supplies an implicit default destructor.
- In a class hierarchy, destructors are executed in the reverse order of construction — the most-derived class's destructor runs first, followed by its base classes'.
- A base class destructor that may be accessed through a pointer to the base class should be declared virtual, so that deleting a derived-class object through a base-class pointer correctly invokes the derived destructor as well.

```
class FileHandler {
    FILE *fp;
public:
    FileHandler(const char *name) {
        fp = fopen(name, "w");
    }
    ~FileHandler() {          // destructor
        if (fp) fclose(fp);   // release the resource
    }
};
```

3.1.8 Uses of Destructors

Destructors are central to safe and predictable resource management in object oriented programming, particularly in languages such as C++ that do not provide automatic garbage collection. Their typical uses include the following.

- Releasing dynamically allocated memory that was obtained with new inside a constructor or other member function, preventing memory leaks.
- Closing files, sockets, pipes or database connections that were opened during the object's lifetime, so that system resources are not held longer than necessary.
- Releasing locks, semaphores or other synchronisation objects that the object may have acquired, avoiding deadlocks.
- Performing final bookkeeping actions, such as decrementing a static counter that tracks how many objects of a class currently exist.
- Logging or auditing that an object has been destroyed, which can be useful for debugging the lifetime of objects in a large program.
- Ensuring that the RAII (Resource Acquisition Is Initialisation) idiom works correctly, where a resource acquired in the constructor is guaranteed to be released in the destructor regardless of how control leaves the enclosing scope.

Because destructors run automatically and deterministically at the end of an object's life, they free the programmer from having to remember to release resources manually at every possible exit point of a program, and they make classes safer and easier to use correctly.

Multiple Choice Questions (MCQs)

1. Which of the following statements about a constructor is correct?

(a) It has a return type of void
(b) It has the same name as the class and no return type
(c) It must always take at least one argument
(d) It can be called explicitly like a normal function
Answer: (b)

2. A constructor that can be called with zero arguments is known as a:

(a) Copy constructor
(b) Parameterized constructor
(c) Default constructor
(d) Destructor
Answer: (c)

3. What happens if a class defines no constructor at all?

(a) The program fails to compile
(b) The compiler provides an implicit default constructor
(c) Objects of the class cannot be created
(d) A destructor is generated instead
Answer: (b)

4. A parameterized constructor is mainly used to:

(a) Destroy an object
(b) Initialise data members with values supplied at object creation
(c) Prevent object copying
(d) Overload operators
Answer: (b)

5. A copy constructor takes as its argument:

(a) No argument
(b) Two objects of different classes
(c) A reference to an object of the same class
(d) An integer value
Answer: (c)

6. The default (compiler-generated) copy constructor performs:

(a) A deep copy
(b) A shallow (member-wise) copy
(c) No copying at all
(d) A copy of only static members
Answer: (b)

7. A deep copy constructor is especially necessary when a class contains:

(a) Only integer data members
(b) Pointers to dynamically allocated memory
(c) Only static members
(d) No data members at all
Answer: (b)

8. The keyword used to prevent a single-argument constructor from being used in implicit conversions is:

(a) static
(b) virtual
(c) explicit
(d) inline
Answer: (c)

9. Constructor overloading means:

(a) A class has multiple destructors
(b) A class has several constructors with different signatures
(c) A constructor calls itself recursively
(d) Only one constructor is allowed per class
Answer: (b)

10. Which of the following correctly declares a destructor for a class named Book?

(a) Book destructor()
(b) ~Book()
(c) delete Book()
(d) void ~Book()
Answer: (b)

11. A destructor is automatically called when:

(a) A new object is created
(b) An object goes out of scope or is explicitly deleted
(c) A class is declared
(d) A header file is included
Answer: (b)

12. Which of the following is TRUE about destructors?

(a) They can be overloaded
(b) They can take arguments
(c) A class can have only one destructor
(d) They must return an integer
Answer: (c)

13. In a class hierarchy, destructors are invoked:

(a) In the same order as constructors
(b) In the reverse order of construction
(c) In random order
(d) Only for the base class
Answer: (b)

14. One of the main uses of a destructor is to:

(a) Allocate new memory for the object
(b) Release resources such as memory or file handles held by the object
(c) Overload the assignment operator
(d) Increase the size of the class
Answer: (b)

15. Which constructor is automatically invoked when an object is passed to a function by value?

(a) Default constructor

(b) Copy constructor
(c) Destructor
(d) Explicit constructor
Answer: (b)

Answer Key

1-b, 2-c, 3-b, 4-b, 5-c, 6-b, 7-b, 8-c, 9-b, 10-b, 11-b, 12-c, 13-b, 14-b, 15-b

Broad / Descriptive Questions

1. What is a constructor? Explain its important properties with the help of a suitable example.

2. Differentiate between a default constructor and a parameterized constructor. Illustrate both with a program.

3. What is a copy constructor? Explain, with an example, why a deep copy constructor is necessary for a class that contains pointer data members.

4. Explain the concept of constructor overloading with a suitable C++ program having at least three overloaded constructors.

5. What is the purpose of the explicit keyword when applied to a single-argument constructor? Explain with an example how it prevents unwanted implicit conversions.

6. What is a destructor? List its important properties and explain how it differs from a constructor.

7. Explain, with examples, any four practical uses of a destructor in resource management.

8. In what order are constructors and destructors called in a class hierarchy involving inheritance? Justify your answer.

9. Why should the compiler-generated default constructor and copy constructor sometimes be considered insufficient? Support your answer with an example.

10. Write a C++ program that defines a class with a parameterized constructor, a copy constructor, and a destructor, and demonstrate the invocation of each.