

Object Oriented Programming

Chapter 3: Constructors and Destructors

Exam-Oriented Questions and Answers — Part 3 (Short, Broad and Coding)

This is a third set of exam-oriented questions on Constructors and Destructors, covering conversion constructors, uniform (brace) initialisation, move constructors, constructor order in multiple inheritance, inherited constructors, and temporary object lifetimes — all new topics not repeated from Parts 1 and 2.

Section A: Short Answer Type Questions

Q1. What is a conversion constructor?

Ans. A conversion constructor is any constructor that can be called with a single argument and is therefore usable by the compiler to implicitly convert a value of that argument's type into an object of the class, unless it is marked explicit.

Q2. What is uniform (brace) initialisation and how does it relate to constructors?

Ans. Uniform initialisation, introduced in C++11, lets an object be initialised using curly braces, e.g. `Point p{3, 4};`, which calls the matching constructor just like the traditional `Point p(3, 4);` syntax, but with the added benefit of preventing narrowing conversions such as passing a double where an int is expected.

Q3. What is a move constructor?

Ans. A move constructor (C++11) initialises a new object by transferring ownership of resources from a temporary (rvalue) object, rather than copying them, leaving the source object in a valid but emptied state. It is written as `ClassName(ClassName &&src)`.

Q4. When is a move constructor preferred over a copy constructor by the compiler?

Ans. When the source object is a temporary or is explicitly cast with `std::move()`, the compiler prefers a move constructor if one is available, since resources can simply be transferred instead of being duplicated, which is more efficient.

Q5. In what order are base class constructors executed in multiple inheritance?

Ans. They are executed strictly in the order the base classes are listed in the derived class's declaration, not in the order they appear in the derived class's constructor initializer list.

Q6. Can a constructor be declared as a friend of another class?

Ans. A constructor itself cannot be declared friend, but a class's constructor can be granted friend access by another class, or a global function can be declared a friend so that it may call a private constructor of that class.

Q7. What is a temporary (anonymous) object? When is its destructor called?

Ans. A temporary object is an unnamed object created implicitly, for example as an intermediate result of an expression like `Point(2,3).show();`. Its destructor is called automatically at the end of the full expression in which it was created, unless its lifetime is extended by binding it to a reference.

Q8. Why must even a const object be created through a normal (non-const) constructor?

Ans. The const-ness of an object only takes effect once construction is complete; while the constructor is running, the object is still being built, so the constructor itself must be an ordinary member function, not a const one, even though the finished object will be const.

Q9. What is the difference between initialisation and assignment as applied to constructors?

Ans. Initialisation gives a data member its very first value at the moment it is created (as in a member initializer list); assignment changes the value of a member that already has one. A member initializer list performs true initialisation, while writing `radius = r;` inside the constructor body actually default-constructs the member first and then assigns to it, which can be less efficient.

Q10. What is a trivial constructor?

Ans. A trivial constructor is a compiler-generated constructor that performs no action beyond what is strictly necessary (it does not initialise members to any particular value, call any user-defined logic, or have virtual functions/base classes involved) — it exists purely so the object's storage is considered constructed.

Q11. What are inherited constructors, and how are they declared in C++11?

Ans. Inherited constructors let a derived class reuse its base class's constructors directly, using the declaration using `Base::Base;` inside the derived class, instead of writing near-identical forwarding constructors by hand.

Q12. Why does declaring using `Base::Base;` not automatically initialise the derived class's own new data members?

Ans. Because the inherited constructor only knows how to initialise the base part of the object; any additional data members introduced by the derived class are default-initialised separately, so if they need specific values, the derived class must still provide its own constructor for that purpose.

Section B: Broad / Descriptive Questions

Q1. What is a move constructor? Explain, with an example, how it differs from a copy constructor and why it improves performance.

Answer:

A move constructor initialises a new object by taking ownership of the resources of a temporary (rvalue) object instead of duplicating them. It is declared as `ClassName(ClassName &&src)`, where the double ampersand indicates an rvalue reference. Inside it, the pointer (or other resource handle) is copied from the source and then the source's pointer is set to `nullptr`, so that when the temporary is eventually destroyed, its destructor does not free memory that now belongs to the new object.

This is more efficient than a copy constructor, which would allocate a fresh block of memory and copy every element — an unnecessary cost when the source object was only a temporary that was about to be discarded anyway.

```
class Buffer {
    int *data;
public:
    Buffer(int size) { data = new int[size]; }
    Buffer(Buffer &&src) noexcept {           // move constructor
        data = src.data;
        src.data = nullptr;
    }
    ~Buffer() { delete[] data; }
};

Buffer makeBuffer() { return Buffer(100); }
int main() {
    Buffer b = makeBuffer();    // move constructor invoked
    return 0;
}
```

Q2. Explain, with an example, the order in which base class constructors and destructors execute when a derived class uses multiple inheritance.

Answer:

When a class inherits from more than one base class, the base class constructors run in the order the base classes are listed in the derived class's own declaration (for example, `class C : public A, public B`), regardless of the order the corresponding calls appear in the derived class's initializer list. The derived class's own constructor body runs only after all base classes have been fully constructed.

Destruction always happens in the exact reverse order: the derived class's destructor body runs first, followed by the base class destructors in reverse of their construction order.

```
class A { public: A() { cout << "A"; } ~A() { cout << "~A"; } };
class B { public: B() { cout << "B"; } ~B() { cout << "~B"; } };
class C : public A, public B {
public:
    C() { cout << "C"; }
    ~C() { cout << "~C"; }
```

```
};  
// Creating a C object prints: A B C  
// Destroying it prints:      ~C ~B ~A
```

Q3. Explain conversion constructors and implicit type conversion, and describe how the `explicit` keyword changes this behaviour.

Answer:

Any constructor that can be invoked with a single argument acts as a conversion constructor by default: the compiler is permitted to use it silently whenever a value of the argument's type is needed in a place where an object of the class is expected — for example, passing a plain number to a function that expects an object, or initialising an object with an = sign and a value of the wrong type.

While convenient in some designs, this can also produce confusing and unintended conversions. Marking a single-argument constructor `explicit` switches this behaviour off: the constructor can still be called directly to create an object, but the compiler will no longer use it automatically for implicit conversions, so any such conversion must be written out clearly by the programmer.

```
class Meters {  
    double value;  
public:  
    Meters(double v) { value = v; }    // conversion constructor  
};  
void printLength(Meters m) { /* ... */ }  
  
int main() {  
    printLength(5.0);    // OK: 5.0 is implicitly converted to Meters  
    return 0;  
}
```

Q4. Explain uniform (brace) initialisation in C++11 and how it improves upon traditional constructor-call syntax.

Answer:

Uniform initialisation allows any object — whether a built-in type, an array, or a class object — to be initialised using curly braces, applying one consistent syntax across the language: `Point p{3, 4};` calls the matching two-argument constructor exactly as `Point p(3, 4);` would.

Its main advantage over the traditional parenthesis syntax is that it disallows narrowing conversions: passing a double where the constructor expects an int, for example, produces a compile-time error with brace initialisation, whereas the parenthesis form would silently truncate the value. It also removes an ambiguity in the older syntax known as the 'most vexing parse', where `Point p();` could be mistaken for a function declaration rather than an object definition.

```
class Point {  
    int x, y;  
public:  
    Point(int a, int b) : x(a), y(b) {}  
};
```

```
int main() {
    Point p1{3, 4};    // uniform initialisation
    // Point p2{3.5, 4}; // ERROR: narrowing conversion double -> int
    return 0;
}
```

Q5. Explain temporary (anonymous) objects with an example, and describe when their destructors are called.

Answer:

A temporary object is an unnamed object created implicitly during the evaluation of an expression — for instance, when a class type is constructed inline and used immediately, such as `Point(2, 3).show();`, or when a function returns an object by value into an expression rather than into a named variable. Because it has no name, the programmer cannot refer to it again after the expression finishes.

Its destructor is called automatically at the end of the full expression in which it was created, once it is no longer needed — unless the temporary is bound to a `const` reference or an `rvalue` reference, which extends its lifetime to match that of the reference.

```
class Point {
    int x, y;
public:
    Point(int a, int b) : x(a), y(b) {
        cout << "Constructed" << endl;
    }
    void show() { cout << x << ", " << y << endl; }
    ~Point() { cout << "Destroyed" << endl; }
};

int main() {
    Point(5, 6).show();    // temporary object: constructed, used, destroyed
    cout << "Next line" << endl;
    return 0;
}
```

Q6. What are inherited constructors in C++11? Explain with an example, and mention one limitation.

Answer:

Inherited constructors let a derived class reuse a base class's constructors without having to write matching forwarding constructors by hand. This is done with a `using` declaration inside the derived class, for example `using Base::Base;`, after which the derived class can be constructed using any constructor signature the base class provides.

```
class Base {
public:
    Base(int x) { cout << "Base(" << x << ")" << endl; }
};

class Derived : public Base {
public:
    using Base::Base;    // inherit Base's constructors
};
```

```
int main() {  
    Derived d(10);    // calls Base(int) via the inherited constructor  
    return 0;  
}
```

The main limitation is that an inherited constructor only initialises the base part of the object; if Derived introduces its own additional data members that need specific initial values, those members will only receive their default initialisation through the inherited constructor, so a dedicated Derived constructor is still needed if custom values are required for them.

Section C: Coding (Programming) Questions

Q1. Write a C++ program that defines both a copy constructor and a move constructor for a class, and shows which one gets called in each case.

Answer:

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

class Data {
    int *value;
public:
    Data(int v) { value = new int(v); }
    Data(const Data &src) {                // copy constructor
        value = new int(*src.value);
        cout << "Copy constructor" << endl;
    }
    Data(Data &&src) noexcept {             // move constructor
        value = src.value;
        src.value = nullptr;
        cout << "Move constructor" << endl;
    }
    ~Data() { delete value; }
};

Data makeData() { return Data(42); }

int main() {
    Data a(10);
    Data b = a;                // copy constructor (a is an lvalue)
    Data c = makeData();       // move constructor (temporary is an rvalue)
    return 0;
}
```

Q2. Write a C++ program with three base classes to demonstrate the exact order in which their constructors and destructors run under multiple inheritance.

Answer:

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

class Wheels { public: Wheels() { cout << "Wheels\n"; } ~Wheels() { cout <<
    "~Wheels\n"; } };
class Engine { public: Engine() { cout << "Engine\n"; } ~Engine() { cout <<
    "~Engine\n"; } };
class Body { public: Body() { cout << "Body\n"; } ~Body() { cout <<
    "~Body\n"; } };

class Car : public Wheels, public Engine, public Body {
public:
    Car() { cout << "Car\n"; }
    ~Car() { cout << "~Car\n"; }
};
```

```
int main() {
    Car c;    // construction order: Wheels, Engine, Body, Car
    return 0;
}    // destruction order: ~Car, ~Body, ~Engine, ~Wheels
```

Q3. Write a C++ program showing a conversion constructor causing an implicit conversion, then fix it using the explicit keyword.

Answer:

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

class Fraction {
    double value;
public:
    Fraction(double v) { value = v; }    // conversion constructor
    double get() { return value; }
};

void showFraction(Fraction f) { cout << f.get() << endl; }

int main() {
    showFraction(0.75);    // implicit conversion: double -> Fraction
    return 0;
}
```

Adding explicit before Fraction(double v) would make showFraction(0.75); a compile-time error, requiring showFraction(Fraction(0.75)); instead.

Q4. Write a C++ program demonstrating uniform (brace) initialisation and how it catches a narrowing conversion error.

Answer:

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

class Age {
    int years;
public:
    Age(int y) : years(y) {}
    void show() { cout << years << endl; }
};

int main() {
    Age a1{25};            // uniform initialisation - OK
    Age a2(25);            // traditional initialisation - also OK
    // Age a3{25.5};      // ERROR: narrowing double -> int is rejected
    a1.show();
    return 0;
}
```

Q5. Write a C++ program that shows a temporary object being constructed and destroyed within a single expression.

Answer:

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

class Greeting {
public:
    Greeting() { cout << "Greeting created" << endl; }
    void say() { cout << "Hello!" << endl; }
    ~Greeting() { cout << "Greeting destroyed" << endl; }
};

int main() {
    Greeting().say();    // temporary object used then destroyed
    cout << "main continues" << endl;
    return 0;
}
```

Q6. Write a C++ program using an inherited constructor (using Base::Base;) so a derived class reuses its base class's constructor.

Answer:

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

class Vehicle {
    int wheels;
public:
    Vehicle(int w) : wheels(w) {
        cout << "Vehicle with " << wheels << " wheels" << endl;
    }
};

class Bike : public Vehicle {
public:
    using Vehicle::Vehicle;    // inherit Vehicle's constructor
};

int main() {
    Bike b(2);    // calls the inherited Vehicle(int) constructor
    return 0;
}
```