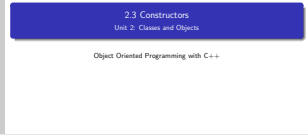
2.3 Constructors

Unit 2: Classes and Objects

Object Oriented Programming with C++



Lecture 12: Constructors in C++

- Unit 2: Classes and Objects
- Topic 2.3: Constructors
- Course: Object Oriented Programming with C++

Lecture 12: Constructors in C++
- Unit 2: Classes and Objects
- Topic 2.3: Constructors
- Course: Object Oriented Programming with C++

Welcome everyone to Lecture 12. Today we are diving into a fundamental concept in Object-Oriented Programming: Constructors. In our last class, we learned how to create classes and instantiate objects. But what happens at the exact moment an object is born? How do we ensure it starts its life in a valid state? That is exactly what constructors are for.

Learning Objectives

- Understand what a constructor is and why it's necessary.
- Learn the key characteristics of constructors in C++.
- Explore Default, Parameterized, and Copy constructors.
- Learn how to use multiple constructors in a single class (Constructor Overloading).

Learning Objectives
- Understand what a constructor is and why it's necessary.
- Learn the key characteristics of constructors in C++.
- Explore Default, Parameterized, and Copy constructors.
- Learn how to use multiple constructors in a single class (Constructor Overloading).

By the end of this 50-minute session, you should be perfectly comfortable defining your own constructors. We will look at the three main types of constructors in C++ and understand how overloading allows a single class to have multiple constructors, giving developers flexibility when creating objects.

What is a Constructor?

- A constructor is a special member function of a class.
- It is automatically invoked whenever an object of its class is created.
- Its primary purpose is to initialize the data members of new objects.
- It allocates memory and sets default or provided values before the object is used.

What is a Constructor?
- A constructor is a special member function of a class.
- It is automatically invoked whenever an object of its class is created.
- Its primary purpose is to initialize the data members of new objects.
- It allocates memory and sets default or provided values before the object is used.

Think of a constructor as a setup routine. When you buy a new smartphone, the factory reset or initial setup wizard configures the language, time zone, and user account before you can use it. Similarly, a constructor sets up an object in memory, ensuring its variables don't contain garbage values.

Characteristics of Constructors

- Name: Must have the exact same name as the class itself.
- Return Type: They do not have a return type, not even 'void'.
- Access Specifier: Usually declared in the 'public' section (though private is possible for specific design patterns like Singleton).
- Invocation: Cannot be called directly like standard methods; invoked implicitly.
- Inheritance: They cannot be inherited, though a derived class can call a base class constructor.

Characteristics of Constructors
- Name: Must have the exact same name as the class itself.
- Return Type: They do not have a return type, not even 'void'.
- Access Specifier: Usually declared in the 'public' section (though private is possible for specific design patterns like Singleton).
- Invocation: Cannot be called directly like standard methods; invoked implicitly.
- Inheritance: They cannot be inherited, though a derived class can call a base class constructor.

These rules are strict in C++. If you add a return type, even void, the compiler will treat it as a standard member function, not a constructor. Always ensure the name matches the class perfectly, including case. We typically place them in the public section because if they were private, external code wouldn't be able to instantiate the object.

Types of Constructors

- C++ provides three primary types of constructors:
- 1. Default Constructor: Takes no arguments.
- 2. Parameterized Constructor: Takes one or more arguments.
- 3. Copy Constructor: Initializes an object using another object of the same class.

Types of Constructors
C++ provides three primary types of constructors:
1. Default Constructor: Takes no arguments.
2. Parameterized Constructor: Takes one or more arguments.
3. Copy Constructor: Initializes an object using another object of the same class.

Depending on how we want to create our objects, we will use different types of constructors. Sometimes we want a blank slate (Default), sometimes we know exactly what data the object should hold right away (Parameterized), and sometimes we want to duplicate an existing object (Copy).

1. The Default Constructor

- A constructor that takes no parameters.
- If no constructor is defined by the programmer, the C++ compiler automatically provides an implicit default constructor.
- Used when creating an object without passing any initial values (e.g., `Student s1;`).
- Often used to assign baseline default values (like zero or empty strings) to member variables.

1. The Default Constructor
- A constructor that takes no parameters.
- If no constructor is defined by the programmer, the C++ compiler automatically provides an implicit default constructor.
- Used when creating an object without passing any initial values (e.g., Student s1;).
- Often used to assign baseline default values (like zero or empty strings) to member variables.

The default constructor is your safety net. If you don't write any constructor at all, C++ quietly generates one for you that allocates memory, though it leaves primitive types uninitialized (containing garbage). If you define one, you take control of setting those baseline values.

Default Constructor: Code Example

- class Student {
- private:
- int rollNo;
- public:
- // Default Constructor
- Student() {
- rollNo = 0;
- cout ¡¡ "Default Constructor called!" ¡¡ endl;
- }
- };
- // Usage:
- Student s1; // Calls Default Constructor

Default Constructor: Code Example
class Student {
private:
int rollNo;
public:
// Default Constructor
Student() {
rollNo = 0;
cout ¡¡ "Default Constructor called!" ¡¡ endl;
}
};
// Usage:
Student s1; // Calls Default Constructor

Here we see a class named Student. Inside the public section, we have a function named Student() with no return type. This is our explicit default constructor. When we declare 'Student s1;', this constructor is triggered immediately, setting rollNo to 0 and printing our debug message.

2. The Parameterized Constructor

- A constructor that accepts one or more arguments.
- Allows initialization of an object with specific, user-defined values at the time of creation.
- Highly useful for passing dynamic data to an object when it's instantiated.
- Note: Once you define a parameterized constructor, the compiler will NO LONGER provide an implicit default constructor.

2. The Parameterized Constructor
- A constructor that accepts one or more arguments.
- Allows initialization of an object with specific, user-defined values at the time of creation.
- Highly useful for passing dynamic data to an object when it's instantiated.
- Note: Once you define a parameterized constructor, the compiler will NO LONGER provide an implicit default constructor.

This is perhaps the most commonly used constructor. When you create an object representing a bank account, you usually want to set the account holder name and initial balance immediately, not later. A parameterized constructor allows this. Also, pay attention to the last point: if you write a parameterized constructor, you lose the free default one provided by the compiler.

Parameterized Constructor: Code Example

```
class Point {
private:
int x, y;
public:
// Parameterized Constructor
Point(int xVal, int yVal) {
x = xVal;
y = yVal;
}
};
// Usage:
Point p1(10, 20); // Implicit call
Point p2 = Point(30, 40); // Explicit call
```


Parameterized Constructor: Code Example
class Point {
private:
int x, y;
public:
// Parameterized Constructor
Point(int xVal, int yVal) {
x = xVal;
y = yVal;
}
};
// Usage:
Point p1(10, 20); // Implicit call
Point p2 = Point(30, 40); // Explicit call

Here the Point class has a constructor taking two integers. We can call it in two ways. The first is implicit, 'Point p1(10, 20)', which is the most common and concise. The second is explicit assignment syntax. Both achieve the exact same thing: instantiating a Point object and passing values to 'x' and 'y'.

3. The Copy Constructor

- A constructor that initializes an object using an existing object of the same class.
- Syntax takes a reference to an object of the same class: `ClassName(const ClassName &obj)`
- Used in scenarios like:
- - Returning an object from a function by value.
- - Passing an object to a function by value.
- - Initializing one object with another.

3. The Copy Constructor
- A constructor that initializes an object using an existing object of the same class.
- Syntax takes a reference to an object of the same class: ClassName(const ClassName &obj)
- Used in scenarios like:
- - Returning an object from a function by value.
- - Passing an object to a function by value.
- - Initializing one object with another.

The Copy Constructor is unique. It takes a reference to another object of its own kind. Why a reference? If we passed it by value, passing the argument would itself call the copy constructor, resulting in an infinite loop! Hence, it is always a reference, and usually 'const' to prevent accidental modification of the source object.

Copy Constructor: Code Example

```
class Box {
int length;
public:
Box(int l) { length = l; } // Parameterized
// Copy Constructor
Box(const Box &b) {
length = b.length;
cout ¡¡ "Copy Constructor called!" ¡¡ endl;
}
};
// Usage:
Box box1(10);
Box box2 = box1; // Calls Copy Constructor
Box box3(box1); // Calls Copy Constructor
```


Copy Constructor: Code Example
class Box {
int length;
public:
Box(int l) { length = l; } // Parameterized
// Copy Constructor
Box(const Box &b) {
length = b.length;
cout ¡¡ "Copy Constructor called!" ¡¡ endl;
}
};
// Usage:
Box box1(10);
Box box2 = box1; // Calls Copy Constructor
Box box3(box1); // Calls Copy Constructor

In this example, box1 is created using the parameterized constructor. When we create box2 and box3, we are using box1 to initialize them. The compiler sees we are creating a new Box from an existing Box, so it routes execution to our Copy Constructor. Both 'Box box2 = box1' and 'Box box3(box1)' trigger this.

Default Copy Constructor & Shallow Copy

- Like the default constructor, if you don't define a copy constructor, the compiler provides one.
- The compiler's copy constructor performs a 'Shallow Copy'.
- Shallow Copy copies the exact values of variables, including memory addresses (pointers).
- This becomes dangerous if your class allocates dynamic memory (using 'new'), as both objects will point to the same memory location.

Default Copy Constructor & Shallow Copy
- Like the default constructor, if you don't define a copy constructor, the compiler provides one.
- The compiler's copy constructor performs a 'Shallow Copy'.
- Shallow Copy copies the exact values of variables, including memory addresses (pointers).
- This becomes dangerous if your class allocates dynamic memory (using 'new'), as both objects will point to the same memory location.

For classes holding basic data types like ints and floats, the compiler-provided copy constructor works perfectly. But the moment your class contains pointers to dynamically allocated memory, a shallow copy means two objects share the same memory pointer. If one deletes that memory, the other object is left with a dangling pointer.

Deep Copy (Why we need custom Copy Constructors)

- To avoid shallow copy issues, we write our own Custom Copy Constructor.
- This allows us to perform a 'Deep Copy'.
- Deep Copy means allocating brand new memory for the new object and then copying the actual data over, rather than just the pointer.
- Rule of Three: If a class needs a custom copy constructor, it likely needs a custom destructor and a custom assignment operator too.

When we write our own copy constructor, we can intercept the copying process. Instead of blindly copying a pointer, we allocate new memory using 'new', and then copy the values from the source's memory into the new object's memory. This is called a deep copy and ensures both objects are completely independent.

Multiple Constructors in a Class

- A class can have more than one constructor. This is known as Constructor Overloading.
- It follows the exact same rules as Function Overloading.
- Constructors must differ in their 'Signature':
- - Number of parameters.
- - Type of parameters.
- - Sequence of parameters.
- The compiler determines which constructor to call based on the arguments provided at object creation.

Multiple Constructors in a Class
- A class can have more than one constructor. This is known as Constructor Overloading.
- It follows the exact same rules as Function Overloading.
- Constructors must differ in their 'Signature':
- - Number of parameters.
- - Type of parameters.
- - Sequence of parameters.
- The compiler determines which constructor to call based on the arguments provided at object creation.

Because a constructor is just a special function, it can be overloaded. This gives the users of your class flexibility. They can create an object with no info (default), partial info, or full info (parameterized), simply depending on how they instantiate the object. The compiler acts as a traffic cop, routing the call to the correct constructor based on the arguments.

Constructor Overloading: Code Example

- class Rectangle {
- int length, width;
- public:
- Rectangle() { length = 0; width = 0; } // Default
- Rectangle(int l) { length = l; width = l; } // Square
- Rectangle(int l, int w) { length = l; width = w; } // Rectangle
- };
- // Usage:
- Rectangle r1; // Calls Default: 0x0
- Rectangle r2(5); // Calls 1-param: 5x5
- Rectangle r3(5, 10); // Calls 2-param: 5x10

Constructor Overloading: Code Example
- class Rectangle {
- int length, width;
- public:
- Rectangle() { length = 0; width = 0; } // Default
- Rectangle(int l) { length = l; width = l; } // Square
- Rectangle(int l, int w) { length = l; width = w; } // Rectangle
- };
- // Usage:
- Rectangle r1; // Calls Default: 0x0
- Rectangle r2(5); // Calls 1-param: 5x5
- Rectangle r3(5, 10); // Calls 2-param: 5x10

Here we have a Rectangle class with three constructors. If I pass no arguments, I get a 0x0 rectangle. If I pass one argument, the class assumes I want a square and sets both sides to that value. If I pass two arguments, it sets distinct length and width. This is constructor overloading in action, providing a highly intuitive API for object creation.

Best Practices: Initialization Lists

- Instead of assigning values inside the constructor body, use Member Initializer Lists.
- Syntax: `ClassName(int a) : memberVar(a) {}`
- Benefits:
- - More efficient (initializes directly instead of initializing then assigning).
- - Required for initializing `const` members and reference members.
- - Required when a base class doesn't have a default constructor.

Best Practices: Initialization Lists
- Instead of assigning values inside the constructor body, use Member Initializer Lists.
- Syntax: ClassName(int a) : memberVar(a) {}
- Benefits:
- - More efficient (initializes directly instead of initializing then assigning).
- - Required for initializing const members and reference members.
- - Required when a base class doesn't have a default constructor.

Before we wrap up, a critical best practice in C++. While assigning inside the constructor body works, it's technically a two-step process: default initialization, then assignment. Initializer lists construct the variable with the right value immediately. For const variables and references, initializer lists are mandatory because they cannot be assigned to after creation.

Summary & Recap

- Constructors are special functions called automatically upon object creation.
- They have no return type and share the class name.
- Default constructors take no arguments; Parameterized take arguments.
- Copy constructors create an object from an existing one, usually requiring a Deep Copy for dynamic memory.
- Constructor overloading allows multiple initialization strategies within a single class.

Summary & Recap
- Constructors are special functions called automatically upon object creation.
- They have no return type and share the class name.
- Default constructors take no arguments; Parameterized take arguments.
- Copy constructors create an object from an existing one, usually requiring a Deep Copy for dynamic memory.
- Constructor overloading allows multiple initialization strategies within a single class.

To summarize, constructors are the birth events of your objects. They ensure your data is safe and valid from millisecond one. We covered default, parameterized, and copy constructors, and how overloading them gives developers flexibility. Understanding these concepts is vital for robust C++ programming.

Q&A and Next Steps

- Any questions on Default, Parameterized, or Copy constructors?
- Next Lecture Preview: Topic 2.4 - Destructors.
- We will learn what happens when an object's lifecycle ends.
- We will discuss memory deallocation and cleanup.

Q&A and Next Steps
- Any questions on Default, Parameterized, or Copy constructors?
- Next Lecture Preview: Topic 2.4 - Destructors.
- We will learn what happens when an object's lifecycle ends.
- We will discuss memory deallocation and cleanup.

Does anyone have questions about how the compiler chooses a constructor, or how deep copying works? If not, please review the code examples in your textbook. Next time, we will look at the opposite of constructors: Destructors. We will learn how to clean up memory when our objects are no longer needed. Thank you for your attention!