# 2.3 Constructors

## Unit 2: Classes and Objects

Object Oriented Programming with C++





# Lecture 16: Constructors in C++

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





Lecture 16: Constructors in C++
- Unit 2: Classes and Objects
- Topic 2.3: Constructors
- Course: Object Oriented Programming with C++

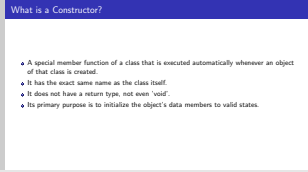
Welcome back everyone. Today we are diving into one of the most critical concepts in C++ object-oriented programming: Constructors. In our last class, we learned how to create a class and instantiate an object. Today, we will learn how to properly initialize those objects the moment they are created.

# What is a Constructor?

- A special member function of a class that is executed automatically whenever an object of that class is created.
- It has the exact same name as the class itself.
- It does not have a return type, not even 'void'.
- Its primary purpose is to initialize the object's data members to valid states.








Think of a constructor as a setup routine. Just as a new building needs its foundation poured before it can be used, an object needs its data initialized before it is safe to interact with. Because it is uniquely tied to the creation process, C++ enforces that the constructor shares the name of the class and omits the return type entirely.

# Key Characteristics of Constructors

- Automatic Invocation: You never call a constructor explicitly like a normal function (e.g., obj.Constructor()); the compiler calls it when memory is allocated.
- Placement: Usually declared in the 'public' section of a class. (Private constructors have special uses, like in the Singleton pattern).
- Overloading: A class can have more than one constructor as long as their parameter lists differ.
- Inheritance: Constructors cannot be inherited, nor can they be virtual.





Key Characteristics of Constructors
- Automatic Invocation: You never call a constructor explicitly like a normal function (e.g., obj.Constructor()); the compiler calls it when memory is allocated.
- Placement: Usually declared in the 'public' section of a class. (Private constructors have special uses, like in the Singleton pattern).
- Overloading: A class can have more than one constructor as long as their parameter lists differ.
- Inheritance: Constructors cannot be inherited, nor can they be virtual.

It's important to remember that constructors are invoked implicitly. When you write 'ClassName obj;', the compiler handles the memory allocation and immediately fires the constructor. Generally, you want them public so external code can create objects. Later in advanced design patterns, you might see private constructors, but for now, keep them public.

# Why Do We Need Constructors?

- The Problem with Uninitialized Data: Local variables in C++ are not initialized by default; they contain garbage values. The same applies to object data members.
- The 'init()' Function Anti-Pattern: Relying on a manual 'initialize()' function is error-prone. Programmers might forget to call it.
- The Solution: Constructors guarantee that an object is never in an invalid or 'garbage' state because initialization is tied directly to creation.





Why Do We Need Constructors?
- The Problem with Uninitialized Data: Local variables in C++ are not initialized by default; they contain garbage values. The same applies to object data members.
- The 'init()' Function Anti-Pattern: Relying on a manual 'initialize()' function is error-prone. Programmers might forget to call it.
- The Solution: Constructors guarantee that an object is never in an invalid or 'garbage' state because initialization is tied directly to creation.

Imagine creating an 'Account' object but forgetting to set the initial balance to zero. If you rely on a manual init() method, forgetting it could lead to severe bugs where the balance is a random garbage value from memory. Constructors solve this by making initialization an unavoidable part of creation.

# Types of Constructors

- 1. Default Constructor: Takes no arguments.
- 2. Parameterized Constructor: Takes one or more arguments.
- 3. Copy Constructor: Takes a reference to another object of the same class.





Types of Constructors
1. Default Constructor: Takes no arguments.
2. Parameterized Constructor: Takes one or more arguments.
3. Copy Constructor: Takes a reference to another object of the same class.

In C++, we primarily categorize constructors into three types based on their signatures. We will spend the rest of the lecture looking at each of these in detail, starting with the Default Constructor.

# 1. The Default Constructor

- A constructor that either has no parameters, or all of its parameters have default arguments.
- If you do not define ANY constructor for a class, the C++ compiler automatically generates a hidden, default constructor.
- The compiler-generated default constructor allocates memory but leaves basic data types (int, float, etc.) uninitialized.





1. The Default Constructor
- A constructor that either has no parameters, or all of its parameters have default arguments.
- If you do not define ANY constructor for a class, the C++ compiler automatically generates a hidden, default constructor.
- The compiler-generated default constructor allocates memory but leaves basic data types (int, float, etc.) uninitialized.

The default constructor is your baseline. If you write 'Point p;', you are invoking the default constructor. A critical rule to remember: the compiler only gives you a free default constructor if you write NO constructors at all. If you write a parameterized constructor, you lose the free default one.

# Default Constructor: Code Example

```
class Point {
private:
int x, y;
public:
// Default Constructor
Point() {
x = 0;
y = 0;
cout ¡¡ "Default Constructor called" ¡¡ endl;
}
};

int main() {
Point p1; // Invokes the default constructor
return 0;
}
```





Default Constructor: Code Example
class Point {
private:
int x, y;
public:
// Default Constructor
Point() {
x = 0;
y = 0;
cout ¡¡ "Default Constructor called" ¡¡ endl;
}
};
int main() {
Point p1; // Invokes the default constructor
return 0;
}

Here we explicitly define a default constructor. When 'Point p1;' is executed in main, the console will print 'Default Constructor called', and the object p1 will safely have its coordinates set to (0, 0) instead of garbage memory.

# The 'Missing Default Constructor' Error

- If you define a Parameterized Constructor, the compiler assumes you want strict control over initialization.
- It will NOT generate a Default Constructor automatically.
- Attempting to create an object without parameters will result in a compilation error unless you explicitly add a Default Constructor.





The 'Missing Default Constructor' Error
- If you define a Parameterized Constructor, the compiler assumes you want strict control over initialization.
- It will NOT generate a Default Constructor automatically.
- Attempting to create an object without parameters will result in a compilation error unless you explicitly add a Default Constructor.

This is a very common beginner mistake. You write a constructor that takes an ID and Name. Then somewhere else, you try to create an array of these objects or just a blank object, and the compiler throws an error. Why? Because by defining a specific constructor, you revoked the compiler's permission to make a default one.

# 2. The Parameterized Constructor

- A constructor that takes one or more arguments.
- Allows you to pass specific, dynamic values to initialize an object's state at the exact moment of creation.
- Provides flexibility to create objects with different initial data.





2. The Parameterized Constructor
- A constructor that takes one or more arguments.
- Allows you to pass specific, dynamic values to initialize an object's state at the exact moment of creation.
- Provides flexibility to create objects with different initial data.

While default constructors are great for setting a baseline (like a 0 balance), parameterized constructors let us create objects that are immediately useful. If I create a Student object, I want to pass their name and roll number right away, rather than creating a blank student and setting them later.

# Parameterized Constructor: Code Example

```
class Point {
private:
int x, y;
public:
// Parameterized Constructor
Point(int x_val, int y_val) {
x = x_val;
y = y_val;
}
};

int main() {
Point p1(10, 20); // Implicit call
Point p2 = Point(30, 40); // Explicit call
return 0;
}
```





Parameterized Constructor: Code Example
class Point {
private:
int x, y;
public:
// Parameterized Constructor
Point(int x_val, int y_val) {
x = x_val;
y = y_val;
}
};
int main() {
Point p1(10, 20); // Implicit call
Point p2 = Point(30, 40); // Explicit call
return 0;
}

Notice how we pass the values (10, 20) inside parentheses right after the object name. This matches the signature of our parameterized constructor. We can do this implicitly, or explicitly using the assignment operator as shown on the second line of main.

# Constructor Initialization Lists (Briefly)

- An alternative, more efficient syntax for initializing member variables.
- Syntax: ConstructorName(args) : member1(arg1), member2(arg2) { ... }
- Required for initializing 'const' members and reference (&) members.
- Often preferred by professional C++ developers over assigning inside the constructor body.





Constructor Initialization Lists (Briefly)
- An alternative, more efficient syntax for initializing member variables.
- Syntax: ConstructorName(args) : member1(arg1), member2(arg2) { ... }
- Required for initializing 'const' members and reference (&) members.
- Often preferred by professional C++ developers over assigning inside the constructor body.

Though assigning values inside the curly braces works, C++ provides Initialization Lists. They initialize the variables before the constructor body even runs, which is slightly more efficient. More importantly, if you have a 'const' class member, you MUST use this syntax, because you cannot assign a value to a const variable after it has been created.

# 3. The Copy Constructor

- A constructor used to create a new object as an exact replica of an existing object.
- Signature: ClassName(const ClassName &old_obj);
- It takes a reference to an object of the same class.
- If not explicitly defined, the compiler provides a default Copy Constructor that performs a 'shallow copy'.





3. The Copy Constructor
- A constructor used to create a new object as an exact replica of an existing object.
- Signature: ClassName(const ClassName &old_obj);
- It takes a reference to an object of the same class.
- If not explicitly defined, the compiler provides a default Copy Constructor that performs a 'shallow copy'.

The Copy Constructor is invoked when you want to clone an object. It's crucial to pass the argument by reference (using the ampersand). If you tried to pass it by value, it would trigger an infinite loop of calling the copy constructor to make the copy for the parameter! The 'const' ensures we don't accidentally modify the object we are copying from.

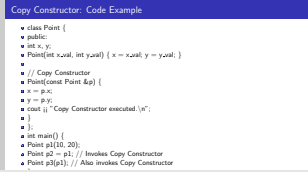
# Copy Constructor: Code Example

```
class Point {
public:
int x, y;
Point(int x_val, int y_val) { x = x_val; y = y_val; }

// Copy Constructor
Point(const Point &p) {
x = p.x;
y = p.y;
cout ¡¡ "Copy Constructor executed.\n";
}
};
int main() {
Point p1(10, 20);
Point p2 = p1; // Invokes Copy Constructor
Point p3(p1); // Also invokes Copy Constructor
```








In main, p1 is created using the parameterized constructor. When we create p2 and initialize it with p1, the copy constructor is triggered. The same happens for p3. Both p2 and p3 are distinct objects in memory, but their x and y values are identical to p1's.

# When is the Copy Constructor called?

- 1. When an object is instantiated and initialized with another object of the same class.
- 2. When an object is passed by value to a function (e.g., void display(Point p)).
- 3. When an object is returned by value from a function.





When is the Copy Constructor called?
- 1. When an object is instantiated and initialized with another object of the same class.
- 2. When an object is passed by value to a function (e.g., void display(Point p)).
- 3. When an object is returned by value from a function.

It's a common misconception that the copy constructor is only called when we explicitly write 'Object A = B'. Behind the scenes, C++ uses it whenever it needs a temporary copy, like when passing arguments by value to functions. This is why passing large objects by reference is generally preferred for performance.

# Shallow Copy vs. Deep Copy (Context)

- Default Copy Constructor: Performs a 'Shallow Copy'. It copies member values exactly as they are.
- The Danger: If the class contains pointers to dynamically allocated memory, a shallow copy copies the pointer address, not the data. Both objects will point to the same memory!
- The Fix: A custom Copy Constructor allows you to perform a 'Deep Copy', allocating new memory for the copied object.





Shallow Copy vs. Deep Copy (Context)
- Default Copy Constructor: Performs a 'Shallow Copy'. It copies member values exactly as they are.
- The Danger: If the class contains pointers to dynamically allocated memory, a shallow copy copies the pointer address, not the data. Both objects will point to the same memory!
- The Fix: A custom Copy Constructor allows you to perform a 'Deep Copy', allocating new memory for the copied object.

While the compiler's default copy constructor is usually fine, it fails dangerously if your class uses pointers (like 'new int'). If two objects share the same memory pointer, deleting one object will delete the memory the other object is still relying on. Writing your own copy constructor is how you prevent this by allocating fresh memory.

# Multiple Constructors (Constructor Overloading)

- Just like normal functions, constructors can be overloaded.
- A class can have a default, multiple parameterized, and a copy constructor all at the same time.
- The compiler determines which constructor to call based on the number and data type of the arguments passed during object creation.





Multiple Constructors (Constructor Overloading)
- Just like normal functions, constructors can be overloaded.
- A class can have a default, multiple parameterized, and a copy constructor all at the same time.
- The compiler determines which constructor to call based on the number and data type of the arguments passed during object creation.

Constructor overloading gives your class versatility. Think of a 'String' class. You might want to create an empty string (default constructor), a string from a char array (parameterized), or a string that's a copy of another string (copy constructor). Overloading makes all of this possible.

# Constructor Overloading: Code Example

```
class Box {
private:
int length;
public:
Box() { length = 0; } // Default
Box(int l) { length = l; } // Parameterized
Box(const Box &b) { length = b.length; } // Copy
};

int main() {
Box b1; // Calls Default
Box b2(10); // Calls Parameterized
Box b3 = b2; // Calls Copy
return 0;
}
```





Constructor Overloading: Code Example
class Box {
private:
int length;
public:
Box() { length = 0; } // Default
Box(int l) { length = l; } // Parameterized
Box(const Box &b) { length = b.length; } // Copy
};

int main() {
Box b1; // Calls Default
Box b2(10); // Calls Parameterized
Box b3 = b2; // Calls Copy
return 0;
}

This slide brings it all together. Here is a single class 'Box' implementing all three types of constructors we discussed today. In main(), you can see how different instantiation syntax routes to different constructors seamlessly.

# Summary & Best Practices

- Constructors are special functions with the same name as the class, used for initialization.
- They do not have return types and are invoked automatically.
- Types: Default, Parameterized, and Copy Constructors.
- Best Practice 1: If you define a parameterized constructor, always define a default constructor explicitly if you need to create blank objects.
- Best Practice 2: If your class uses dynamic memory (pointers), you MUST write your own Copy Constructor.





Summary & Best Practices
- Constructors are special functions with the same name as the class, used for initialization.
- They do not have return types and are invoked automatically.
- Types: Default, Parameterized, and Copy Constructors.
- Best Practice 1: If you define a parameterized constructor, always define a default constructor explicitly if you need to create blank objects.
- Best Practice 2: If your class uses dynamic memory (pointers), you MUST write your own Copy Constructor.

To wrap up: Constructors are your mechanism for safe object creation. Remember the rule of thumb regarding default constructors disappearing, and be cautious when dealing with pointers to avoid shallow copy bugs.