# Object-Oriented Programming (OOP) Fundamentals

Welcome to Week 3 of your OOP journey! This lecture dives into the core mechanics of classes, objects, and their interactions, focusing on practical implementation details crucial for building robust software. We will demystify how class components work together and how objects communicate.

### Class Structure

Understanding the blueprint for object creation.

### Function Definition

Implementing member functions both inside and outside the class definition.

### Object Interaction

Passing objects as arguments to functions for inter-object communication.

### Constructors

Automating the initialization of objects upon creation.

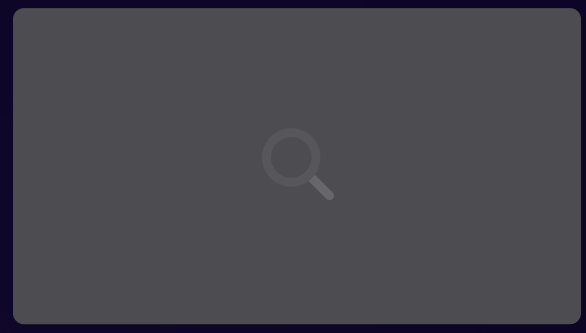
By the end of this session, you will have a solid foundation for defining, initializing, and managing objects in a practical programming environment.

# Chapter 1: Examples of Class – The Blueprint

A **class** is the fundamental building block of OOP. Think of it as a blueprint or a template for creating objects. It defines the structure and behavior that all objects of that class will possess.

## Defining a Class: Structure

A class bundles data (variables, known as **attributes** or **data members**) and functions (known as **methods** or **member functions**) that operate on that data. This concept is called **encapsulation**.



> In simple terms, if a blueprint for a 'Car' is the class, it defines properties like color and speed (data members) and actions like accelerate and brake (member functions).

### Code Example: Simple 'Dog' Class

```
class Dog {
public:
  // Data Members (Attributes)
  std::string name;
  int age;

  // Member Function (Method)
  void bark() {
    std::cout << name << " says Woof!" << std::endl;
  }
};
```

In this example, `Dog` is the class. `name` and `age` are the attributes, and `bark()` is the behavior. When you create an object from this class, it gets its own copies of these attributes and can perform the `bark()` action.

# Member Functions Defined Outside the Class

While simple member functions can be defined directly inside the class definition, larger or more complex functions are often defined **outside** the class body for better readability and organization, especially in larger projects.

## Using the Scope Resolution Operator (::)

When defining a member function outside its class, we must use the `Scope Resolution Operator (::)` to tell the compiler which class the function belongs to. This links the function implementation back to its class blueprint.

### → Declaration Inside

The function prototype (header) is declared within the class definition (usually in a header file). This tells the compiler the function exists.

### → Definition Outside

The actual implementation (function body) is written outside the class body (usually in a source file, like a `.cpp` file). This keeps the class definition clean.

### → Syntax Linkage

The syntax looks like: `ReturnType ClassName::FunctionName(parameters) { // body }`

### Code Example: External Function Definition

```
// Class Definition (e.g., in a header file)
class Rectangle {
public:
 int length;
 int width;
 void displayArea(); // Function declaration only
};

// Function Definition (e.g., in a source file)
void Rectangle::displayArea() {
 int area = length * width;
 std::cout << "The area is: " << area << std::endl;
}
```

# Objects as Function Arguments: Enabling Communication

One of the key features of OOP is how objects interact. Objects can be passed to functions just like standard data types (integers, strings, etc.). This mechanism allows us to write functions that compare, modify, or display the data contained within different objects.

## Why Pass Objects?

- Comparison: A function might compare two 'Student' objects based on their grades.
- Calculation: A function might take a 'Vector' object and calculate its magnitude.
- Modification: A function might take a 'Balance' object and update its value after a transaction.

Passing objects ensures that the function operates on meaningful, structured data defined by the class, rather than just raw variables.

### Passing Methods

Objects can be passed to functions in three primary ways, similar to fundamental types:

1. **Pass by Value:** A copy of the object is created and passed to the function. Changes inside the function do not affect the original object.
2. **Pass by Reference:** The function receives an alias (reference) to the original object. Changes inside the function directly affect the original object.
3. **Pass by Pointer:** The memory address of the object is passed. This is commonly used for efficiency or when dynamic memory allocation is involved.

# Example: Objects as Arguments - Comparing Data

Let's look at a concrete example where we pass two objects of the same class to a function to compare their internal data. We will use a `Time` class and a function to find the greater time duration.

### Define the Class

The `Time` class holds two attributes: `hours` and `minutes`.

### Define the Function

A standalone function, `compareTime(Time t1, Time t2)`, takes two `Time` objects as arguments.

### Logic

The function converts both times to a single unit (total minutes) and compares them to determine which time is longer.

## Code Example: Comparing Objects

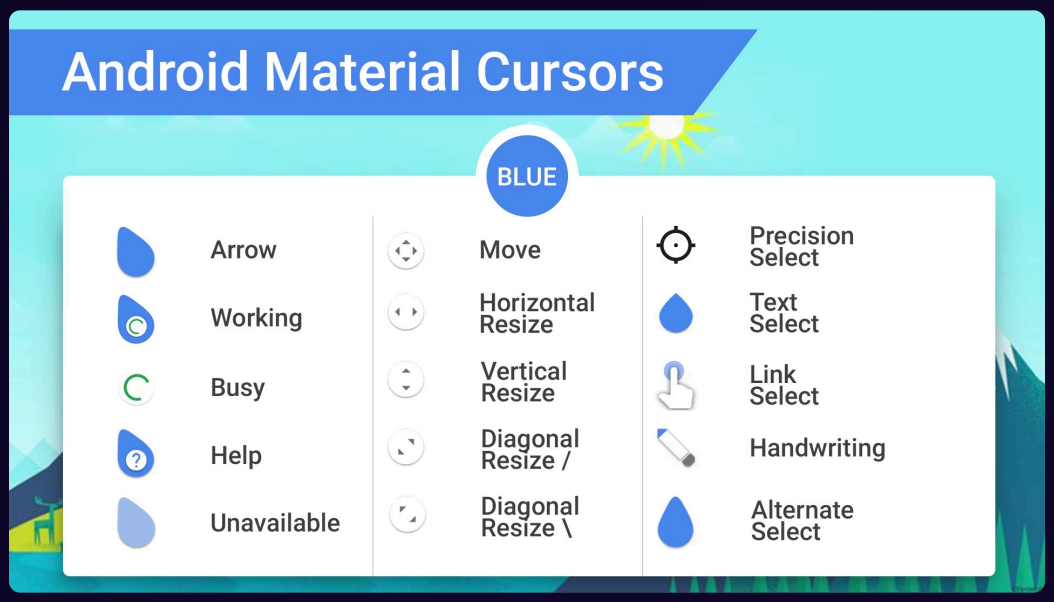
```
class Time {
public:
 int hours;
 int minutes;
};

// Function taking two Time objects as arguments (Pass by Value)
void compareTime(Time t1, Time t2) {
 int totalMin1 = t1.hours * 60 + t1.minutes;
 int totalMin2 = t2.hours * 60 + t2.minutes;

 if (totalMin1 > totalMin2) {
 std::cout << "Time 1 is longer." << std::endl;
 } else if (totalMin2 > totalMin1) {
 std::cout << "Time 2 is longer." << std::endl;
 } else {
 std::cout << "Both times are equal." << std::endl;
 }
}

// In main:
// Time meetingStart, meetingEnd;
// compareTime(meetingStart, meetingEnd);
```

# The Object as an Argument: Passing by Reference

When passing large objects, copying them (pass by value) can be inefficient. Passing by reference (&) allows the function to work directly with the original object, saving memory and time. This is especially critical when the function needs to modify the object's state.

## When to Use Pass by Reference

- Efficiency: For large objects, to avoid the overhead of copying data.
- Modification: When you explicitly need the function to change the data members of the original object.
- Const Reference: If you want efficiency but prevent modification, pass by constant reference (e.g., `const Time& t`). This is the best practice for passing large objects if they should not be changed.




### Code Example: Modifying an Object

This function updates the coordinates of a `Point` object.

```
class Point {
public:
 int x, y;
};

// Function modifies the object 'p' directly
void movePoint(Point& p, int deltaX, int deltaY) {
 p.x += deltaX;
 p.y += deltaY;
 std::cout << "Point moved to (" << p.x << ", " << p.y << ")" << std::endl;
}

// In main:
// Point current; current.x = 0; current.y = 0;
// movePoint(current, 5, 10); // current is now (5, 10)
```

# Defining the Constructor: Object Birth

A **Constructor** is a special member function that is automatically called when an object of a class is created. Its main purpose is to initialize the object's data members.

## Key Characteristics of Constructors

- Name: A constructor always has the same name as the class.
- No Return Type: It has no return type, not even `void`.
- Automatic Call: It cannot be called explicitly by the programmer; it is invoked automatically by the system when the object is instantiated.
- Initialization: It prepares the object for use by assigning initial values to its attributes, preventing uninitialized data errors.

## Types of Constructors

### Default Constructor

Takes no arguments. Used when an object is declared without initial values (e.g., `MyClass obj;`).

### Parameterized Constructor

Takes one or more arguments, allowing the user to initialize data members at the time of creation (e.g., `MyClass obj(5, "data");`).

### Copy Constructor

Takes a reference to an object of the same class as an argument. Used to create a new object as a copy of an existing object.

# Implementing the Parameterized Constructor

The parameterized constructor is vital for ensuring that new objects start in a valid and meaningful state. It requires input arguments that correspond to the initial values of the data members.

## Code Example: 'Book' Class Constructor

In the `Book` class, we want every book object to be initialized with a title and an ISBN upon creation.

```
class Book {
private:
  std::string title;
  long isbn;

public:
  // Parameterized Constructor
  Book(std::string t, long i) {
    title = t; // Initialize title
    isbn = i;  // Initialize isbn
    std::cout << "Book object created: " << title << std::endl;
  }

  void displayDetails() {
    std::cout << "Title: " << title << ", ISBN: " << isbn << std::endl;
  }
};

// Object Creation (Instantiation) in main
// Book b1("The OOP Guide", 9781234567890);
// Book b2("Coding Fundamentals", 9780001112223);
```

## Understanding Initialization

When you write `Book b1("The OOP Guide", 9781234567890);`, the compiler automatically matches the arguments to the parameterized constructor, executing the initialization code inside.

```
import photomover.cli.web.StartWebOptions

fun main(arguments: Array<String>) {
  val commands = mapOf(
      "organize" to wrapOperation(::organize, OrganizeOptions()),
      "upload" to wrapOperation(::upload, UploadOptions()),
      "startWeb" to wrapOperation(::start, StartWebOptions())
  val args = if (arguments.size == 0) {
    array("startWeb")
  } else {
    arguments
  }
  val command = commands[args[0]]
  if (command == null) {
    println("Command '${args[0]}' is not supported.")
    println("Please specify one of following: ${commands.keySet()}")
  } else {
    println("Starting ${args[0]}")
    command(args.drop(1))
  }
}

fun wrapOperation<T>(
    operation: (options: T) -> Unit, defaultOptions: T): (args: List<String>) -> Unit {
  return { args ->
    val parser = CmdLineParser(defaultOptions)
    try {
      parser.parseArgument(args)
    } catch (ex: CmdLineException) {
      println(ex.getMessage())
      println(parser.printUsage(System.err))
    }
    println("args: $defaultOptions")
    operation(defaultOptions)
  }
}
```

# Best Practice: Member Initializer Lists

While direct assignment inside the constructor body works, a more efficient and preferred method in C++ is to use an **Initializer List** to initialize data members.

## Efficiency and Order

The initializer list directly initializes the members, whereas assignment inside the body involves two steps: first, the member is default-initialized (if applicable), and then it is assigned a value. Using a list is usually faster and required for initializing `const` members or reference members.

### Syntax with Initializer List

```
class Product {
private:
  const int id;
  double price;

public:
  // Initializer List (best practice)
  Product(int i, double p) : id(i), price(p) {
    // Constructor body can be empty or used for other setup
    std::cout << "Product " << id << " initialized." << std::endl;
  }
};
```

### Syntax without Initializer List (Less Efficient)

```
class Product {
private:
  int id;
  double price;

public:
  // Direct Assignment
  Product(int i, double p) {
    id = i;    // Assignment happens here
    price = p; // Assignment happens here
  }
};
```

For professional and high-performance programming, mastering the use of initializer lists is essential for proper object construction.

# Key Takeaways and Next Steps

We have covered crucial practical aspects of class and object mechanics this week. These techniques are fundamental to writing well-structured and maintainable OOP code.

## Class Definition

The class acts as a mold, encapsulating attributes and methods. Remember the importance of separating interface (class definition) and implementation (function body).

## Object Communication

Objects are passed as function arguments to enable interaction. Choose between pass-by-value, reference, or constant reference based on efficiency and modification needs.

## Constructor Mastery

Constructors automate initialization, ensuring objects start in a valid state. Parameterized constructors allow dynamic initial values, while initializer lists represent the most efficient way to achieve this.

## What's Next?

Next week, we will expand on object relationships by introducing concepts of **composition** and the special properties of **Destructors** (the functions that handle object cleanup).

Review Practice Problems

Preview Next Topic