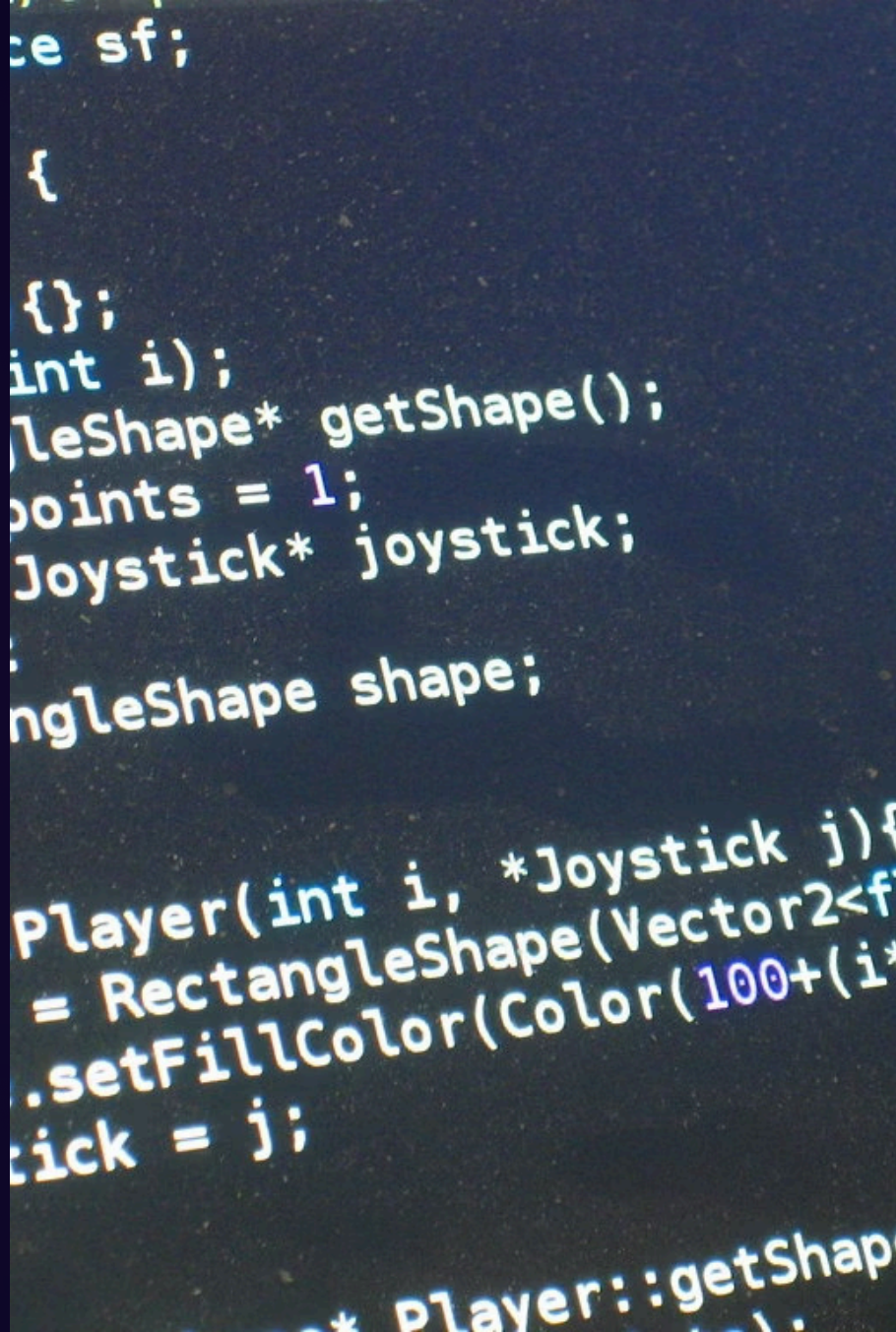
Introduction to Object-Oriented Programming in C++

Welcome to the world of Object-Oriented Programming (OOP)! This is Week 1, where we'll explore the fundamental concepts that make C++ one of the most powerful programming languages. OOP is a way of thinking about and organizing code that mirrors the real world—breaking down complex problems into smaller, manageable objects that work together.

Throughout this lecture, you'll learn how to structure your code using primitives, user-defined data types, structures, and enumerations. These building blocks will give you the foundation to write cleaner, more organized, and more maintainable code.



What is Object-Oriented Programming (OOP)?

Object-Oriented Programming is a programming paradigm—a way of thinking about and writing code—that organizes software design around objects and data rather than functions and logic. Think of it like building with LEGO blocks: instead of creating one massive, complicated structure, you build smaller pieces (objects) that each have their own properties and behaviors, then snap them together to create something amazing.

Key Principles of OOP:

Encapsulation

Bundle data and functions together, hiding internal details from the outside world

Abstraction

Show only what's necessary; hide complexity behind a simple interface

Inheritance

Create new classes from existing ones, reusing code and building hierarchies

Polymorphism

Allow objects to take many forms and respond differently to the same message

C++ makes OOP easy by providing built-in support for these concepts through classes, access specifiers, and inheritance mechanisms. Whether you're building a small utility or a massive application, OOP helps you write code that's organized, reusable, and easier to maintain.

Primitive Data Types in C++

Before we can build complex structures, we need to understand the basic building blocks: primitive data types. These are the fundamental data types provided by C++ that store simple values. Think of them as the atomic elements of programming—you can't break them down further, but you can combine them to create something more complex.

Common Primitive Data Types:

int

Stores whole numbers (integers). Uses 4 bytes. Range: -2,147,483,648 to 2,147,483,647

float

Stores decimal numbers (single precision). Uses 4 bytes. Example: 3.14, 2.5

double

Stores decimal numbers (double precision). Uses 8 bytes. More accurate than float

char

Stores single characters. Uses 1 byte. Example: 'A', 'z', '5'

bool

Stores true or false values. Uses 1 byte. Essential for conditional logic

Code Example:

```
int age = 20;
float height = 5.8;
double pi = 3.14159265359;
char grade = 'A';
bool isPassing = true;
```

User-Defined Data Types

While primitive data types are useful, they're limited. Real-world programming requires more complex data structures. User-defined data types allow you to create your own data types that combine multiple primitives or other data types into a single, meaningful unit. This is where OOP truly begins—you're defining the blueprint for objects that represent real-world entities.

Types of User-Defined Data Types:

→ Structures (struct)

Group related variables and functions together. All members are public by default

→ Classes (class)

Similar to structures but more powerful. Members are private by default, allowing better encapsulation

→ Unions (union)

Share memory space—only one member can hold a value at a time

→ Enumerations (enum)

Define a set of named integer constants—useful for representing fixed sets of values

User-defined data types are the foundation of OOP. They let you model real-world concepts like "Student," "Car," or "BankAccount" directly in your code, making programs more intuitive and maintainable.

Introduction to Structures

A **structure** is a user-defined data type that groups multiple variables (called members or fields) under a single name. Imagine a student record—it contains a name, ID, GPA, and major. Without structures, you'd need separate variables for each. With structures, you bundle everything together logically. Structures are perfect for representing real-world entities that have multiple related properties.

Why Use Structures?

- **Organization:** Keep related data together, making code clearer and easier to understand
- **Reusability:** Create a template you can use multiple times throughout your program
- **Maintainability:** Modify all related data in one place rather than scattered throughout code
- **Simplicity:** Pass all related data as a single parameter instead of many separate ones

Structure Syntax (Basic):

```
struct Student {
    int id;
    string name;
    float gpa;
    string major;
};
```

Notice the semicolon at the end—this is required! The members (id, name, gpa, major) define what data each Student object will hold. By default, all members in a struct are **public**, meaning they can be accessed from anywhere in your code.

Declaring Structure Variables

Once you've defined a structure, you need to create actual instances (variables) of it. Think of the structure definition as a blueprint—declaring a variable is like building an actual house from that blueprint. Each variable is a separate object with its own copy of the members.

How to Declare Structure Variables:

After defining your structure, you can declare variables in several ways:

```
// Method 1: Declare after structure definition
struct Book {
    string title;
    string author;
    int pages;
};

Book myBook;  // Single variable

// Method 2: Multiple variables
Book book1, book2, book3;

// Method 3: Initialize while declaring
Book favorite = {"1984", "George Orwell", 328};
```

Memory Allocation:

When you declare a structure variable, the computer allocates memory to store all its members. The total memory depends on the member types. For example, if a struct has two integers (4 bytes each) and one float (4 bytes), it needs 12 bytes total.

Each variable is independent—if you change `book1.pages`, it doesn't affect `book2.pages`. They're separate objects with separate memory locations.

Accessing Members of a Structure

Once you've declared a structure variable, you need to access and manipulate its members. This is done using the **dot operator (.** or **arrow operator (->)** depending on whether you're working with the variable directly or through a pointer. Accessing members is straightforward and essential for working with structures.

The Dot Operator (.) - For Direct Variables:

```
struct Employee {
    int id;
    string name;
    float salary;
};

Employee emp1;
emp1.id = 101;          // Assign value to id
emp1.name = "Alice";    // Assign value to name
emp1.salary = 50000.0;  // Assign value to salary

cout << emp1.name;      // Display: Alice
cout << emp1.salary;    // Display: 50000
```

The Arrow Operator (->) - For Pointers:

```
Employee* emp2 = new Employee();
emp2->id = 102;
emp2->name = "Bob";
emp2->salary = 55000.0;

cout << emp2->name;  // Display: Bob
```

Common Operations:

- **Read:** Access a member's current value
- **Write:** Modify a member's value
- **Calculate:** Perform operations using member values
- **Compare:** Check if member values meet certain conditions

Initialization of Structure Variables

Initialization means giving initial values to structure members when the variable is created. This is crucial because uninitialized variables contain garbage values (random data left in memory). Proper initialization ensures your program starts with known, correct values. C++ provides several ways to initialize structures, each with different benefits.

Initialization Methods:

Method 1: Member-by-Member Assignment

```
struct Product {
    int productId;
    string name;
    float price;
};

Product p1;
p1.productId = 1;
p1.name = "Laptop";
p1.price = 999.99;
```

Method 2: Aggregate Initialization (in order)

```
Product p2 = {2, "Mouse", 29.99};
```

Method 3: Designated Initialization (C++20, out of order)

```
Product p3 = {.name = "Keyboard", .price = 79.99, .productId = 3};
```

Method 4: Zero Initialization

```
Product p4 = {};  // All members set to 0/empty
```

Best Practices:

Initialize Every Member

Never leave members with garbage values. Initialize all members, even if to 0 or empty string

Use Meaningful Initial Values

Give members values that make sense in your program's context

Consider Using Default Values

In later chapters (classes), you'll learn about constructors that set default values automatically

Nested Structures and Complex Data Organization

A **nested structure** is a structure that contains another structure as a member. This allows you to create hierarchical, multi-level data organizations that mirror real-world complexity. For example, a Company might contain multiple Departments, and each Department contains Employees. Nesting structures lets you express these relationships naturally in code.

Why Nest Structures?

- **Logical Organization:** Group related data at multiple levels of abstraction
- **Real-World Modeling:** Better represent hierarchical relationships
- **Reduced Redundancy:** Avoid repeating data that belongs to a higher level

Example: Student with Address

```
struct Address {
    string street;
    string city;
    string state;
    int zipcode;
};

struct Student {
    int rollNumber;
    string name;
    Address home;  // Nested structure!
};

// Declaring and initializing
Student s1;
s1.rollNumber = 1001;
s1.name = "John";
s1.home.street = "123 Main St";
s1.home.city = "Springfield";
s1.home.state = "IL";
s1.home.zipcode = 62701;
```

Accessing Nested Members:

```
cout << s1.home.city;  // Use dot operator twice
cout << s1.name;       // Use dot once for non-nested
```

More Complex Nesting:

```
struct Company {
 string companyName;
 Address headquarters; // First level nesting
 struct Department {
 string deptName;
 int budget;
 } dept; // Second level nesting
};

Company c1;
c1.headquarters.city = "New York"; // Two levels deep
```

Enumerations: Defining Named Constants

An **enumeration** (or **enum**) is a user-defined data type that lets you create a set of named integer constants. Instead of using mysterious numbers like 0, 1, and 2 throughout your code, enums let you use meaningful names like RED, GREEN, and BLUE. This makes code far more readable and less error-prone. Enums are perfect for representing fixed sets of related values like colors, days of the week, or status codes.

Why Use Enumerations?

Readability

Code is clearer when you use `Monday` instead of `1`

Type Safety

Compiler catches invalid values; you can't accidentally use 99 for a day

Maintainability

Change constant values in one place, and all uses update automatically

Documentation

Enum names serve as inline documentation of what values are valid

Declaring and Using Enumerations:

```
// Basic enum declaration
enum Color { RED, GREEN, BLUE };

Color myColor = RED;

// Enum with explicit values
enum TrafficLight { STOP = 0, CAUTION = 1, GO = 2 };

// Enum with struct
struct Car {
    string brand;
    TrafficLight status;  // Using enum as member type
};

Car myCar;
myCar.status = CAUTION;
```

Practical Example: Student Grade System

```
enum Grade {
    A = 90,
    B = 80,
    C = 70,
    D = 60,
    F = 0
};

struct StudentRecord {
    string name;
    Grade currentGrade;
};

StudentRecord student1;
student1.name = "Emma";
student1.currentGrade = A;  // Clear and meaningful!
```

Pro Tip: By default, enum constants start at 0 and increment by 1. You can override this by specifying explicit values, as shown in the TrafficLight example above.