Data Structures PREREQUISITE

C++ From Scratch

Made By: Kevin Harvey

🎥 Introduction


```cpp
#include <iostream>
int main() {
    std::cout << "I like pizza!" << std::endl;
    std::cout << "It's really good!" << std::endl;
    return 0;
}
// which equals to print("hello world") in python
```

📽️ Introduction

```cpp
#include <iostream>
int main() {
    std::cout << "I like pizza!" << std::endl;
    std::cout << "It's really good!" << std::endl;
    return 0;
}
// which equals to print("hello world") in python
```

1. `#include <iostream>`
    - Think of it like **importing the built in functions** in Python.
    - Basically brings in C++'s input/output and other functions.
    - (Not *exactly* the same, but close enough for now 😅)

🎥 Introduction

```cpp
#include <iostream>
int main() {
    std::cout << "I like pizza!" << std::endl;
    std::cout << "It's really good!" << std::endl;
    return 0;
}
// which equals to print("hello world") in python
```

2. `int main() {`
   `}`

- **ده إيه؟**
  - This is the **main function**, the entry point of every C++ program. Without it, the program doesn’t run
  - كلمة `int` معناها إن الدالة دي **بترجع رقم صحيح (integer)**

🎥 Introduction

```cpp
#include <iostream>
int main() {
    std::cout << "I like pizza!" << std::endl;
    std::cout << "It's really good!" << std::endl;
    return 0;
}
// which equals to print("hello world") in python
```

3. `std::cout << "I like pizza!" << std::endl;`
    a. `std::cout` = “standard character output” → prints stuff to the console.
    b. `cout` ”character out” → which means that im requesting characters or a string tc be outputted (”A,”Galala”,...)
    c. `<<` = insertion operator → sends data into `cout`.
    d. `"I like pizza!"` → the text that will be displayed.
    e. `std::endl` = end line → moves the cursor to the next line (like pressing Enter) and flushes the buffer (forces the text to show immediately).

🎥 Introduction

```cpp
#include <iostream>
int main() {
    std::cout << "I like pizza!" << std::endl;
    std::cout << "It's really good!" << std::endl;
    return 0;
}
// which equals to print("hello world") in python
```

4. `return 0;`
    a. Ends the `main()` function.
    b. Returning `0` tells the OS: "✅ Program ended successfully."
    c. Returning another number (like `1`) usually means some kind of error.

🎥 Introduction


```cpp
#include <iostream>
int main() {
    std::cout << "I like pizza!" << std::endl;
    std::cout << "It's really good!" << std::endl;
    return 0;
}
// which equals to print("hello world") in python
```

5. `std`
    a. Short for **standard**.
    b. It’s like a **folder** that stores all the important tools from the **C++ Standard Library**.
    c. Inside this “folder” you’ll find `cout`, `cin`, `endl`, etc.

☝🏻 Alternatively, You could write `using namespace std;` after including iostream to avoid typing std all the time

🔢 Integers

```cpp
# include <iostream>
int main() {
    // integers (whole numbers)
    int a = 5; // in python: a = 5
    int b = 6; // in python: b = 6
    int sum = a + b; // in python: sum = a + b
    std::cout << a << std::endl; // in python: print(a)
    std::cout << b << std::endl; // in python: print(b)
    std::cout << sum << std::endl; // in python: print(sum)
}
```

1. `std::cout` = “standard character output” → prints stuff to the console.
2. `int a=5;` defines that a = 5
3. `int sum = a + b` sum is the variable of the result of a + b

🧩 **`int` = whole numbers.**

➡️ Like Python’s `a = 5`, but stricter.

👥 Doubles (Decimals)


```cpp
int main() {
    double price = 120.99; // in python: price = 120.99
    double gpa = 3.5; // in python: gpa = 3.5
    double temperature = 98.6; // in python: temperature = 98.6
    std::cout << price << std::endl; // in python: print(price)
    std::cout << gpa << std::endl; // in python: print(gpa)
    std::cout << temperature << std::endl; // in python: print(temperature)
    return 0;
}
```

`double` : clarifies that this datatype is a double (or a decimal)

⚠️ Why is it called a double (2)?

Because it can accept both integers and decimals no problem!

🔡 Chars (Single Characters)


```cpp
int main() {
    // char (single character)
    char grade = 'A'; // in python: grade = 'A'
    char letter = 'E'; // in python: letter = 'E'
    char symbol = '!'; // in python: symbol = '!'
    std::cout << grade << std::endl; // in python: print(grade)
    std::cout << letter << std::endl; // in python: print(letter)
    std::cout << symbol << std::endl; // in python: print(symbol)
    return 0;
}
```

🟫 `char` = only **one** character at a time.

> ⚠️ Use single quotes `'A'`, not `"A"`.

✅ Booleans

```cpp
#include <iostream>
int main() {
    bool isStudent = true; // in python: isStudent = True
    bool isTeacher = false; // in python: isTeacher = False
    bool isGood = true; // in python: isGood = True

    std::cout << isStudent << std::endl; // in python: print(isStudent)
    std::cout << isTeacher << std::endl; // in python: print(isTeacher)
    std::cout << isGood << std::endl; // in python: print(isGood)
    return 0;
}
```

`bool` = `true` or `false` — that's it.

🧵 Strings

```cpp
#include <iostream>
int main() {
    std::string name = "Kevin"; // in python: name = "Kevin"
    std::string day = "Monday"; // in python: day = "Monday"
    std::string age = "20"; // in python: age = "20"
    std::string food = "Pizza"; // in python: food = "Pizza"
    std::cout << "Hello " << name << std::endl; // in python: print("Hello", name)
    std::cout << "you are "<< age << " years old" << std::endl; // in python: print("you are ", age, " years old")
    return 0;
}
```

☝️ `std::string` is special — not a built-in type like `int` or `char`.

🔒 Constants


```cpp
#include <iostream>
int main() {
    const double pi = 3.1415926; // in python: pi = 3.1415926
    std::cout << pi << std::endl; // in python: print(pi)
    return 0;
}
```

⚠️ What's the difference between constants and integers/doubles?

constants are unchangeable variables in the code and cant be changed later on in the code.

once you set them, you cant change them

🧍 User Input


```cpp
#include <iostream>
int main() {
    std::string name;
    std::cout << "What's your name? "; //in python: print("What's your name? ")
    std::cin >> name; //in python: name = input()
    std::cout << "Hello, " << name << "!" << std::endl; //in python: print("Hello, " + name + "!")
    // output: Hello, Kevin!
    return 0;
}
```

`std::cin >> name;` : it's used to take input from the user (in this case the user's name)

- Works kinda like `input()` in Python, but uses `>>` instead of `=`.

☝🏻
- Unlike `std::cout` which uses `<<` (insertion operator) , `std::cin` uses `>>` (extraction operator).

Code Explanation:

1. We define that we're making a string but leaving it blank after
2. then we ask the user to enter their name using the string `"What's your name? "`
3. then we prompt the user using `std::cin >>` to take input from them then storing it in the value `name`

+ − Increments & Decrements


```cpp
int main() {
    int x = 5;
    x ++; // in python: x += 1 (increment by 1)
    x += 2; // in python: x += 2 (increment by 2)
    std::cout << x << std::endl; // in python: print(x)
    x --; // in python: x -= 1 (decrement by 1)
    x -= 2; // in python: x -= 2 (decrement by 2)
    std::cout << x << std::endl; // in python: print(x)
    // output: 8,5
    return 0;
}
```

⚠️ `++` can only be used if you want to increment by 1

`+=2` increments by 2 (adds 2 on x so in this code the equation is as following $(5 + 1 + 2) = 8$ and then take this output and put it in the decrement equation $(8 - 1 - 2)$ which equals to 5

Built In Functions

🔺 `max()` function — returns the highest value between numbers

```cpp
#include <iostream>
int main() {
    int x = 5;
    int y = 10;
    int z = std::max(x, y); // in python: z = max(x, y)
    std::cout << z << std::endl; // in python: print(z)
    return 0;
    // output: 10
}
```

➡️ **Output:** `10`

Built In Functions

🔻 `min()` function — returns the lowest value between numbers

```cpp
#include <iostream>
int main() {
    int x = 5;
    int y = 10;
    int z = std::min(x, y); // in python: z = min(x, y)
    std::cout << z << std::endl; // in python: print(z)
    return 0;
    // output: 5
}
```

➡️ **Output:** `5`

Built In Functions

⚡ `pow()` — Power Function (Returns x^y)

```cpp
#include <iostream>
#include <cmath>
int main() {
    int z;
    z = pow(2, 4); // in python, this would be z = 2 ** 4 ( 2⁴ = 16)
    std::cout << z << std::endl;
    return 0;
    // output: 16
}
```

`#include <cmath>` : contains functions like pow(), sqrt(),...

`pow()` : returns the power of two numbers

Built In Functions

🧱 `sqrt()` – Square Root Function (returns $\sqrt{x}$)

```cpp
#include <iostream>
#include <cmath>
int main() {
    int z;
    z = sqrt(16); // in python, this would be z = math.sqrt(16)
    std::cout << z << std::endl;
    return 0;
    // output: 4
}
```

Built In Functions

🎯 `round()`, `ceil()`, `floor()`

```
double a = round(2.5); // 2 (round down to the closest lowest integer)
double b = ceil(2.3);  // 3 (round up to the closest highest integer)
double c = floor(2.9); // 2 (round down to the closest lowest integer)
```

🔀 if, else & else if Statements


```cpp
#include <iostream>
int main() {
    int z= 20; // in python: z = 20
    if (z > 10) { // in python: if z > 10:
        std::cout << "z is greater than 10" << std::endl; // in python: print("z is greater than 10")
    } else if (z == 10) { // in python: elif z == 10:
        std::cout << "z is equal to 10" << std::endl; // in python: print("z is equal to 10")
    } else { // in python: else:
        std::cout << "z is less than 10" << std::endl; // in python: print("z is less than 10")
    }
    return 0;
}
```

⚙️ Switch Case (A shortcut to if, else and else if statements)


```cpp
#include <iostream>
int main() {
    int z= 20;
    switch (z) {
        case 10: // in python: if z == 10:
            std::cout << "Value is 10";
            break;
        case 20: // in python: elif z == 20:
            std::cout << "Value is 20";
            break;
        case 30: // in python: elif z == 30:
            std::cout << "Value is 30";
            break;
        default: // in python: else:
            std::cout << "Value is not 10, 20 or 30";
    }
    return 0;
}
```

⚠️

- `case` means “In case `z==10` then do ....”
- ⚡ Shortcut for `if/else if` when checking *one variable* repeatedly.

Ternary Operator (Another Alternative to if and else statements)


```cpp
#include <iostream>
int main() {
    //ternary operator
    // syntax: condition ? expression1 : expression2
    int a = 10, b = 20;
    a == b? std::cout << "a is equal to b" : std::cout << "a is not equal to b"; // in python: print("a is equal to b") if a == b else print("a is not equal to b")
    return 0;
}
```

Logical Operators


```cpp
#include <iostream>
int main() {
    // logical operators
    // && (and), || (or), ! (not)
    // && : checks if both conditions are true
    // || : checks if at least one condition is true
    // !  : reverses the condition
    int z= 20;
    int x= 10;
    if (z <= 0 && x > 0){ // in python: if z <= 0 and x > 0:
        std::cout << "Hello, World!" << std::endl;
    }
    else if (z > 0 || x > 0){ // in python: elif z > 0 or x > 0:
        std::cout << "Hello, Peeps!" << std::endl;
    }
    else if (!z){ // in python: elif not z:
        std::cout << "Hello, Geeks!" << std::endl;
    }
    else{ // in python: else:
        std::cout << "Hello, Universe!" << std::endl;
    }
    return 0;
}
```

Symbol	Meaning	Example
`&&`	and	`if (a && b)`
`||`	or	`if (a || b)`
`!`	not	`if (!a)`

🔁 Loops

🌀 While Loop

```cpp
#include <iostream>
int main() {
    int z = 10;
    while (z < 20) { // in python: while z < 20:
        std::cout << "Value of z: " << z << std::endl; // in python: print("Value of z:", z)
        z++; // in python: z += 1
    }
    return 0;
}
```

🔁 Loops

🔄 Do-While Loop

```cpp
#include <iostream>
int main() {
    int number; // Variable to store user input
    do { // do-while loop is done until the condition in the while is false
        std::cout << "Enter a Positive number: "; // in python: print("Enter a Positive number: ")
        std::cin >> number; // in python: number = int(input())
    } while (number < 0);
    std::cout << "You entered: " << number << std::endl;
    return 0;
}
```

Explanation:

the `do` function will keep running until the `while` condition is false

🔁 Loops

💫 For Loop

```cpp
#include <iostream>
int main() {
    // for loops
    // syntax: for(initialization of variable; condition; increment) {
    //     // code to be executed
    // }
    for(int i = 0; i < 10; i++) {  // in python: for i in range(10):
        std::cout << "Hello, World!" << std::endl; // in python: print("Hello, World!")
    }
    return 0;
}
```

⏭️ 🔗Continue and break


```cpp
// Continue
#include <iostream>
int main() {
    for(int i = 0; i < 10; i++) {  // in python: for i in range(10):
        if (i == 5) {  // in python: if i == 5:
            continue;  // in python: continue (skip current iteration in this case it skips 5)
            std::cout << i << std::endl;  // in python: print(i)
            // output: 0, 1, 2, 3, 4, 6, 7, 8, 9 (notice that 5 isnt there)
        }
    }
    return 0;
}
```

⏭️🔗Continue and break


```cpp
// break
#include <iostream>
int main() {
    for(int i = 0; i < 10; i++) {  // in python: for i in range(10):
        if (i == 5) {  // in python: if i == 5:
            break;  // in python: break (stops before 5 by 1 and exits the loop)
            std::cout << i << std::endl;  // in python: print(i)
            // output: 0 1 2 3 4
        }
    }
    return 0;
}
```

🧠 User-Defined Functions


```cpp
#include <iostream>
// Syntax: void {FunctionName}(){
}
void UserDefinedFunction() { // in python def UserDefinedFunction():
    std::cout << "This is a user-defined function." << std::endl; // in python: print("This is a user-defined function.")
}
int main() {
    UserDefinedFunction();

    return 0;
}
```

🧩 `void` = returns nothing.

If you want to return something, use `int`, `double`, etc.

🚗 Arrays


```cpp
#include <iostream>
int main() {
    std::string cars[] = {"Mercedes", "BMW", "MG", "Peugeot"}; // in python cars = ["Mercedes", "BMW", "MG", "Peugeot"]
    std::cout << cars[0] << std::endl; // in python print(cars[0]) outputs Mercedes
    std::cout << cars[1] << std::endl; // in python print(cars[1]) outputs BMW
    std::cout << cars[2] << std::endl; // in python print(cars[2]) outputs MG
    std::cout << cars[3] << std::endl; // in python print(cars[3]) outputs Peugeot
    return 0;
}
```

☝🏻 ⚙️ Arrays start at index **0**.

🎯 Pointers


```cpp
#include <iostream>
int main() {
    int age = 25;
    int *pAge = &age; // Pointer to age
    std::cout << "Age: " << pAge << std::endl;  // output: 0x7ff7b9d6db68 (memory address of age)
    return 0;
}
```

☝🏻

- 📍 Pointers = store **memory addresses**, not values.
- So `pAge` points to where `age` is living in memory 🧠.

🎯 Pointers


```
int num = 10;
int* ptr = &num;
```

- `&num` → gives the address of `num`
- `*ptr` → holds that address
- `ptr` → accesses the value at that address (a.k.a. **dereferencing**) in this example the value is 10

💡 Think of it like a **treasure map** —

`ptr` is the map 🗺️, and `*ptr` is the treasure 💎.

1. Memory Basics

When you run a C++ program, memory is divided mainly into two zones:

- **Stack:**
  - Stores variables created inside functions.
  - Automatically managed — memory is freed when the function ends.
  - Fast but limited size.
  - Example:

```
void func() {
    int x = 5; // x is on the stack
}
```

- **Heap:**
  - Used for **dynamic memory allocation** (you control it).
  - Slower but much bigger.
  - You must manually free memory (or use smart pointers).
  - Example:

```
int* p = new int(10); // allocated on heap
delete p; // free it!
```

💡 **Stack = automatic, Heap = manual.**

🎯 2.Pointers

1. **What is a Pointer?**
   - A pointer is a variable that *stores the memory address* of another variable.
   - Example:
     Here, `ptr` stores the memory address of `x`.

```
int x = 10;
int *ptr = &x;
```

2. **Why Do We Need Pointers?**
   - To directly access and modify memory.
   - Useful in dynamic memory allocation (`new` and `delete`).
   - Enables passing large objects to functions *without copying* them.
   - Allows creating complex structures like linked lists, trees, etc.

🎯 2.Pointers

3. **What Happens If You Don’t Reference a Pointer?**
   - If a pointer isn’t assigned a valid address, it can contain *garbage data* (random memory address).
   - Accessing that causes *undefined behavior* (crash, weird output, etc.).
   - Always initialize pointers:

```
int* ptr = nullptr; // Safe default
```

4. **What is a Null Pointer?**
   - A pointer that points to *nothing* (address = 0).
   - Used to signal that the pointer is *not currently referencing anything*.
   - Example:

```
int* ptr = nullptr;
if (ptr == nullptr) {
    cout << "Pointer is empty!" << endl;
}
```

🎯 2.Pointers

Common Mistakes

❌ Uninitialized pointer:

```
int* p;  // random address (danger!)
*p = 10; // the program will crash
```

✅ Always initialize:

```
int* p = nullptr; // safe
```

🔗 3. References

A **reference** is like a **nickname** for an existing variable — it's another name, not another variable.

```
int x = 5;
int& ref = x;
ref = 10;
cout << x; // prints 10
```

💡 **Pointer vs Reference:**

Feature	Pointer	Reference
Can be null?	Yes (`nullptr`)	No
Reassigned?	Yes	No
Syntax	`*` and `&`	Just `&` when declaring
Access	Must dereference	Acts like normal variable

🧩 4. Classes & Objects

What's a Class?

A **class** is like a *blueprint* for objects — it defines what they are and what they can do.

```cpp
#include <iostream>
#include <string>

using namespace std;
class Car { // in python: class Car:
// attributes
public:
    string brand; // in python: self.brand
    int speed; // in python: self.speed

    void drive() {
        cout << brand << " is driving at " << speed << " km/h!" << endl;
    }
};
int main() {
    Car myCar; // in python: myCar = Car()
    myCar.brand = "Toyota"; // in python: myCar.brand = "Toyota"
    myCar.speed = 100; // in python: myCar.speed = 100
    myCar.drive(); // in python: myCar.drive()
    return 0;
}
```

What's an Object?

An **object** is an *instance* of a class — like a real car built from the blueprint.

```cpp
Car myCar; // in python: myCar = Car()
myCar.brand = "Toyota"; // in python: myCar.brand = "Toyota"
myCar.speed = 100; // in python: myCar.speed = 100
myCar.drive(); // in python: myCar.drive()
```

💼 5. Attributes and Methods

- **Attributes** = variables inside a class (like `brand`, `speed`)
- **Methods** = functions inside a class (like `drive()`)

They're often declared under **public**, **private**, or **protected** access:

```cpp
#include <iostream>

class BankAccount { // in python: class BankAccount: (class)
private: // private attributes
    double balance; // in python: self.balance

public:
    void deposit(double amount) { // in python: def deposit(self, amount): (method)
        balance += amount; // in python: self.balance += amount
    }

    double getBalance() { // in python: def get_balance(self):
        return balance; // in python: return self.balance
    }
};
```

💡 `private` → only accessible inside the class

💡 `public` → accessible from anywhere

6. Constructors & Destructors

Constructor

A **constructor** runs automatically when an object is created.

Used to initialize variables or set up things.

```
#include <iostream>
class Student {
public: // public access modifier
    std::string name;
    int age;

    // Constructor
    Student(string n, int a) { // in python: def __init__(self, n, a):
        name = n; // in python: self.name = n
        age = a; // in python: self.age = a
    }
};
```

```
Student s1("Kevin", 21);
cout << s1.name; // Kevin
```

🏗️ 6. Constructors & Destructors

Destructor

A **destructor** runs automatically when an object is destroyed (goes out of scope).

Used to release memory or clean up.

```
class Student {
public:
    Student() {
        cout << "Constructor called!" << endl;
    }
    ~Student() {
        cout << "Destructor called!" << endl;
    }
};

void createStudent() {
    Student s;
} // when function ends, destructor is called
```

💡 It's like:

🛠️ Constructor = setup phase

🔥 Destructor = cleanup phase

🏗️ 6. Constructors & Destructors

⚡ Quick Recap

Concept	Description	Example
Pointer	Stores address	`int* p = &x;`
Reference	Alias for variable	`int& ref = x;`
Class	Blueprint	`class Car { ... };`
Object	Instance of class	`Car c;`
Attribute	Class variable	`string name;`
Method	Class function	`void drive();`
Constructor	Initializes object	`Student("Ali", 20);`
Destructor	Cleans up	`~Student()`

🏛️ 7. Encapsulation — “Hide your mess.”

Definition:

Encapsulation means **wrapping up data and methods together** inside a class — and hiding the details you don’t want the outside world to mess with.

```cpp
#include <iostream>

class BankAccount { // in python: class BankAccount: (Hidden from the outside world)
private: // private attributes
    double balance; // in python: self.balance

public:
    void deposit(double amount) { // in python: def deposit(self, amount):
        balance += amount; // in python: self.balance += amount
    }

    double getBalance() { // in python: def get_balance(self):
        return balance; // in python: return self.balance
    }
};
```

Here:

- `balance` is **private**, so no one can directly change it.
- You can only interact with it using the **public** methods like `deposit()`.

💡 **Encapsulation = data protection + clean interface.**

👉 Think of it like Instagram: you can see someone’s posts (public), but not their DMs (private).

🧩 8. Inheritance — “Kids get traits from parents.”

Definition:

Inheritance means **a class can “inherit” properties and methods** from another class.

It’s how we reuse code and create class hierarchies.

```cpp
#include <iostream>
using namespace std;
class Animal { // in python: class Animal:
public:
    void eat() { // in python: def eat(self):
        cout << "Eating..." << endl; // in python: print("Eating...")
    }
};

class Dog : public Animal { // in python: class Dog(Animal):
public:
    void bark() { // in python: def bark(self):
        cout << "Woof! 🐶" << endl; // in python: print("Woof! 🐶")
    }
};
```

```cpp
int main() {
    Dog myDog; // in python: myDog = Dog()
    myDog.eat(); // in python: myDog.eat()
    myDog.bark(); // in python: myDog.bark()
    return 0;
}
```

💡 `Dog` inherits all public stuff from `Animal`.

🧩 8. Inheritance — “Kids get traits from parents.”

INHERITANCE الوراثة

- **It’s a way that allows a new class to inherit all of the features another class had**

فلنفترض ان الراجل ده لونه ابيض و شعره اسود

وقتها لما يخلف هيتنقل كل صفاته للطفل بتاعه

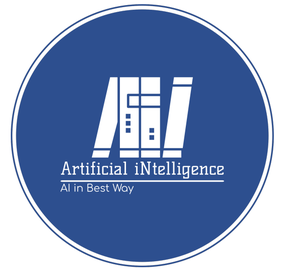
بس وقتها ممكن يزيد عليه حاجات من جينات الام

Is a

حيوان

فا في الطبيعي هيورث صفات الحيوان بس هيبقى ليه صفات زيادة

🧩 8. Inheritance — “Kids get traits from parents.”



Types of Inheritance:

Type	Syntax	Description
Single	`class Dog : public Animal`	One parent
Multiple	`class Cyborg : public Human, public Robot`	Many parents (Parents are `Human` and `Robot`) and son is `Cyborg`
Multilevel	`class Puppy : public Dog`	Child of a child (which is a grandparent)
Hierarchical	Multiple children from one parent	
Hybrid	Combo of types (complex 😅)	

Access Specifiers in Inheritance

Access in Base	`public` inheritance	`protected` inheritance	`private` inheritance
`public`	stays public	becomes protected	becomes private
`protected`	stays protected	stays protected	becomes private
`private`	not inherited	not inherited	not inherited

💡 99% of the time, you’ll use **public inheritance** — it’s the default in OOP examples.

🧠 9. Polymorphism — “Same function, different vibes.”

Definition:

Polymorphism = **same name, different behavior.**

(“Poly” = many, “morph” = forms.)

It lets us treat different objects in the same way — but each one behaves differently.

📍 Static Polymorphism (Compile-time)

Function Overloading — same function name, different parameters.

```
#include <iostream>
class Math {
public:
    int add(int a, int b) { return a + b; } // in python: def add(self, a, b):
    double add(double a, double b) { return a + b; } // in python: def add(self, a: float, b: float) -> float:
};
```

Both are called `add()`, but the compiler picks the right one based on arguments.

🧠 9. Polymorphism — “Same function, different vibes.”

📍 Dynamic Polymorphism (Runtime)

Using **virtual functions** and **inheritance**.

```
class Animal {
public:
    virtual void sound() {
        cout << "Some sound..." << endl;
    }
};

class Dog : public Animal {
public:
    void sound() override {
        cout << "Woof! 🐶" << endl;
    }
};

class Cat : public Animal {
public:
    void sound() override {
        cout << "Meow! 🐱" << endl;
    }
};
```

```
Animal* a1 = new Dog();
Animal* a2 = new Cat();

a1->sound(); // Woof!
a2->sound(); // Meow!
```

💡 The magic here is **virtual** — it tells C++ to look at the *actual object type* at runtime, not just the pointer type.

Without `virtual`, both would’ve printed “Some sound...”

Thank you so much for tuning in!❤️