**Learning C++23 – Complete Course Notes**

**Part 1 – Getting Started**

1. Why Learn C++?
2. How C++ Works (Compiler, Linker, Executable)
3. Hello World in Visual Studio

**Part 2 – C++ Fundamentals**

1. Variables & Data Types
2. Functions & Parameters
3. Conditions & Loops
4. Pointers & References
5. Arrays & Strings
6. Header Files & Compilation Units

**Part 3 – Object-Oriented Programming**

1. Classes & Structs
2. Constructors & Destructors
3. Inheritance & Polymorphism
4. Virtual Functions & Interfaces
5. Encapsulation & Visibility (public/private/protected)

**Part 4 – Advanced Language Features**

1. Static, const, and mutable
2. Operator Overloading & this pointer
3. Templates & Macros
4. Lambdas & Function Pointers
5. Namespaces & Modules (C++20/23)

**Part 5 – Memory Management**

1. Stack vs Heap
2. Object Lifetime
3. new / delete & Smart Pointers
4. Move Semantics & R-value references
5. Advanced Memory Safety

**Part 6 – STL & Data Structures**

1. Vectors, Arrays, Maps
2. Iterators & Algorithms
3. Optionals, Variants, Any (C++17+)
4. Unions & Type Punning

**Part 7 – Performance & Concurrency**

1. Timing & Benchmarking
2. Threads & Async
3. Optimizations & Best Practices

**Part 8 – Debugging & Workflow**

1. Debugging with Visual Studio
2. Using Libraries & Linking
3. Best Practices for Large Projects

## Part 1 – Getting Started

### 1. Why Learn C++?

C++ is one of the most influential and widely used programming languages. It combines:

* **Low-level control** (like C, you can work directly with memory, pointers, hardware).
* **High-level abstractions** (classes, templates, STL, lambdas).
* **Performance** → Often chosen for systems programming, game engines (Unreal Engine), financial systems, real-time apps.

If your main concern is speed, efficiency, and control → **C++ is the right choice**.  
If you don’t need performance and memory control, languages like Python may be more productive.

### 2. How C++ Works (Compiler, Linker, Executable)

When you write a .cpp file, it doesn’t directly “run.” Instead, it goes through several stages:

1. **Preprocessing**
   * Handles #include, #define, and macros.
   * Example:
   * #include <iostream>

→ literally pastes the contents of <iostream> into your file before compilation.

1. **Compilation**
   * Each .cpp file (translation unit) is compiled into an **object file** (.obj on Windows, .o on Linux).
   * The compiler turns your code into **machine instructions** for the CPU.
2. **Linking**
   * Combines object files + libraries → final executable (.exe, .out).
   * If a function is declared but not defined, you’ll get **linker errors** (“unresolved external symbol”).

Key difference from Python/JavaScript:

* **C++:** compiled → fast execution.
* **Python:** interpreted → slower, but easier to develop.

### 3. Hello World in Visual Studio

Your very first program:

#include <iostream> // gives access to std::cout

int main() {

std::cout << "Hello, World!" << std::endl;

return 0; // program ends successfully

}

* #include <iostream> → imports input/output stream.
* main() → entry point of every C++ program.
* std::cout → console output.
* std::endl → newline + flush buffer.

In **Visual Studio**:

1. Create a new C++ Console Project.
2. Replace main.cpp content with the code above.
3. Run with **Ctrl+F5**.

## Part 2 – C++ Fundamentals

### 4. Variables and Data Types

C++ provides several built-in data types and also allows user-defined types.

**Basic built-in types:**

* int → integer numbers
* float, double → floating-point numbers
* char → single character (1 byte)
* bool → true/false
* void → represents no value (used for functions that return nothing)

**Example:**

int age = 25;

double pi = 3.14159;

char grade = 'A';

bool is\_active = true;

**Strings:**  
C++ has two main types of strings:

* C-style strings: char name[] = "John";
* Modern C++: std::string name = "John"; (preferred, requires <string>).

**Best practice:** Always use std::string for safety and convenience.

### 5. Functions and Parameters

Functions allow you to split code into reusable blocks.

**Example:**

int add(int a, int b) {

return a + b;

}

int main() {

int result = add(5, 3);

std::cout << "Sum = " << result << std::endl;

}

**Return types:**

* Functions can return any type or void if nothing is returned.
* Functions can take parameters by value, by reference, or by pointer.

**Parameter passing:**

void by\_value(int x) { x = 20; }

void by\_reference(int& x) { x = 20; }

void by\_pointer(int\* x) { \*x = 20; }

int main() {

int a = 10;

by\_value(a); // does not change a

by\_reference(a); // changes a to 20

by\_pointer(&a); // also changes a to 20

}

### 6. Conditions and Loops

C++ has standard conditional statements and loops.

**If-else statement:**

int x = 5;

if (x > 0) {

std::cout << "Positive" << std::endl;

} else {

std::cout << "Non-positive" << std::endl;

}

**Loops:**

// for loop

for (int i = 0; i < 5; i++) {

std::cout << i << " ";

}

// while loop

int j = 0;

while (j < 5) {

std::cout << j << " ";

j++;

}

// do-while loop

int k = 0;

do {

std::cout << k << " ";

k++;

} while (k < 5);

**Range-based for (modern C++):**

std::vector<int> values = {1, 2, 3, 4};

for (int v : values) {

std::cout << v << std::endl;

}

### 7. Pointers and References

**Pointers** hold memory addresses.

int x = 10;

int\* ptr = &x; // pointer to x

std::cout << \*ptr; // dereference pointer, prints 10

**References** are aliases for variables.

int y = 20;

int& ref = y; // ref is another name for y

ref = 30; // changes y as well

**Differences:**

* A pointer can be reassigned to point to another variable; a reference cannot.
* A pointer can be null (nullptr), but a reference must always refer to an object.

### 8. Arrays and Strings

**C-style arrays**

int numbers[5] = {1, 2, 3, 4, 5};

std::cout << numbers[2]; // prints 3

* Fixed size, cannot grow dynamically.
* Memory is contiguous.

**Modern C++ alternatives**

* std::array<T, N> → safer, fixed size, stack allocated.
* std::vector<T> → dynamic size, heap allocated.

Example with std::array:

#include <array>

std::array<int, 3> arr = {1, 2, 3};

for (int v : arr) std::cout << v << " ";

Example with std::vector:

#include <vector>

std::vector<int> vec = {1, 2, 3};

vec.push\_back(4); // add element

for (int v : vec) std::cout << v << " ";

**Strings**

* Old style:
* char name[] = "John";
* Modern style (preferred):
* #include <string>
* std::string name = "John";
* std::cout << name.size();

### 9. Header Files and Compilation Units

C++ code is organized into **header files** (.h or .hpp) and **source files** (.cpp).

* Header files → contain **declarations** (function prototypes, class definitions).
* Source files → contain **definitions** (actual implementations).

Example:

**math.h**

#pragma once

int add(int a, int b);

**math.cpp**

#include "math.h"

int add(int a, int b) {

return a + b;

}

**main.cpp**

#include <iostream>

#include "math.h"

int main() {

std::cout << add(2, 3);

}

**Compilation process:**

* Each .cpp file is compiled into an object file.
* The **linker** combines them into the final executable.

### 10. Control Flow Keywords

* break → exits a loop early.
* continue → skips to the next loop iteration.
* return → exits a function.

Example:

for (int i = 0; i < 5; i++) {

if (i == 2) continue; // skip i == 2

if (i == 4) break; // stop the loop

std::cout << i << " ";

}

## Part 3 – Object-Oriented Programming

### 11. Classes and Structs

Classes and structs are used to define **user-defined types**. They can hold data (members) and behavior (methods).

**Example of a struct:**

struct Point {

int x;

int y;

};

**Example of a class:**

class Person {

private:

std::string name;

int age;

public:

Person(std::string n, int a) : name(n), age(a) {}

void greet() {

std::cout << "Hello, my name is " << name

<< " and I am " << age << " years old." << std::endl;

}

};

**Difference between struct and class:**

* struct → members are public by default.
* class → members are private by default.

### 12. Constructors and Destructors

* **Constructor:** Special method called when an object is created.
* **Destructor:** Special method called when an object goes out of scope or is deleted.

class File {

private:

std::string filename;

public:

File(std::string f) : filename(f) {

std::cout << "Opening " << filename << std::endl;

}

~File() {

std::cout << "Closing " << filename << std::endl;

}

};

int main() {

File f("data.txt"); // Constructor runs

} // Destructor runs automatically here

This technique is called **RAII** (Resource Acquisition Is Initialization): resources are acquired in the constructor and released in the destructor.

### 13. Inheritance

Inheritance allows one class to reuse code from another.

class Animal {

public:

void eat() { std::cout << "Eating..." << std::endl; }

};

class Dog : public Animal {

public:

void bark() { std::cout << "Woof!" << std::endl; }

};

int main() {

Dog d;

d.eat(); // from Animal

d.bark(); // from Dog

}

Types of inheritance:

* public → base class public members stay public.
* protected → base class public members become protected.
* private → base class members become private.

### 14. Polymorphism and Virtual Functions

Polymorphism means “many forms.” It allows different classes to be treated through a common interface.

class Animal {

public:

virtual void speak() {

std::cout << "..." << std::endl;

}

};

class Dog : public Animal {

public:

void speak() override {

std::cout << "Woof!" << std::endl;

}

};

class Cat : public Animal {

public:

void speak() override {

std::cout << "Meow!" << std::endl;

}

};

int main() {

Animal\* a = new Dog();

a->speak(); // "Woof!" because function is virtual

delete a;

}

* virtual keyword tells the compiler to use **dynamic dispatch** (runtime lookup).
* override keyword ensures you are actually overriding a virtual function.

### 15. Interfaces (Pure Virtual Functions)

C++ has no dedicated interface keyword, but **pure virtual functions** allow us to define abstract interfaces.

class Drawable {

public:

virtual void draw() = 0; // pure virtual function

};

class Circle : public Drawable {

public:

void draw() override {

std::cout << "Drawing Circle" << std::endl;

}

};

* A class with at least one pure virtual function is called an **abstract class**.
* It cannot be instantiated directly.