**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.

### 16. Encapsulation and Visibility

Encapsulation means hiding internal details of a class and exposing only what is necessary.

**Access specifiers in C++:**

* private → accessible only inside the class.
* protected → accessible in the class and its derived classes.
* public → accessible from anywhere.

class BankAccount {

private:

double balance; // hidden from outside

public:

BankAccount(double initial) : balance(initial) {}

void deposit(double amount) { balance += amount; }

void withdraw(double amount) { if (amount <= balance) balance -= amount; }

double get\_balance() const { return balance; }

};

This ensures the balance cannot be changed directly, only through controlled methods.

### 17. The static Keyword

**Use case 1: Static variables inside functions**

* Preserved across function calls.

void counter() {

static int count = 0; // created once

count++;

std::cout << "Count: " << count << std::endl;

}

int main() {

counter(); // 1

counter(); // 2

}

**Use case 2: Static class members**

* Shared by all instances of the class.

class Student {

public:

static int count;

Student() { count++; }

};

int Student::count = 0;

int main() {

Student s1, s2;

std::cout << Student::count; // prints 2

}

**Use case 3: Static functions in classes**

* Belong to the class, not to any object. Cannot access non-static members.

### 18. The const and mutable Keywords

**const with variables:**

* const int x = 10; → cannot change value.

**const with pointers:**

const int\* p1; // pointer to const int (value cannot change)

int\* const p2; // const pointer (address cannot change)

const int\* const p3; // both const

**const with member functions:**

class Person {

private:

std::string name;

public:

Person(std::string n) : name(n) {}

std::string get\_name() const { return name; } // cannot modify members

};

**mutable keyword:**

* Allows a variable to be modified even in a const function.

class Logger {

private:

mutable int count = 0;

public:

void log() const { count++; } // valid because count is mutable

};

### 19. Operator Overloading

C++ allows defining how operators work for user-defined types.

Example:

class Vector {

public:

int x, y;

Vector(int a, int b) : x(a), y(b) {}

Vector operator+(const Vector& other) {

return Vector(x + other.x, y + other.y);

}

};

int main() {

Vector v1(2, 3), v2(4, 5);

Vector v3 = v1 + v2; // uses overloaded +

std::cout << v3.x << "," << v3.y; // 6,8

}

* Operators are just functions.
* Common overloads: +, -, \*, ==, << (stream output).

### 20. The this Pointer

Inside class methods, this is a pointer to the current object.

class Person {

private:

std::string name;

public:

Person(std::string n) : name(n) {}

void greet() {

std::cout << "Hello, I am " << this->name << std::endl;

}

};

Uses of this:

* Disambiguate variable names.
* Return the current object (\*this).

## Part 4 – Memory Management

### 21. Stack vs Heap Memory

**Stack memory**

* Automatically managed.
* Stores local variables and function calls.
* Fast allocation and deallocation.

void func() {

int x = 5; // x is on the stack

}

**Heap memory**

* Manually managed.
* Used for dynamic allocation.
* Slower than stack.

void func() {

int\* p = new int(5); // allocated on heap

delete p; // must free manually

}

**Key difference:**

* Stack memory is freed automatically when scope ends.
* Heap memory must be explicitly managed (or use smart pointers).

### 22. Copy Constructor and Assignment Operator

When you copy objects, C++ may perform a **shallow copy** unless you define your own.

class MyClass {

private:

int\* data;

public:

MyClass(int value) { data = new int(value); }

// Copy constructor

MyClass(const MyClass& other) {

data = new int(\*other.data);

}

// Assignment operator

MyClass& operator=(const MyClass& other) {

if (this == &other) return \*this;

delete data;

data = new int(\*other.data);

return \*this;

}

~MyClass() { delete data; }

};

This prevents issues like **double deletion**.

### 23. Move Semantics (C++11 and later)

Move semantics optimize performance by transferring resources instead of copying.

class MyString {

private:

char\* data;

public:

MyString(const char\* str) {

data = new char[strlen(str)+1];

strcpy(data, str);

}

// Move constructor

MyString(MyString&& other) noexcept {

data = other.data;

other.data = nullptr;

}

~MyString() { delete[] data; }

};

Usage:

MyString s1("Hello");

MyString s2 = std::move(s1); // moves instead of copies

### 24. Smart Pointers

Smart pointers manage heap memory automatically. They are in <memory>.

**unique\_ptr**

* Owns an object exclusively.
* Cannot be copied.

#include <memory>

std::unique\_ptr<int> p1 = std::make\_unique<int>(10);

**shared\_ptr**

* Reference-counted ownership.
* Multiple pointers can own the same object.

#include <memory>

std::shared\_ptr<int> p2 = std::make\_shared<int>(20);

**weak\_ptr**

* Observes an object managed by shared\_ptr without increasing reference count.
* Prevents circular references.

### 25. Memory Safety and RAII

RAII (Resource Acquisition Is Initialization):

* Resources are acquired in the constructor.
* Released in the destructor.

Example with file handling:

class File {

private:

std::FILE\* f;

public:

File(const char\* path) { f = fopen(path, "r"); }

~File() { if (f) fclose(f); }

};

Benefit: Even if exceptions occur, resources are properly released.

## Part 5 – Standard Template Library (STL) and Data Structures

The STL provides ready-to-use data structures and algorithms. It saves time and ensures efficiency.

### 26. Vectors, Arrays, and Lists

**std::vector** (dynamic array, most commonly used)

#include <vector>

#include <iostream>

int main() {

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

v.push\_back(4);

for (int n : v) std::cout << n << " "; // 1 2 3 4

}

* Stores elements in contiguous memory.
* Fast random access.
* Can resize dynamically.

**std::array** (fixed size, safer than raw array)

#include <array>

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

**std::list** (doubly linked list)

#include <list>

std::list<int> l = {1, 2, 3};

l.push\_front(0);

* Efficient insertions/removals in the middle.
* Slower random access compared to vector.

### 27. Maps and Sets

**std::map** → key-value pairs, ordered by key.

#include <map>

std::map<std::string, int> ages;

ages["Alice"] = 25;

ages["Bob"] = 30;

**std::unordered\_map** → key-value pairs, fast lookup (uses hashing).

#include <unordered\_map>

std::unordered\_map<std::string, int> scores;

scores["Math"] = 95;

scores["Physics"] = 88;

**std::set** → stores unique values in sorted order.

#include <set>

std::set<int> s = {3, 1, 2, 2}; // duplicates ignored

**std::unordered\_set** → unique values, order not guaranteed, fast lookup.

### 28. Iterators

Iterators are like pointers used to traverse containers.

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

for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {

std::cout << \*it << " ";

}

Modern C++ prefers range-based for loops:

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

### 29. Algorithms

The <algorithm> header provides common operations:

#include <algorithm>

std::vector<int> v = {5, 3, 8, 1};

std::sort(v.begin(), v.end()); // sort

bool found = std::binary\_search(v.begin(), v.end(), 3); // search

std::reverse(v.begin(), v.end()); // reverse

### 30. Optional, Variant, and Any (C++17 and later)

**std::optional** → represents a value that may or may not exist.

#include <optional>

std::optional<int> getValue(bool flag) {

if (flag) return 42;

return std::nullopt;

}

**std::variant** → holds one of several types.

#include <variant>

std::variant<int, std::string> data;

data = 10;

data = "Hello";

**std::any** → holds any single value (type erased).

#include <any>

std::any a = 5;

a = std::string("text");

### 31. Unions

Unions share memory between members. Only one field is active at a time.

union Data {

int i;

float f;

char c;

};

Data d;

d.i = 10;

std::cout << d.i;

Used in low-level programming, not common in high-level C++ code.

## Part 6 – Performance and Concurrency

### 32. Measuring Time and Benchmarking

To measure how long code takes to run, use <chrono>.

#include <iostream>

#include <chrono>

int main() {

auto start = std::chrono::high\_resolution\_clock::now();

// Code to measure

long long sum = 0;

for (int i = 0; i < 1e6; i++) sum += i;

auto end = std::chrono::high\_resolution\_clock::now();

std::chrono::duration<double> elapsed = end - start;

std::cout << "Elapsed time: " << elapsed.count() << " seconds\n";

}

* Use std::chrono::steady\_clock for precise measurements.
* Benchmarking libraries like Google Benchmark can be used for professional testing.

### 33. Multithreading

C++11 introduced <thread> for concurrency.

#include <iostream>

#include <thread>

void task(int id) {

std::cout << "Thread " << id << " is running\n";

}

int main() {

std::thread t1(task, 1);

std::thread t2(task, 2);

t1.join();

t2.join();

}

* join() → wait for the thread to finish.
* detach() → thread runs independently.

### 34. Mutex and Synchronization

When multiple threads access shared data, use a **mutex** to prevent race conditions.

#include <iostream>

#include <thread>

#include <mutex>

int counter = 0;

std::mutex mtx;

void increment() {

for (int i = 0; i < 100000; i++) {

std::lock\_guard<std::mutex> lock(mtx);

counter++;

}

}

int main() {

std::thread t1(increment);

std::thread t2(increment);

t1.join();

t2.join();

std::cout << "Counter = " << counter << "\n";

}

* std::lock\_guard automatically locks/unlocks a mutex.
* Prevents data corruption in multithreaded programs.

### 35. Async and Futures

C++ provides std::async for running tasks asynchronously.

#include <future>

#include <iostream>

int compute() {

return 42;

}

int main() {

std::future<int> result = std::async(compute);

std::cout << "Result = " << result.get() << "\n";

}

* std::future retrieves the result later.
* std::promise can be used for more control.

### 36. Thread Pools (C++20 and libraries)

Instead of creating/destroying threads repeatedly, thread pools reuse threads.

C++ standard does not yet have a built-in thread pool, but you can implement or use libraries like Boost.

Simple example using multiple threads in a pool-like fashion:

#include <vector>

#include <thread>

void work(int id) {

std::cout << "Worker " << id << " running\n";

}

int main() {

std::vector<std::thread> workers;

for (int i = 0; i < 4; i++) {

workers.emplace\_back(work, i);

}

for (auto& t : workers) t.join();

}

## Part 7 – Debugging, Error Handling, and Best Practices

### 37. Debugging in C++

Debugging tools help track down logic errors, crashes, or performance bottlenecks.

**Debugging with Visual Studio**

* Place breakpoints (F9) in your code.
* Run in debug mode (F5).
* Step through code line by line (F10 / F11).
* Inspect variables in the watch window.
* Use the call stack to trace function calls.

**Common runtime errors**

* Segmentation fault → accessing invalid memory (e.g., dereferencing nullptr).
* Buffer overflow → writing outside array bounds.
* Memory leaks → forgetting to delete dynamically allocated memory (use smart pointers).

### 38. Error Handling with Exceptions

C++ uses try, catch, and throw for exceptions.

#include <iostream>

#include <stdexcept>

int divide(int a, int b) {

if (b == 0) throw std::runtime\_error("Division by zero");

return a / b;

}

int main() {

try {

std::cout << divide(10, 0);

} catch (const std::exception& e) {

std::cout << "Error: " << e.what() << std::endl;

}

}

* Only throw exceptions for exceptional situations.
* Catch exceptions by reference (const std::exception&).
* Avoid throwing raw types like int or char.

### 39. Assertions

Assertions help detect logic errors during development.

#include <cassert>

int main() {

int x = 5;

assert(x > 0); // program aborts if false

}

* Assertions are disabled in release mode (by defining NDEBUG).
* Use them for conditions that should always be true in development.

### 40. Logging

Instead of only using std::cout, structured logging can help.

Basic example:

#include <iostream>

void log(const std::string& msg) {

std::cout << "[LOG] " << msg << std::endl;

}

int main() {

log("Application started");

}

For production, use logging libraries such as **spdlog** or **glog**.

### 41. Best Practices

1. Prefer modern C++ features (smart pointers, auto, range-based for).
2. Use RAII to manage resources automatically.
3. Avoid raw new and delete unless absolutely necessary.
4. Keep functions small and focused.
5. Minimize global variables.
6. Use const wherever possible.
7. Follow consistent naming conventions.
8. Write unit tests to validate code correctness.

### 42. Large Project Organization

* Group related classes into **modules/namespaces**.
* Keep headers clean (only include what is necessary).
* Use forward declarations to reduce compile time.
* Break large projects into libraries.
* Prefer CMake for cross-platform builds.

## Part 8 – Modern C++ (Advanced Features C++11 → C++23)

### 43. Templates

Templates allow writing generic code that works with any type.

**Function template:**

template <typename T>

T add(T a, T b) {

return a + b;

}

int main() {

std::cout << add(2, 3) << std::endl; // int

std::cout << add(2.5, 3.1) << std::endl; // double

}

**Class template:**

template <typename T>

class Box {

T value;

public:

Box(T v) : value(v) {}

T get() { return value; }

};

int main() {

Box<int> b1(10);

Box<std::string> b2("Hello");

}

### 44. Lambdas (Anonymous Functions)

Introduced in C++11, lambdas are inline functions.

auto square = [](int x) { return x \* x; };

std::cout << square(5); // 25

**With captures:**

int factor = 2;

auto multiply = [factor](int x) { return x \* factor; };

std::cout << multiply(10); // 20

Capture options:

* [ ] → no capture
* [=] → capture by value
* [&] → capture by reference
* [=, &x] → by value, except x by reference

### 45. constexpr (Compile-Time Computation)

constexpr allows functions to be evaluated at compile time if possible.

constexpr int square(int x) {

return x \* x;

}

int arr[square(5)]; // size = 25, evaluated at compile time

This improves performance by moving computations to compilation.

### 46. auto and decltype

* auto → deduces the type automatically.

auto x = 10; // int

auto y = 3.14; // double

* decltype → extracts the type of an expression.

int a = 5;

decltype(a) b = 10; // b is also int

### 47. Range-based For Loops (C++11)

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

for (auto n : v) {

std::cout << n << " ";

}

With reference:

for (auto& n : v) n \*= 2; // modifies elements

### 48. Structured Bindings (C++17)

Allows unpacking multiple values.

std::pair<int, std::string> p = {1, "Alice"};

auto [id, name] = p;

std::cout << id << " " << name;

### 49. Modules (C++20)

Modules improve compilation speed and reduce header complexity.

// math.ixx

export module math;

export int add(int a, int b) { return a + b; }

// main.cpp

import math;

#include <iostream>

int main() {

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

}

Instead of #include, we use import.

### 50. Coroutines (C++20)

Coroutines are functions that can suspend and resume.

#include <coroutine>

#include <iostream>

struct Generator {

struct promise\_type {

int current\_value;

auto get\_return\_object() { return Generator{this}; }

auto initial\_suspend() { return std::suspend\_always{}; }

auto final\_suspend() noexcept { return std::suspend\_always{}; }

void return\_void() {}

auto yield\_value(int value) { current\_value = value; return std::suspend\_always{}; }

void unhandled\_exception() { std::terminate(); }

};

promise\_type\* promise;

};

Generator counter() {

for (int i = 1; i <= 3; i++) {

co\_yield i; // yield suspends

}

}

Though complex, coroutines enable async/await-like programming in modern C++.

### 51. Concepts and Constraints (C++20)

Concepts allow restricting template parameters.

#include <concepts>

template <std::integral T>

T add(T a, T b) {

return a + b;

}

int main() {

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

// std::cout << add(2.5, 3.1); // ERROR: not integral

}

### 52. New Features in C++23

* **std::expected** → safer alternative to exceptions.

#include <expected>

std::expected<int, std::string> divide(int a, int b) {

if (b == 0) return std::unexpected("Division by zero");

return a / b;

}

* **std::ranges updates** → cleaner algorithms with range pipelines.

#include <ranges>

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

for (int n : v | std::views::filter([](int x){ return x % 2 == 0; })) {

std::cout << n << " ";

}

* **Multidimensional std::mdspan** → flexible way to handle multi-dimensional arrays.

# C++23 Learning Roadmap – Complete Summary

This roadmap organizes the topics we covered into a **progressive learning path**, guiding you from beginner to advanced mastery.

## Stage 1 – Beginner (Foundations)

**Goal:** Understand syntax, variables, functions, and control flow.

1. Getting Started with C++
   * What is C++ and why use it
   * Compilation process (preprocessing, compiling, linking)
   * Writing and running "Hello World"
2. Basics
   * Variables and data types (int, double, char, bool, std::string)
   * Input/output with std::cin and std::cout
   * Operators (arithmetic, comparison, logical)
3. Control Flow
   * if, else, switch
   * Loops: for, while, do-while
   * Range-based for
4. Functions
   * Function definitions and prototypes
   * Parameters (by value, reference, pointer)
   * Return types
5. Arrays, Strings, and Headers
   * Raw arrays vs std::array vs std::vector
   * std::string vs C-style strings
   * Organizing code with headers (.h) and source files (.cpp)

## Stage 2 – Intermediate (Core C++ Features)

**Goal:** Learn object-oriented programming and memory management.

1. Object-Oriented Programming
   * Classes vs structs
   * Constructors, destructors
   * Encapsulation (private, protected, public)
   * static, const, and mutable members
2. Inheritance and Polymorphism
   * Base and derived classes
   * Virtual functions and overriding
   * Abstract classes (interfaces)
3. Operator Overloading and this Pointer
   * Overloading +, ==, <<, etc.
   * Using this to access the current object
4. Memory Management
   * Stack vs heap memory
   * Copy constructors and assignment operators
   * Move semantics (std::move)
   * Smart pointers (unique\_ptr, shared\_ptr, weak\_ptr)
   * RAII (Resource Acquisition Is Initialization)

## Stage 3 – Standard Template Library (STL)

**Goal:** Use standard containers and algorithms.

1. Containers

* std::vector, std::array, std::list
* std::map, std::unordered\_map
* std::set, std::unordered\_set

1. Iterators and Algorithms

* Iterating with iterators
* Algorithms: sort, reverse, binary\_search

1. Modern STL Types

* std::optional
* std::variant
* std::any
* union (legacy, low-level)

## Stage 4 – Advanced (Performance and Concurrency)

**Goal:** Write high-performance, multithreaded, and safe applications.

1. Performance Measurement

* <chrono> for benchmarking

1. Concurrency

* std::thread, join, detach
* Synchronization with std::mutex, std::lock\_guard
* std::async, std::future, std::promise
* Thread pools (custom or libraries)

1. Debugging and Error Handling

* Debugging in Visual Studio
* Exceptions (try, catch, throw)
* Assertions (assert)
* Logging best practices

## Stage 5 – Modern C++ (C++11 → C++23)

**Goal:** Master modern language features.

1. Templates and Generic Programming

* Function templates
* Class templates

1. Modern Syntax

* Lambdas and captures
* auto, decltype
* constexpr
* Range-based for loops
* Structured bindings

1. C++20/23 Features

* Concepts and constraints
* Modules (import)
* Coroutines (co\_await, co\_yield)
* Ranges (std::views::filter, transform)
* std::expected (C++23)
* std::mdspan (multi-dimensional array views)

## Stage 6 – Best Practices and Large Projects

**Goal:** Write maintainable, scalable, production-grade C++ code.

1. Best Practices

* Prefer smart pointers over raw pointers
* Use RAII for resource safety
* Use const wherever possible
* Minimize global variables
* Write small, testable functions

1. Large Project Organization

* Organizing with namespaces and modules
* Reducing compile times with forward declarations
* Using CMake for builds
* Unit testing frameworks (e.g., Google Test)

### Final Note

C++ is a large language. You do not need to master every feature at once. The recommended approach is:

1. Master the **basics and OOP** (Stages 1–2).
2. Learn the **STL** (Stage 3) because it saves time and avoids reinventing data structures.
3. Move into **memory management and performance** (Stage 4).
4. Adopt **modern features (C++11/20/23)** (Stage 5).
5. Apply **best practices** when building real-world projects (Stage 6).

Great idea — a **TODO App** is an excellent practice project because it forces you to work with **UI, persistence (storage), and cross-platform deployment**. Let’s break this down into **requirements** and **steps**.

TODO APPLICATION

## 1. Requirements

### Functional Requirements

* Add new tasks
* Edit existing tasks
* Mark tasks as completed / uncompleted
* Delete tasks
* Store tasks persistently (saved between app launches)
* (Optional) Categorize tasks (e.g., Work, Personal)
* (Optional) Due dates and reminders

### Non-Functional Requirements

* Runs on **Windows and Android**
* Clean and simple UI
* Fast and lightweight
* Offline support (no internet needed)

## 2. Technology Choices

Since you want **both Windows and Android**:

1. **C++ Backend** (core logic)
   * Manage tasks in memory (using classes and STL containers).
   * Handle persistence (e.g., save/load from JSON or SQLite).
   * This part is portable.
2. **Cross-platform Framework for UI**  
   Options:
   * **Qt (C++ framework)** → Works on Windows, Android, Linux, macOS.
   * **wxWidgets** → Cross-platform but less popular for mobile.
   * **Flutter (Dart, not C++)** → Great for cross-platform UI, but not C++.
   * **C++ with JNI (Android NDK)** → Direct Android integration, but heavy.

For **C++ developer experience**, **Qt** is the best balance (UI + Android + Windows).

1. **Storage**
   * Use **SQLite** (cross-platform database).
   * Or save tasks in **JSON files** with libraries like nlohmann/json.

## 3. Project Structure

Example using Qt + SQLite:

/todo-app

/src

main.cpp // entry point

Task.h/.cpp // Task class

TaskManager.h/.cpp // Manage list of tasks

Database.h/.cpp // SQLite wrapper

MainWindow.ui // UI file (Qt Designer)

/platform

windows/ // Windows build files

android/ // Android build files

CMakeLists.txt // build system

## 4. Implementation Steps

### Step 1 – Core Logic (C++ only, console version)

* Define a Task class (id, title, isCompleted, dueDate).
* Create TaskManager to handle add, edit, remove, toggle.
* Save/load tasks to JSON or SQLite.

### Step 2 – GUI with Qt

* Create a main window.
* Show a list of tasks (QListWidget / QTableView).
* Add input fields (QLineEdit, QPushButton).
* Connect signals and slots (addTaskButton.clicked -> addTask()).

### Step 3 – Persistence

* On app start → load tasks from database/file.
* On app close → save tasks.

### Step 4 – Android Build

* Configure Qt for Android SDK/NDK.
* Reuse the same codebase, only adjust UI scaling for mobile.

### Step 5 – Extra Features

* Add categories/tags.
* Add notifications (Qt Android extras for alarms).
* Sync tasks with cloud (if you want).

## 5. What You Should Do Now

1. Install tools:
   * **Windows** → Visual Studio + Qt Creator + CMake
   * **Android** → Android Studio + Qt for Android (NDK/SDK setup)
2. Start small:
   * Build a **console TODO app in C++** (no UI).
   * Then add **Qt GUI** on Windows.
   * Finally, compile for **Android**.