ZOLAXTECH

C++
Start to Advanced

From Your First Program to Modern, High-Performance C++

A Beginner-to-Advanced Roadmap for Syntax, Memory, OOP, the STL & Concurrency

---

zolaxtech.com.et

Free Learning Resource

Table of Contents

Welcome

C++ is one of the most powerful and enduring languages in software engineering the language behind operating systems, game engines, browsers, trading systems, and embedded devices where performance and control over memory truly matter. It has a reputation for being difficult, but that reputation mostly comes from skipping the fundamentals. This guide builds you up in order: core syntax first, then memory and pointers, then object-oriented design, then the Standard Template Library, then the modern features that make contemporary C++ dramatically friendlier than the language's old reputation suggests.

By the end, you will understand not just how to write C++ that compiles, but how to write C++ that is safe, efficient, and idiomatic the kind senior engineers expect to see in a real codebase.

> **How to use this guide?**
> Read it in order the first time through each chapter builds on the last, especially Parts One and Three, where memory concepts compound quickly. Type out every code example yourself rather than copy-pasting. Each chapter ends with a short practice exercise; do not skip these.

What you will be able to build by the end

- Command-line programs and utilities with proper input handling and error checking
- Class hierarchies using inheritance, polymorphism, and operator overloading
- Memory-safe applications using smart pointers instead of raw new/delete
- Generic, reusable code using templates and the STL's containers and algorithms
- Concurrent programs that run multiple tasks safely using threads and mutexes
- A properly structured multi-file project built and managed with CMake

Prerequisites

None. This guide assumes no prior programming experience, though any prior exposure to a C-like language (Java, C#, JavaScript) will make the early chapters feel familiar.

PART ONE

C++ Foundations

This part is a self-contained crash course in core C++ syntax the same territory C brought into the world, refined with C++'s stronger type system and richer standard library. Work through it carefully; the rest of the guide builds directly on these chapters.

CHAPTER 1:

Getting Started with C++

Setting up your toolchain and writing your first program

1.1 Installing a Compiler

C++ code is compiled directly to native machine code, so you need a compiler rather than just a runtime. Popular options are GCC (g++), Clang, and MSVC (bundled with Visual Studio on Windows). On Linux or macOS, install one from a terminal:

```
# Debian/Ubuntu
sudo apt install g++

# macOS (Xcode command line tools)
xcode-select --install
```

Confirm your installation:

```
g++ --version
```

For an editor, Visual Studio Code with the C/C++ extension, CLion, or Visual Studio all work well.

1.2 Your First Program

```
#include <iostream>

int main() {
    std::cout << "Hello, ZolaxTech!" << std::endl;
    return 0;
}
```

Save this as hello.cpp, then compile and run it:

```
g++ hello.cpp -o hello
```

```
./hello
```

Breaking it down

- #include <iostream> pulls in the standard input/output library.
- int main () every C++ program's entry point; returning 0 signals success to the operating system.
- std::cout << ... sends output to the console using the stream insertion operator.

✎ Try it yourself

1. Modify the program to print your own name instead of “ZolaxTech”.
2. Add a second std::cout statement that prints a short message on its own line.

CHAPTER 2:

Variables, Data Types & Operators

Storing and manipulating data in C++

2.1 Declaring Variables

C++ is statically typed you declare a variable's type before using it, and that type cannot change.

```
int age = 25;
double price = 19.99;
char grade = 'A';
bool isActive = true;
std::string name = "Selam";
```

2.2 Fundamental Data Types

Type	Typical Size	Example	Used for
int	4 bytes	int count = 10;	Whole numbers
double	8 bytes	double total = 3.75;	Decimal numbers
bool	1 byte	bool isDone = false;	True/false values
char	1 byte	char letter = 'z';	Single characters
long long	8 bytes	long long big = 9000000000LL;	Very large whole numbers

2.3 Operators

```
int sum = 5 + 3;            // arithmetic: + - * / %
bool isEqual = (5 == 5);    // comparison: == != > < >= <=
bool result = (true && false); // logical: && || !
```

Note on constants

Prefer const (or constexpr for compile-time constants) over plain variables whenever a value should not change: const double PI = 3.14159; This documents intent and lets the compiler catch accidental modification.

✎ Try it yourself

1. Declare variables for a product name, price, and quantity, then calculate and print the total cost.
2. Write an expression that checks whether a number is even using the % operator.

CHAPTER 3

Control Flow

Making decisions and repeating actions

3.1 Conditional Statements

```
int score = 82;

if (score >= 90) {
    std::cout << "Grade: A" << std::endl;
} else if (score >= 80) {
    std::cout << "Grade: B" << std::endl;
} else {
    std::cout << "Grade: C or below" << std::endl;
}
```

3.2 Switch Statements

```
int day = 3;
switch (day) {
    case 1: std::cout << "Monday"; break;
```

```
    case 2: std::cout << "Tuesday"; break;
    case 3: std::cout << "Wednesday"; break;
    default: std::cout << "Another day";
}
```

3.3 Loops

```
// for loop known number of repetitions
for (int i = 0; i < 5; i++) {
    std::cout << "Count: " << i << std::endl;
}

// while loop repeats while a condition is true
int n = 0;
while (n < 3) {
    std::cout << "n is " << n << std::endl;
    n++;
}

// range-based for iterate a container directly (C++11+)
int values [] = {10, 20, 30};
for (int v : values) {
    std::cout << v << std::endl;
}
```

✎ Try it yourself

1. Write a loop that prints all even numbers between 1 and 20.
2. Write a program that uses if/else to categorize a temperature as Cold, Mild, or Hot.

CHAPTER 4

Functions

Organizing code into reusable pieces

4.1 Declaring and Calling Functions

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

int main() {
    int result = add(3, 4);
    std::cout << result << std::endl; // 7
    return 0;
}
```

4.2 Default Arguments & Overloading

```
double calculateArea(double width, double height = 1.0) {
    return width * height;
}

int add(int a, int b) { return a + b; }
double add(double a, double b) { return a + b; } // overload
```

4.3 Passing by Value vs. by Reference

By default, arguments are copied (pass by value). Passing by reference avoids the copy and lets a function modify the caller's variable directly.

```
void increment (int value) {       // pass by value  caller's variable unaffected
    value++;
}

void incrementRef(int& value) {    // pass by reference  modifies the original
    value++;
}

int x = 5;
increment(x);     // x is still 5
incrementRef(x);  // x is now 6
```

Best practice

Pass small types (int, double, bool) by value. Pass large objects (std::string, std::vector, custom classes) by const reference (const std::string& name) when you only need to read them this avoids an expensive copy.

✎ Try it yourself

1. Write a function isPrime(int n) that returns true if n is prime.
2. Write a function swapValues(int& a, int& b) that swaps two integers using references.

CHAPTER 5

Arrays & Strings

Working with sequences of data

5.1 C-Style Arrays

```
int numbers [5] = {10, 20, 30, 40, 50};
std::cout << numbers[2] << std::endl; // 30
std::cout << "Size: " << sizeof(numbers) / sizeof(numbers[0]) << std::endl;
```

A safer alternative

Modern C++ code generally prefers std::array (fixed size) or std::vector (dynamic size) over raw C-style arrays both are covered in Chapter 12 and offer bounds-aware, safer behavior.

5.2 std::string

```
#include <string>

std::string name = "Selam";
std::string greeting = "Hello, " + name + "!";
std::cout << greeting << std::endl;
std::cout << "Length: " << name.length() << std::endl;
std::cout << "Upper first letter: " << name[0] << std::endl;
```

5.3 Multidimensional Arrays

```
int grid[2][3] = {
    {1, 2, 3},
    {4, 5, 6}
};
```

```
std::cout << grid[1][2] << std::endl; // 6
```

✎ Try it yourself

1. Write a program that stores five student names in an array of std::string and prints them all.
2. Write a function that takes a std::string and returns true if it is a palindrome.

CHAPTER 6

Pointers & References

Understanding memory addresses C++'s defining feature

Pointers are what set C++ apart from most other languages you may have used and they are also where beginners get stuck. Take this chapter slowly; it underpins everything from Chapter 11 onward.

6.1 What Is a Pointer?

A pointer is a variable that stores a memory address rather than a value directly.

```
int age = 25;
int* agePtr = &age; // agePtr holds the address of age

std::cout << age << std::endl;      // 25  the value
std::cout << &age << std::endl;     // e.g. 0x7ffee...  the address
std::cout << agePtr << std::endl;   // same address
std::cout << *agePtr << std::endl;  // 25  dereferencing: "the value at this address"
```

6.2 Pointers vs. References

	Pointer	Reference
Can be reassigned	Yes	No bound at creation
Can be null	Yes (nullptr)	No, must refer to a valid object
Syntax to declare	int* p = &x;	int& r = x;
Typical use	Optional or reassignable links, dynamic memory	Function parameters, aliasing a value safely

6.3 A Common Pitfall: Dangling Pointers

```
int* dangerous() {
    int local = 42;
    return &local;   // BUG: local is destroyed when the function returns
}
```

Rule of thumb

Never return the address of a local (stack) variable from a function. Once the function ends, that memory is no longer valid the pointer becomes "dangling." Chapter 11 introduces smart pointers, which prevent most memory bugs like this one entirely.

✎ Try it yourself

1. Declare an int, a pointer to it, and print both the value and the address using the pointer.
2. Write a function that takes two int pointers and swaps the values they point to.

PART TWO

Object-Oriented C++

C++ was built on top of C specifically to add object-oriented programming. This part covers classes, inheritance, polymorphism, and operator overloading the tools you'll use to model real-world problems cleanly.

CHAPTER 7

Classes & Objects

Defining your own types

7.1 A Basic Class

```
class Car {
public:
    std::string model;
    int year;
```

```
    void displayInfo() {
        std::cout << year << " " << model << std::endl;
    }
};

int main() {
    Car myCar;
    myCar.model = "Toyota Corolla";
    myCar.year = 2022;
    myCar.displayInfo();
}
```

7.2 Access Specifiers

Specifier	Meaning
public	Accessible from anywhere the object is visible
private	Accessible only from within the class itself (the default)
protected	Accessible within the class and any derived classes

7.3 Encapsulation with Getters and Setters

```
class Account {
private:
    double balance;

public:
    double getBalance() const { return balance; }
    void deposit(double amount) {
        if (amount > 0) balance += amount;
    }
};
```

The const keyword on methods

A method marked const, like getBalance() const above, promises it will not modify the object's data. Mark every method that only reads data as const it documents intent and lets the compiler catch mistakes.

✎ Try it yourself

1. Create a Rectangle class with private width and height, and public methods getArea() and getPerimeter().
2. Create a Student class with a private grade list and a method to compute the average.

CHAPTER 8

Constructors, Destructors & the Rule of Three/Five

Controlling object creation and cleanup

8.1 Constructors

```
class Car {
public:
    std::string model;
    int year;

    Car(std::string model, int year) {    // constructor
        this->model = model;
        this->year = year;
    }
};

Car myCar("Toyota Corolla", 2022);   // no 'new' needed for a stack object
```

8.2 Destructors

A destructor runs automatically when an object goes out of scope, and is the natural place to release any resources the object owns.

```
class FileHandle {
public:
    FileHandle() { std::cout << "Opening file" << std::endl; }
    ~FileHandle() { std::cout << "Closing file" << std::endl; } // destructor
};
```

8.3 The Rule of Three (and Five)

If your class manages a resource directly (like raw memory), and you need to define any one of the copy constructor, copy assignment operator, or destructor, you almost always need to define all three the classic Rule of Three. Modern C++ adds move semantics, extending it to the Rule of Five.

Special member	Purpose
Destructor	Cleans up resources when an object is destroyed
Copy constructor	Defines what happens when an object is copied: Car b = a;
Copy assignment operator	Defines what happens on b = a; for existing objects
Move constructor (C++11+)	Efficiently transfers resources from a temporary object
Move assignment operator (C++11+)	Efficiently transfers resources on assignment from a temporary

The modern escape hatch

In practice, the best way to satisfy the Rule of Five is to avoid needing it: use smart pointers and standard containers (Chapters 11–12) instead of raw owned resources, and let the compiler generate correct special members for you automatically.

✎ Try it yourself

1. Add a constructor and destructor to a Logger class that print a message on creation and destruction.
2. Explain in your own words (as a code comment) why copying a class that owns a raw pointer is dangerous without a proper copy constructor.

CHAPTER 9
Inheritance & Polymorphism

Building class hierarchies

9.1 Inheritance

```
class Animal {
public:
   void eat() { std::cout << "Eating..." << std::endl; }
};

class Dog : public Animal {
public:
   void bark() { std::cout << "Woof!" << std::endl; }
};

Dog myDog;
myDog.eat();   // inherited from Animal
myDog.bark();  // defined in Dog
```

9.2 Virtual Functions & Polymorphism

Marking a base class method virtual allows a derived class to override it, and lets code working through a base-class pointer or reference automatically call the correct derived version.

```
class Shape {
public:
   virtual double area() const { return 0; }
   virtual ~Shape() {} // always give a base class a virtual destructor
};

class Circle : public Shape {
   double radius;
public:
   Circle(double r) : radius(r) {}
   double area() const override { return 3.14159 * radius * radius; }
};

Shape* shape = new Circle(5.0);
std::cout << shape->area() << std::endl; // calls Circle's version
delete shape;
```

Why the virtual destructor matters?

Without a virtual destructor on Shape, deleting a Circle through a Shape* only runs Shape's destructor, leaking any resources Circle owns. Always give a virtual destructor to any class meant to be used polymorphically.

9.3 Abstract Classes

```
class Payable {
public:
   virtual double calculatePayment() const = 0; // pure virtual  no implementation
   virtual ~Payable() {}
};

class Employee : public Payable {
   double hours, rate;
public:
   Employee(double h, double r) : hours(h), rate(r) {}
   double calculatePayment() const override { return hours * rate; }
};
```

✎ Try it yourself

1. Create a Shape hierarchy with Circle, Rectangle, and Triangle, each overriding a virtual area () method.
2. Store several Shape* objects in a std::vector<Shape*> and print the total area of all of them.

CHAPTER 10

Operator Overloading

Making your own types behave like built-in ones

C++ lets you redefine what operators like +, ==, and << mean for your own classes, so objects of your types can be used as naturally as ints or doubles.

```
class Vector2D {
public:
   double x, y;
   Vector2D(double x, double y) : x(x), y(y) {}
```

```
    Vector2D operator+(const Vector2D& other) const {
        return Vector2D(x + other.x, y + other.y);
    }

    bool operator==(const Vector2D& other) const {
        return x == other.x && y == other.y;
    }
};

// Overload << so std::cout can print a Vector2D directly
std::ostream& operator<<(std::ostream& os, const Vector2D& v) {
    os << "(" << v.x << ", " << v.y << ")";
    return os;
}

Vector2D a(1, 2), b(3, 4);
Vector2D c = a + b;      // uses operator+
std::cout << c << std::endl; // uses operator<<, prints (4, 6)
```

Guideline

Only overload an operator when its meaning stays intuitive + for combining two Vector2D values makes sense; + to mean “merge two Employee records” likely does not. When in doubt, use a clearly named method instead.

✎ Try it yourself

1. Add operator- and operator* (scalar multiplication) to the Vector2D class above.
2. Overload operator< for a Product class so a std::vector<Product> can be sorted by price.

PART THREE

Memory & Modern C++

This part covers what truly separates C++ from most other languages: precise control over memory, and the modern language features (from C++11 onward) that make that control dramatically safer and more convenient than it used to be.

CHAPTER 11

Dynamic Memory & Smart Pointers

11.1 Manual Memory: new and delete

```
int* value = new int(42);     // allocate on the heap
std::cout << *value << std::endl;
delete value;                 // must free manually, or this leaks

Car* myCar = new Car("Civic", 2021);
myCar->displayInfo();
delete myCar;
```

The core problem

Forget a single delete, and that memory leaks for the life of the program. Delete twice, or use a pointer after deleting it, and the program has undefined behavior often a crash. This is precisely the class of bug smart pointers were designed to eliminate.

11.2 std::unique_ptr sole ownership

```
#include <memory>

std::unique_ptr<Car> myCar = std::make_unique<Car>("Civic", 2021);
myCar->displayInfo();
// no delete needed  automatically freed when myCar goes out of scope
```

11.3 std::shared_ptr shared ownership

```
std::shared_ptr<Car> a = std::make_shared<Car>("Civic", 2021);
std::shared_ptr<Car> b = a;   // both now share ownership
std::cout << a.use_count() << std::endl; // 2
// freed automatically once the LAST shared_ptr to it is gone
```

Smart pointer	Ownership model	Use when…
std::unique_ptr	Exactly one owner	You need exclusive ownership (the default choice)
std::shared_ptr	Multiple shared owners	Several parts of your code must share ownership
std::weak_ptr	Non-owning observer	You need to reference a shared_ptr without extending its life

Modern best practice

In modern C++, raw new and delete are rarely written directly in application code. Reach for std::make_unique or std::make_shared by default, and treat manual new/delete as a sign something needs to be modernized.

✎ Try it yourself

1. Rewrite a class that currently uses new/delete internally to use a std::unique_ptr member instead.
2. Create two std::shared_ptr instances pointing to the same object and print use_count() as each goes out of scope.

CHAPTER 12

The Standard Template Library

Containers and algorithms that do the heavy lifting

The STL is one of C++'s greatest strengths: a large, battle-tested library of generic containers and algorithms that eliminates the need to hand-write data structures for most everyday tasks.

12.1 std::vector a dynamic array

```
#include <vector>

std::vector<std::string> tasks;
tasks.push_back("Design the schema");
tasks.push_back("Write the parser");
tasks.pop_back();

for (const auto& task : tasks) {
    std::cout << task << std::endl;
}
```

12.2 std::map key-value pairs

```
#include <map>

std::map<std::string, double> prices;
prices["Keyboard"] = 45.00;
prices["Mouse"] = 20.00;
```

```
std::cout << prices["Keyboard"] << std::endl; // 45
```

12.3 Common Containers at a Glance

Container	Description
std::vector	Dynamic array; fast random access, grows automatically
std::array	Fixed-size array with STL-style interface
std::map	Sorted key-value pairs (red-black tree, O(log n) lookup)
std::unordered_map	Hash-table key-value pairs (average O(1) lookup)
std::set	Sorted collection of unique values
std::list	Doubly linked list; fast insert/remove anywhere

12.4 Algorithms

```
#include <algorithm>

std::vector<int> nums = {5, 3, 8, 1, 9};
std::sort(nums.begin(), nums.end());

bool hasEight = std::find(nums.begin(), nums.end(), 8) != nums.end();
int total = std::accumulate(nums.begin(), nums.end(), 0);
```

Guideline

Reach for a standard container and algorithm before hand-writing your own. std::sort alone is a highly tuned introsort implementation a hand-rolled bubble sort will rarely, if ever, beat it.

✎ Try it yourself

1. Build a std::vector<int> of ten numbers, sort it, then print the minimum and maximum using std::min_element / std::max_element.
2. Build a std::map<std::string, int> word-frequency counter over a short sentence split into words.

CHAPTER 13
Templates & Generic Programming

Writing code that works with any type

13.1 Function Templates

```
template <typename T>
T maxOf(T a, T b) {
   return (a > b) ? a : b;
}

std::cout << maxOf(3, 7) << std::endl;         // T = int
std::cout << maxOf(3.5, 2.1) << std::endl;     // T = double
std::cout << maxOf(std::string("a"), std::string("b")) << std::endl;
```

13.2 Class Templates

```
template <typename T>
class Box {
   T contents;
public:
   Box(T value) : contents(value) {}
   T getContents() const { return contents; }
};

Box<int> intBox(42);
Box<std::string> stringBox("hello");
```

What the STL is built on

Every container you learned in Chapter 12 std::vector<T>, std::map<K, V> is itself a class template. Once you understand templates, you understand exactly what powers the entire STL underneath.

✎ Try it yourself

1. Write a template function swapValues<T>(T& a, T& b) that works for any type.
2. Write a simple Pair<T1, T2> class template holding two values of possibly different types.

CHAPTER 14

Modern C++ Features

What C++11 through C++20 added

14.1 auto type inference

```
auto count = 10;              // inferred as int
auto price = 19.99;           // inferred as double
auto name = std::string("Selam");

std::vector<int> nums = {1, 2, 3};
for (auto& n : nums) { n *= 2; } // much cleaner than spelling out the iterator type
```

14.2 Lambda Expressions

```
auto add = [](int a, int b) { return a + b; };
std::cout << add(3, 4) << std::endl; // 7

// Lambdas are commonly passed straight into algorithms:
std::vector<int> nums = {5, 3, 8, 1, 9};
std::sort(nums.begin(), nums.end(), [](int a, int b) { return a > b; }); // descending
```

14.3 Move Semantics (C++11)

Move semantics let a resource be transferred from a temporary object instead of copied, which is why standard containers became dramatically faster in C++11 without any change to your own code.

```
std::vector<std::string> makeNames() {
    std::vector<std::string> names = {"Abel", "Selam", "Kebede"};
    return names; // moved out, not copied  the compiler does this automatically
}
```

14.4 Structured Bindings (C++17)

```
std::map<std::string, int> ages = {{"Selam", 30}, {"Abel", 25}};
for (const auto& [name, age] : ages) {
    std::cout << name << " is " << age << std::endl;
}
```

Feature	Introduced in	Why it matters
auto	C++11	Reduces verbosity, especially with iterators and templates

Lambdas	C++11	Inline, anonymous functions essential for algorithms
Move semantics	C++11	Avoids unnecessary deep copies of large objects
Structured bindings	C++17	Unpacks pairs/tuples/structs cleanly
Concepts	C++20	Readable, checked constraints on template parameters

✎ Try it yourself

1. Rewrite a for-loop you wrote earlier in this guide using auto and a range-based for loop.
2. Write a lambda that filters a std::vector<int> to only even numbers using std::copy_if.

PART FOUR

Advanced C++

This final part covers the professional skills needed to ship real C++ software: handling failures gracefully, reading and writing files, running work concurrently, structuring a multi-file project, and testing what you build.

```
#include <stdexcept>

double divide(double a, double b) {
    if (b == 0) {
        throw std::invalid_argument("Division by zero");
    }
    return a / b;
}

try {
    double result = divide(10, 0);
} catch (const std::invalid_argument& e) {
    std::cout << "Error: " << e.what() << std::endl;
} catch (const std::exception& e) {
    std::cout << "Unexpected error: " << e.what() << std::endl;
}
```

> **RAII: C++'s answer to cleanup**
> Because destructors run automatically even when an exception unwinds the stack, wrapping a resource in a class (as std::unique_ptr and std::vector already do for you) guarantees clean-up happens no matter how a function exits. This pattern is called RAII Resource Acquisition Is Initialization and it is the idiomatic C++ alternative to manual clean-up code.

CHAPTER 15

Exception Handling

Writing code that fails gracefully

CHAPTER 16

File I/O

Reading and writing data on disk

16.1 Writing a File

```
#include <fstream>

std::ofstream outFile("notes.txt");
if (outFile.is_open()) {
    outFile << "Meeting at 10am" << std::endl;
    outFile << "Buy office supplies" << std::endl;
    outFile.close();
}
```

16.2 Reading a File

```
#include <fstream>
#include <string>

std::ifstream inFile("notes.txt");
std::string line;
while (std::getline(inFile, line)) {
    std::cout << line << std::endl;
}
inFile.close();
```

RAII in practice

std::ofstream and std::ifstream automatically close their file when they go out of scope, even without an explicit .close() call another everyday example of RAII.

CHAPTER 17

Multithreading & Concurrency

Running work in parallel safely

17.1 Launching a Thread

```
#include <thread>

void printNumbers() {
    for (int i = 1; i <= 5; i++) {
        std::cout << i << std::endl;
    }
}

int main() {
    std::thread t(printNumbers);
    t.join(); // wait for the thread to finish
    return 0;
}
```

17.2 Protecting Shared Data with a Mutex

When two threads read and write the same variable, you get a data race undefined, unpredictable behavior. A std::mutex ensures only one thread touches the shared data at a time.

```
#include <mutex>

int counter = 0;
std::mutex counterMutex;

void increment() {
    for (int i = 0; i < 1000; i++) {
        std::lock_guard<std::mutex> lock(counterMutex); // locks for this scope
        counter++;
    }
```

```
}

std::thread t1(increment);
std::thread t2(increment);
t1.join();
t2.join();
std::cout << counter << std::endl; // reliably 2000
```

Why std::lock_guard

std::lock_guard uses RAII to release the mutex automatically at the end of its scope even if an exception is thrown so you never forget to unlock it. Prefer it over calling mutex.lock() / mutex.unlock() manually.

✎ Try it yourself

1. Launch two threads that each print their own thread id ten times, and observe the interleaving.
2. Add a std::mutex to protect a shared std::vector that both threads push_back into.

CHAPTER 18

Build Systems with CMake

Managing real, multi-file projects

Real C++ projects span many .cpp and .h files and often depend on external libraries. CMake is the de facto standard build-system generator used across the C++ industry to manage this.

18.1 A Minimal CMakeLists.txt

```
cmake_minimum_required(VERSION 3.16)
project(ZolaxApp)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_executable(zolax_app src/main.cpp src/car.cpp)
```

18.2 Building the Project

```
mkdir build && cd build
cmake ..
cmake --build .
```

```
./zolax_app
```

18.3 Typical Project Layout

Path	Purpose
src/	Your .cpp implementation files
include/	Your .h / .hpp header files
CMakeLists.txt	Build configuration
build/	Generated build output (kept out of version control)
tests/	Unit test source files

CHAPTER 19

Debugging, Testing & Best Practices

Writing dependable C++

19.1 Debugging with gdb

```
g++ -g main.cpp -o main   # -g keeps debug symbols
gdb ./main
(gdb) break main
(gdb) run
(gdb) next
```

19.2 Unit Testing with Catch2

```
#include <catch2/catch_test_macros.hpp>

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

TEST_CASE("addition works") {
    REQUIRE(add(2, 2) == 4);
}
```

19.3 Practical Best Practices

- Compile with warnings enabled (-Wall -Wextra) and treat warnings as bugs waiting to happen

- Prefer smart pointers and STL containers over raw new/delete and C-style arrays
- Mark every method that doesn't modify state as const
- Run a sanitizer (-fsanitize=address) during development to catch memory bugs early
- Keep functions short and single-purpose it makes unit testing dramatically easier

APPENDIX

Where to Go Next

Suggested Learning Path

1. Rebuild every code example in this guide by hand and compile it yourself.
2. Build three small practice projects: a command-line to-do list, a simple bank-account simulator, and a text-based inventory system using the STL.
3. Learn a debugger and a sanitizer thoroughly they will save you more time than any other single skill.
4. Study the STL in more depth: iterators, custom comparators, and the <algorithm> header.
5. Read (or contribute to) an open-source C++ project on GitHub to see idiomatic style in practice.
6. Package a finished console or GUI project with CMake and share it with a friend to try building.

Quick Reference: C++ Concept → Where You'll Use It

C++ Concept	Where You'll Use It
Pointers & References	Function parameters, dynamic memory, linked structures
Classes & Inheritance	Every custom type and class hierarchy you design
Smart Pointers	Any owned, dynamically allocated resource
Templates	Generic functions/classes; everything the STL is built on
STL Containers/Algorithms	Everyday data storage, searching, and sorting
Threads & Mutexes	Any task that should run concurrently with the rest of your app

Thank you for learning with ZolaxTech

More free resources at zolaxtech.com.et