**Advanced C++ Training Program**

Contents

1. Introduction to C++11 and Beyond

2. Dynamic Memory Management

3. STL Associative Containers & Algorithms

4. Design Patterns and Best Practices

5. Concurrency and Parallelism

7. Optimizing Code Performance

8. Advanced Language Features

**1. Introduction to C++11 and Beyond**

Evolution of C++: from C++98 to C++20  
Modern syntax and language improvements  
Smart pointers, auto, nullptr, range-based for, enum class  
Lambda expressions and move semantics  
Rationale behind changes and their real-world impact

**2. Dynamic Memory Management**

Heap vs Stack allocation  
Manual memory management pitfalls  
RAII (Resource Acquisition Is Initialization)  
Smart pointers (std::unique*ptr, std::shared*ptr, std::weak\_ptr)  
Custom deleters and ownership semantics

**3. STL Associative Containers & Algorithms**

Overview of STL architecture  
Associative containers: map, unordered*map, set, multimap, unordered*set  
Iterators, predicates, function objects  
Using std::algorithm: find\_if, sort, transform, accumulate, etc.  
Performance trade-offs and container selection guidelines

**4. Design Patterns and Best Practices**

Core OO patterns in C++: Strategy, Observer, Factory, Singleton, Visitor  
SOLID principles applied in modern C++  
Dependency injection, policy-based design, interface segregation  
Avoiding anti-patterns in C++  
Industry Best Practices: Writing clean and maintainable code, adhering to coding standards.  
Exception Handling: Best practices for using exceptions and error handling in C++.

**5. Concurrency and Parallelism**

Thread management with std::thread  
Mutexes, condition variables, locks (std::mutex, std::lock*guard, std::unique*lock)  
Thread-safe data sharing  
Task-based concurrency: std::async, std::future, std::promise

**6. Templates and Metaprogramming**

Template basics: function, class, and variadic templates  
SFINAE, decltype, auto, and type traits - ***Substitution Failure Is Not An Error***  
constexpr and compile-time computation  
Modern alternatives: Concepts, type deduction

**7. Optimizing Code Performance**

Identifying performance bottlenecks  
Value vs reference semantics, copy elision, move semantics  
Inlining, loop unrolling, memory alignment  
Cache-awareness and memory access patterns  
Compiler optimization flags and profiling

**8. Advanced Language Features**

Rvalue references and move semantics in depth  
decltype, noexcept, override, final, explicit  
Lambda captures and closures  
Coroutines  
Modules  
Structured bindings and deconstruction

**9. Linking and Binary Structure**

Lib vs DLL: Differences and use-cases  
Compiling/Loading/Linking Static Libraries (LIB)  
Compiling/Loading/Linking Dynamic Libraries (DLL)  
Best practices for cross-platform binary compatibility

Also, following are the list of tools we use –

* C++ 20
* VS 2022
* Cmake + Ninja
* Conan and Nuget
* Win 10/11
* Alma Linux
* GTest and GMock
* ReSharper, TICS, SonarQube

**Introduction to C++11 and Beyond**

Evolution of C++: from C++98 to C++20  
Modern syntax and language improvements  
Smart pointers, auto, nullptr, range-based for, enum class  
Lambda expressions and move semantics  
Rationale behind changes and their real-world impact

**Introduction to C++11 and Beyond**

C++ has evolved significantly since its original standardization in **C++98**. The **C++11 standard** marked a turning point, often called **“Modern C++”**, as it introduced features that made the language safer, faster, and easier to use. Later standards (C++14, C++17, C++20, C++23) built upon this, making C++ a more expressive and efficient language.

**🔹 Why C++11 Was a Big Deal**

Before C++11:

* C++98/03 was powerful but verbose, unsafe in places, and lacked modern conveniences.
* Programmers had to rely on manual memory management and workarounds.

With C++11:

* The language became safer (smart pointers, nullptr).
* More efficient (move semantics).
* Easier to write (auto, range-based for loops, lambdas).
* More expressive (constexpr, uniform initialization).

**🔹 Key Features in C++11**

1. **auto keyword** → Type deduction for variables.
2. auto x = 10; // int
3. auto y = 3.14; // double
4. **nullptr** → Type-safe null pointer.
5. int\* p = nullptr;
6. **Range-based for loop**
7. for (auto val : {1, 2, 3, 4}) {
8. std::cout << val << " ";
9. }
10. **Lambda expressions** (inline anonymous functions).
11. auto add = [](int a, int b) { return a + b; };
12. std::cout << add(3, 4);
13. **Move semantics & rvalue references** → Optimize performance by moving instead of copying.
14. std::string s1 = "Hello";
15. std::string s2 = std::move(s1); // ownership transferred
16. **Smart pointers** (std::unique\_ptr, std::shared\_ptr).
17. auto ptr = std::make\_unique<int>(42);
18. **Uniform initialization**
19. std::vector<int> v{1, 2, 3, 4};
20. **constexpr** → Compile-time constants.
21. constexpr int square(int x) { return x \* x; }

**🔹 Beyond C++11**

Each standard after C++11 polished and extended features:

**✅ C++14**

* Generic lambdas (auto in lambda params).
* Return type deduction for functions.
* std::make\_unique.

**✅ C++17**

* Structured bindings.
* auto [a, b] = std::pair(1, 2);
* if constexpr for compile-time branching.
* std::optional, std::variant, std::any.
* Filesystem library (std::filesystem).

**✅ C++20**

* **Concepts** (better templates).
* **Ranges** library.
* **Coroutines** (co\_await, co\_yield).
* **Modules** (replaces headers).
* std::span, std::format.

**✅ C++23**

* std::expected for error handling.
* Range improvements (views::join\_with, etc.).
* More constexpr support.
* Refinements to modules and standard library.

**🔹 Why Learn Modern C++?**

* Cleaner and safer code.
* Performance without sacrificing readability.
* Widely used in **system programming, game engines, finance, AI/ML frameworks, and embedded systems**.
* Interviews and real-world projects expect C++11+ knowledge.

**Summary**:  
C++11 was the **foundation of Modern C++**, introducing auto, lambdas, move semantics, smart pointers, and more. Later standards (14, 17, 20, 23) refined these ideas, added new abstractions, and made the language more **expressive and efficient**.

**Evolution of C++: From C++98 to C++20**

C++ has been standardized in multiple versions. Each version introduced new features to make C++ **safer, faster, and easier to use** while retaining its performance and low-level control.

**📌 C++98 (First Standardized C++)**

* Finalized in **1998**, it was the first ISO standard for C++.
* Included **templates**, **exceptions**, **STL (Standard Template Library)**, and **RAII (Resource Acquisition Is Initialization)** concepts.

✅ Key features:

* Classes, inheritance, polymorphism
* Operator overloading
* Templates (function & class templates)
* Standard Template Library (vector, map, list, etc.)
* Exception handling

👉 Example:

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> v;

v.push\_back(10);

v.push\_back(20);

for (size\_t i = 0; i < v.size(); ++i)

cout << v[i] << " ";

}

**📌 C++03 (Minor Update)**

* Released in **2003**, mainly bug fixes and clarifications to C++98.
* No major new features, but provided a more stable base.

**📌 C++11 (The Game Changer – “Modern C++”)**

* Released in **2011**, huge leap forward.
* Introduced **move semantics**, **auto**, **lambdas**, **nullptr**, **smart pointers**, and **range-based for loops**.

✅ Key features:

* auto type deduction
* Range-based for loops
* Lambda functions
* Smart pointers (unique\_ptr, shared\_ptr)
* Rvalue references & move semantics
* Uniform initialization {}
* constexpr for compile-time evaluation
* nullptr instead of NULL

👉 Example:

#include <iostream>

#include <vector>

#include <memory>

using namespace std;

int main() {

auto v = vector<int>{1, 2, 3, 4}; // auto + uniform init

for (auto x : v) cout << x << " "; // range-based for

auto ptr = make\_unique<int>(42); // smart pointer

cout << "\nValue: " << \*ptr;

auto add = [](int a, int b) { return a + b; }; // lambda

cout << "\nSum: " << add(3, 5);

}

**📌 C++14 (Polishing C++11)**

* Released in **2014**, mostly small improvements.
* Made C++11 easier to use.

✅ Key features:

* Generic lambdas (auto in lambda params).
* Return type deduction for functions.
* std::make\_unique utility.
* Relaxed constexpr functions.

👉 Example:

auto mul = [](auto a, auto b) { return a \* b; }; // generic lambda

cout << mul(2, 3) << " " << mul(2.5, 3.5);

**📌 C++17 (More Expressive and Safer)**

* Released in **2017**, added features for **structured programming** and **better standard library support**.

✅ Key features:

* Structured bindings (auto [x, y] = ...;)
* if constexpr (compile-time branching)
* std::optional, std::variant, std::any
* std::filesystem for file handling
* Fold expressions for templates

👉 Example:

#include <iostream>

#include <tuple>

#include <optional>

using namespace std;

optional<int> findValue(bool ok) {

if (ok) return 42;

return nullopt;

}

int main() {

auto [a, b] = make\_pair(1, 2); // structured binding

cout << a << ", " << b << "\n";

auto val = findValue(true);

if (val) cout << "Found: " << \*val;

}

**📌 C++20 (Major Leap Forward)**

* Released in **2020**, the biggest update since C++11.
* Introduced **concepts, ranges, coroutines, and modules**.

✅ Key features:

* **Concepts** (better template constraints).
* **Ranges** library (functional-style operations on collections).
* **Coroutines** (co\_await, co\_yield, co\_return).
* **Modules** (faster compilation, replacing some headers).
* std::span, std::format, std::chrono improvements.

👉 Example:

#include <iostream>

#include <vector>

#include <ranges>

using namespace std;

int main() {

vector<int> nums = {1, 2, 3, 4, 5, 6};

// Ranges: filter even numbers and square them

auto result = nums | views::filter([](int n){ return n % 2 == 0; })

| views::transform([](int n){ return n \* n; });

for (int n : result)

cout << n << " "; // Output: 4 16 36

}

**📊 Summary Table: Evolution of C++**

| **Standard** | **Year** | **Highlights** |
| --- | --- | --- |
| **C++98** | 1998 | First ISO standard, templates, STL, exceptions |
| **C++03** | 2003 | Minor bug fixes, refinements |
| **C++11** | 2011 | *Modern C++ begins* → auto, lambdas, smart pointers, move semantics |
| **C++14** | 2014 | Generic lambdas, make\_unique, return type deduction |
| **C++17** | 2017 | Structured bindings, if constexpr, std::optional, filesystem |
| **C++20** | 2020 | Concepts, ranges, coroutines, modules, std::format, std::span |

✅ **In short:**

* **C++98/03** → Classic C++.
* **C++11** → Start of *Modern C++*.
* **C++14** → Cleaner C++11.
* **C++17** → Safer & expressive C++.
* **C++20** → Powerful & futuristic C++.

**Modern Syntax and Language Improvements in C++**

**1. Type Deduction**

**Old (C++98/03)**

std::vector<int>::iterator it = v.begin();

**Modern**

auto it = v.begin(); // C++11

* auto deduces type automatically.
* decltype (C++11) and decltype(auto) (C++14) allow type reflection.

**2. Range-based for loops**

**Old**

for (size\_t i = 0; i < v.size(); ++i) {

std::cout << v[i] << " ";

}

**Modern**

for (auto x : v) { // C++11

std::cout << x << " ";

}

* Cleaner syntax.
* Works with arrays, containers, and ranges.

**3. Smart Pointers**

**Old**

int\* p = new int(42);

delete p; // Must remember!

**Modern**

auto up = std::make\_unique<int>(42); // C++14

auto sp = std::make\_shared<int>(100); // automatic ref-counting

* No manual delete.
* Memory leaks reduced.

**4. Move Semantics & Rvalue References (C++11)**

Improves performance by **moving** instead of **copying**.

std::string s1 = "Hello";

std::string s2 = std::move(s1); // move constructor

* Avoids unnecessary deep copies.
* Crucial for performance in STL containers.

**5. Uniform Initialization (Brace {})**

**Old**

int arr[3] = {1, 2, 3};

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

**Modern**

std::vector<int> v{1, 2, 3}; // C++11 uniform init

* Works consistently across arrays, structs, classes.

**6. Lambda Expressions**

**Old**

Had to define small functions separately.

**Modern**

auto add = [](int a, int b) { return a + b; }; // C++11

std::cout << add(3, 4);

* C++14: generic lambdas ([](auto a, auto b) { return a + b; })
* C++17: lambdas can capture by move
* C++20: lambdas are constexpr by default

**7. constexpr and consteval**

* constexpr (C++11, improved in C++14/17/20): functions evaluated at **compile-time**.
* consteval (C++20): must be evaluated at compile-time.

constexpr int square(int x) { return x \* x; }

static\_assert(square(5) == 25, "Compile-time check!");

**8. Nullptr**

**Old**

int\* p = NULL; // could be confused with 0

**Modern**

int\* p = nullptr; // C++11

* Type-safe, avoids ambiguity.

**9. Structured Bindings (C++17)**

auto [a, b] = std::pair(10, 20);

std::cout << a << ", " << b;

* Cleaner unpacking of tuples, structs, and maps.

**10. if constexpr (C++17)**

Compile-time branching inside templates.

template<typename T>

void printType(T t) {

if constexpr (std::is\_integral\_v<T>)

std::cout << "Integer\n";

else

std::cout << "Not an integer\n";

}

**11. Modules (C++20)**

Replaces header files, reduces compile times, avoids #include mess.

// math.ixx

export module math;

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

// main.cpp

import math;

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

**12. Ranges (C++20)**

Functional style programming with containers.

#include <ranges>

#include <vector>

#include <iostream>

using namespace std;

int main() {

vector<int> v{1,2,3,4,5,6};

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

| ranges::views::transform([](int x){ return x \* x; })) {

cout << n << " "; // Output: 4 16 36

}

}

**13. Coroutines (C++20)**

Enable async-style programming.

#include <coroutine>

#include <iostream>

generator<int> counter() {

for (int i = 0; i < 3; ++i)

co\_yield i; // suspend/resume

}

**14. Other Useful Improvements**

* std::optional, std::variant, std::any (C++17).
* std::string\_view (C++17): lightweight string wrapper.
* std::span (C++20): safer array handling.
* std::format (C++20): modern replacement for printf.

**📊 Summary of Modern Syntax Improvements**

| **Feature** | **Introduced** |
| --- | --- |
| auto, nullptr, range-for, lambdas, move semantics | C++11 |
| Generic lambdas, make\_unique, return type deduction | C++14 |
| Structured bindings, if constexpr, optional/variant, filesystem | C++17 |
| Concepts, ranges, coroutines, modules, span, format | C++20 |

✅ **Modern C++ = concise, safer, more expressive, faster.**  
Where C++98/03 was verbose and tricky, **Modern C++ gives the productivity of Python/Java with C++ performance**.

**Modern C++ Key Features**

**1. Smart Pointers (C++11)**

Before C++11, we had to manage memory manually (new/delete), which was error-prone and caused leaks.  
Smart pointers manage memory automatically.

**Types:**

* **std::unique\_ptr** → owns resource exclusively.
* **std::shared\_ptr** → shared ownership (reference counted).
* **std::weak\_ptr** → non-owning reference, prevents cyclic references.

👉 Example:

#include <iostream>

#include <memory>

using namespace std;

int main() {

auto up = make\_unique<int>(42); // unique\_ptr

cout << "Unique: " << \*up << endl;

auto sp1 = make\_shared<int>(100); // shared\_ptr

auto sp2 = sp1; // share ownership

cout << "Shared: " << \*sp1 << " Count: " << sp1.use\_count() << endl;

}

✅ Benefit: **No manual delete, no leaks**.

**2. auto (C++11)**

Automatically deduces the type of a variable from its initializer.

👉 Example:

auto x = 10; // int

auto y = 3.14; // double

auto s = "Hello"; // const char\*

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

for (auto it = v.begin(); it != v.end(); ++it) {

cout << \*it << " ";

}

✅ Benefit: **Cleaner code**, especially with long STL types.

**3. nullptr (C++11)**

Replaces NULL (which was just 0 and could cause ambiguity).

👉 Example:

void func(int x) { cout << "int\n"; }

void func(int\* p) { cout << "pointer\n"; }

int main() {

func(0); // calls int version (confusing with NULL!)

func(nullptr); // calls pointer version (clear!)

}

✅ Benefit: **Type-safe null pointer**.

**4. Range-based for loop (C++11)**

Cleaner iteration over arrays/containers.

👉 Example:

#include <vector>

#include <iostream>

using namespace std;

int main() {

vector<int> v{1, 2, 3, 4, 5};

// Old style

for (size\_t i = 0; i < v.size(); ++i)

cout << v[i] << " ";

cout << endl;

// Modern style

for (auto x : v) // range-based for

cout << x << " ";

}

✅ Benefit: **Concise & safer loop syntax**.

**5. enum class (C++11)**

Traditional enums were not type-safe (could implicitly convert to int).  
enum class is strongly typed and scoped.

👉 Example:

enum class Color { Red, Green, Blue };

int main() {

Color c = Color::Red;

if (c == Color::Red) {

cout << "Red color\n";

}

// int x = Color::Red; // ❌ error (unlike old enums)

int y = static\_cast<int>(Color::Green); // ✅ explicit conversion

}

✅ Benefit: **Strong typing**, avoids name clashes.

**📊 Quick Comparison Table**

| **Feature** | **Old C++ (98/03)** | **Modern C++ (11+)** |
| --- | --- | --- |
| **Smart pointers** | new / delete (manual) | unique\_ptr, shared\_ptr (automatic) |
| **auto** | Must write full type | Compiler deduces type |
| **nullptr** | NULL (same as 0) | Type-safe nullptr |
| **Range-for** | for (int i=0; i<n; i++) | for (auto x : container) |
| **Enums** | Weakly typed, global scope | enum class → strongly typed, scoped |

✅ **In short**:  
These features make Modern C++ code **shorter, safer, and less error-prone** compared to classic C++.

**1. Lambda Expressions (C++11)**

Lambdas are **anonymous functions** you can define inline.  
They are useful for **short tasks**, especially in algorithms, callbacks, and event handling.

**General Syntax:**

[capture\_list](parameters) -> return\_type {

// function body

}

**Example 1: Basic Lambda**

#include <iostream>

using namespace std;

int main() {

auto add = [](int a, int b) { return a + b; };

cout << "Sum: " << add(3, 5) << endl;

}

✅ Instead of writing a separate function, you define it **inline**.

**Example 2: Capture Variables**

#include <iostream>

using namespace std;

int main() {

int x = 10, y = 20;

auto printSum = [x, y]() { // capture by value

cout << "Sum: " << (x + y) << endl;

};

printSum();

auto increment = [&x]() { // capture by reference

x++;

};

increment();

cout << "x after increment: " << x << endl;

}

✅ Capturing allows lambdas to **use local variables**.

**Example 3: With STL Algorithm**

#include <vector>

#include <algorithm>

#include <iostream>

using namespace std;

int main() {

vector<int> nums = {1, 2, 3, 4, 5};

// Print only even numbers

for\_each(nums.begin(), nums.end(),

[](int n) { if (n % 2 == 0) cout << n << " "; });

}

✅ Great for **short inline predicates**.

**Modern Improvements:**

* **C++14:** Generic lambdas

auto multiply = [](auto a, auto b) { return a \* b; };

cout << multiply(2, 3.5);

* **C++17:** Capture by move
* **C++20:** Lambdas can be constexpr by default

**🔹 2. Move Semantics (C++11)**

C++11 introduced **rvalue references (&&)** and **move semantics**.  
This avoids expensive **deep copies** when temporary objects are used.

**Problem in Old C++**

#include <iostream>

#include <vector>

using namespace std;

vector<int> createVector() {

vector<int> v = {1, 2, 3, 4, 5};

return v; // copy happens here in C++98

}

int main() {

vector<int> data = createVector(); // costly copy

}

✅ In **C++98/03**, return caused a **deep copy**.

**Move Semantics in C++11**

#include <iostream>

#include <vector>

using namespace std;

vector<int> createVector() {

vector<int> v = {1, 2, 3, 4, 5};

return v; // C++11: uses move constructor

}

int main() {

vector<int> data = createVector(); // moved, not copied

}

✅ Compiler automatically uses **move constructor** if available.

**Explicit std::move**

#include <iostream>

#include <string>

using namespace std;

int main() {

string s1 = "Hello";

string s2 = std::move(s1); // s1 moved into s2

cout << "s2: " << s2 << endl; // "Hello"

cout << "s1: " << s1 << endl; // empty/undefined

}

* std::move casts an lvalue into an rvalue, enabling move semantics.
* After move, the **source object is valid but unspecified** (often empty).

**Move Constructor & Move Assignment**

class MyClass {

int\* data;

int size;

public:

MyClass(int s) : size(s), data(new int[s]) {}

// Move constructor

MyClass(MyClass&& other) noexcept : data(other.data), size(other.size) {

other.data = nullptr;

other.size = 0;

}

// Move assignment

MyClass& operator=(MyClass&& other) noexcept {

if (this != &other) {

delete[] data;

data = other.data;

size = other.size;

other.data = nullptr;

other.size = 0;

}

return \*this;

}

~MyClass() { delete[] data; }

};

✅ Enables **efficient transfer of resources** instead of deep copy.

**📊 Quick Summary**

| **Feature** | **Purpose** | **Example** |
| --- | --- | --- |
| **Lambda expressions** | Inline anonymous functions | [](int x){ return x\*x; } |
| **Move semantics** | Avoid unnecessary copies, transfer ownership | string s2 = std::move(s1); |

👉 **In short:**

* **Lambdas** = write shorter, functional-style code (great with STL).
* **Move semantics** = massive performance boost when working with **large objects, vectors, strings**.

**Rationale Behind Modern C++ Changes and Their Real-World Impact**

**🔹 1. Smart Pointers**

**Rationale**

* In old C++ (98/03), memory had to be managed with new/delete.
* This led to **memory leaks**, **dangling pointers**, and **double deletes**.
* Developers often reinvented RAII wrappers in every project.

**Real-world Impact**

* Smart pointers (unique\_ptr, shared\_ptr, weak\_ptr) standardize memory safety.
* **Automatic cleanup** → fewer leaks.
* Encouraged use of **RAII**, which is critical in large projects like browsers, games, OS kernels.

💡 Example:  
In **Google Chrome** and **Mozilla Firefox**, smart pointers are heavily used to manage DOM nodes and GPU buffers without risking leaks.

**🔹 2. auto**

**Rationale**

* C++98/03 required **verbose types**, especially with STL iterators.
* Type repetition made code harder to maintain.

**Real-world Impact**

* auto improves **readability and maintainability**.
* Reduces **boilerplate code**, especially in templates and STL-heavy codebases.
* Encourages **generic programming**.

💡 Example:  
In **machine learning frameworks (like TensorFlow C++)**, auto is used to iterate over complex template types without 3-line-long type names.

**🔹 3. nullptr**

**Rationale**

* Old NULL was just 0, causing **ambiguous overloads**.
* Safety issue: NULL could accidentally be treated as an integer.

**Real-world Impact**

* nullptr is **type-safe**, preventing subtle bugs.
* Makes APIs clearer when a pointer vs. integer is expected.

💡 Example:  
In **game engines** (like Unreal Engine), nullptr prevents crashes due to ambiguous overloads when checking object validity.

**🔹 4. Range-based for**

**Rationale**

* Looping over containers with indices was verbose and error-prone.
* Iterators required extra syntax, especially for beginners.

**Real-world Impact**

* **Cleaner iteration**, similar to Python’s for x in list.
* Reduces **off-by-one errors**.
* Widely adopted in embedded systems, finance, and simulations where clarity and safety matter.

💡 Example:  
In **quantitative finance software**, analysts iterate over millions of trades more clearly with range-for loops, reducing risk of index mistakes.

**🔹 5. enum class**

**Rationale**

* Old enums were **weakly typed** and lived in the global namespace.
* Implicit conversion to int caused hidden bugs.

**Real-world Impact**

* Strongly typed enums prevent invalid comparisons (Color::Red == Shape::Circle ❌).
* Safer in **large codebases** with many enums.
* Improves readability and reduces namespace pollution.

💡 Example:  
In **automotive software**, enum class Gear {Park, Drive, Reverse}; ensures that gear states don’t accidentally mix with unrelated integer values.

**🔹 6. Lambda Expressions**

**Rationale**

* Writing small helper functions was too verbose.
* STL algorithms (std::sort, std::for\_each) required function pointers or functors.

**Real-world Impact**

* Makes code **more functional and concise**.
* Encourages use of **higher-order functions**.
* Reduces boilerplate, especially in **multithreading and event-driven code**.

💡 Example:  
In **GUI frameworks**, lambdas make event handlers compact:

button.onClick([]{ std::cout << "Clicked!"; });

**🔹 7. Move Semantics**

**Rationale**

* C++98 copied everything (expensive for big objects like std::vector, std::string).
* Performance bottleneck in high-performance apps.

**Real-world Impact**

* Move semantics transfer resources instead of copying.
* Huge performance gains in **big data**, **game engines**, **scientific computing**, and **real-time systems**.
* Reduces unnecessary memory allocations.

💡 Example:  
In **video game asset loading**, moving large textures and meshes between containers avoids costly deep copies, making loading screens faster.

**🔹 Broader Impacts of Modern C++ Changes**

* **Safety**: Smart pointers, nullptr, enum class → fewer runtime crashes.
* **Productivity**: auto, range-for, lambdas → less boilerplate, cleaner code.
* **Performance**: Move semantics, constexpr → faster, more efficient.
* **Maintainability**: Strong typing and modern syntax → fewer hidden bugs in large teams.

**📊 Summary Table**

| **Feature** | **Rationale** | **Real-World Impact** |
| --- | --- | --- |
| **Smart pointers** | Manual memory management error-prone | Safer memory handling in browsers, games |
| **auto** | Verbose type declarations | Cleaner, maintainable code in ML/finance |
| **nullptr** | NULL ambiguous (int vs pointer) | Prevents pointer misuse in engines, APIs |
| **Range-for** | Verbose & error-prone loops | Concise, safer iteration in large data apps |
| **enum class** | Weakly typed enums | Stronger type safety in embedded/automotive |
| **Lambdas** | Verbose functors | Inline callbacks in GUIs, STL, threads |
| **Move semantics** | Copies were expensive | Massive performance boost in big data, games |

✅ **In short:**  
Modern C++ changes were driven by **three pillars**:

1. **Safety** (fewer crashes & leaks).
2. **Productivity** (shorter, clearer code).
3. **Performance** (move semantics, constexpr).

**2. Dynamic Memory Management**

Heap vs Stack allocation  
Manual memory management pitfalls  
RAII (Resource Acquisition Is Initialization)  
Smart pointers (std::unique*ptr, std::shared*ptr, std::weak\_ptr)  
Custom deleters and ownership semantics

**Dynamic Memory Management in C++**

Dynamic memory management allows programs to **allocate memory at runtime** rather than compile time. This is crucial when the size of required data isn’t known until execution.

**1. Static vs Dynamic Memory**

* **Static Memory (Stack)**
  + Allocated automatically (e.g., local variables).
  + Lifetime tied to scope.
  + Fast allocation/deallocation.
  + Size must be known at compile time.
* **Dynamic Memory (Heap)**
  + Allocated explicitly at runtime.
  + Lifetime controlled manually (before C++11).
  + Requires manual delete to free.
  + More flexible but error-prone.

**2. Traditional Dynamic Memory (C++98 style)**

#include <iostream>

using namespace std;

int main() {

// Allocate a single int

int\* ptr = new int(10);

cout << \*ptr << endl;

// Allocate an array

int\* arr = new int[5];

for (int i = 0; i < 5; i++) arr[i] = i + 1;

// Free memory

delete ptr; // For single element

delete[] arr; // For arrays

return 0;

}

**⚠️ Problems:**

* Memory leaks (forgetting delete).
* Dangling pointers (using after delete).
* Double delete.
* Exception safety issues.

**3. Modern C++ (C++11 and beyond)**

C++11 introduced **RAII-based Smart Pointers** to solve these problems.  
They automatically manage memory and free resources when no longer needed.

**🔹 unique\_ptr (exclusive ownership)**

#include <iostream>

#include <memory>

using namespace std;

int main() {

unique\_ptr<int> uptr(new int(42));

cout << \*uptr << endl;

// Ownership can be transferred

unique\_ptr<int> uptr2 = move(uptr);

return 0; // Memory automatically freed

}

**🔹 shared\_ptr (shared ownership)**

#include <iostream>

#include <memory>

using namespace std;

int main() {

shared\_ptr<int> sp1 = make\_shared<int>(100);

shared\_ptr<int> sp2 = sp1; // Reference count increases

cout << \*sp1 << ", " << \*sp2 << endl;

cout << "Use count: " << sp1.use\_count() << endl;

return 0; // Automatically freed when last shared\_ptr goes out of scope

}

**🔹 weak\_ptr (non-owning reference, avoids cyclic references)**

#include <iostream>

#include <memory>

using namespace std;

struct Node {

shared\_ptr<Node> next;

weak\_ptr<Node> prev; // prevents cyclic reference

};

int main() {

auto n1 = make\_shared<Node>();

auto n2 = make\_shared<Node>();

n1->next = n2;

n2->prev = n1; // safe, avoids memory leak

return 0;

}

**4. Move Semantics and Dynamic Memory**

C++11 move semantics make dynamic memory more efficient by **transferring ownership** instead of copying.

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> v1 = {1, 2, 3, 4};

vector<int> v2 = move(v1); // moves ownership instead of deep copy

cout << "Size of v1: " << v1.size() << endl;

cout << "Size of v2: " << v2.size() << endl;

return 0;

}

**5. Real-world Impact**

* **Before C++11**: manual memory management → leaks, crashes.
* **After C++11**: smart pointers + move semantics → safer, cleaner, faster code.
* Modern C++ encourages **Resource Acquisition Is Initialization (RAII)**: tie resources to object lifetime.

✅ **Summary:**

* Use new/delete only in rare, low-level cases.
* Prefer std::make\_unique and std::make\_shared.
* Avoid raw pointers for ownership.
* Use weak\_ptr to prevent cyclic dependencies.

**Heap vs Stack Allocation**

**1. Stack Allocation**

* **Definition**:  
  Stack is a region of memory that stores **function calls, local variables, and return addresses**.  
  It follows the **LIFO (Last In, First Out)** principle.
* **Characteristics**:
  + Memory is managed automatically.
  + Fast allocation & deallocation.
  + Size is limited (defined at program start, usually a few MBs).
  + Variables are destroyed once they go out of scope.
* **Example**:
* void foo() {
* int x = 10; // Stored on stack
* double arr[100]; // Entire array on stack
* } // x and arr are destroyed here automatically
* ✅ Advantages:
  + Fast, automatic cleanup.
  + No memory leaks.
* ❌ Disadvantages:
  + Limited size.
  + Lifetime tied to scope (cannot persist after function ends).

**2. Heap Allocation**

* **Definition**:  
  Heap is a region of memory used for **dynamic memory allocation** at runtime.  
  Managed using **new / delete** (or better: smart pointers).
* **Characteristics**:
  + Memory persists until explicitly freed.
  + More flexible but **slower** than stack.
  + Risk of **memory leaks** if not properly managed.
  + Fragmentation possible.
* **Example**:
* void foo() {
* int\* p = new int(20); // Allocated on heap
* double\* arr = new double[5]; // Dynamic array on heap
* delete p; // Must free memory
* delete[] arr; // Must free arrays separately
* }
* ✅ Advantages:
  + Flexible size.
  + Lifetime can extend beyond scope.
* ❌ Disadvantages:
  + Manual management (before C++11).
  + Slower access than stack.
  + Memory leaks & dangling pointers possible.

**3. Comparison Table**

| **Feature** | **Stack** | **Heap** |
| --- | --- | --- |
| **Speed** | Very fast (automatic) | Slower (manual / runtime) |
| **Size** | Limited, small (few MB) | Large (depends on RAM) |
| **Lifetime** | Scope-based (auto free) | Manual / programmer-defined |
| **Management** | Compiler | Programmer / Smart pointers |
| **Risk** | Stack overflow | Memory leaks, fragmentation |

**4. Real-World Impact**

* Use **stack** for small, short-lived data (e.g., local variables, small arrays).
* Use **heap** for large, persistent data structures (e.g., objects in game engines, linked lists, trees).
* Modern C++ (C++11 and beyond) encourages **smart pointers (std::unique\_ptr, std::shared\_ptr)** for safe heap usage.

✅ **Best Practice in Modern C++**

* Prefer **stack allocation** whenever possible.
* Use **RAII & smart pointers** when heap allocation is necessary.
* Avoid raw new and delete unless working with legacy code.

**Manual Memory Management Pitfalls**

**1. Memory Leaks**

* **Cause:** Forgetting to delete memory allocated with new.
* **Effect:** Program consumes more and more memory → crashes or slows down.
* **Example:**
* void leak() {
* int\* p = new int(42);
* // Forgot to delete p
* }

🔴 Every call leaks 4 bytes (or more, depending on alignment).

**2. Double Delete**

* **Cause:** Accidentally calling delete twice on the same pointer.
* **Effect:** Undefined behavior → often crashes.
* **Example:**
* int\* p = new int(10);
* delete p;
* delete p; // ❌ Double free error

**3. Dangling Pointers**

* **Cause:** Using a pointer after delete.
* **Effect:** Undefined behavior (can corrupt memory).
* **Example:**
* int\* p = new int(5);
* delete p;
* \*p = 10; // ❌ Accessing freed memory

**4. Uninitialized Pointers**

* **Cause:** Using a pointer without assigning it a valid address.
* **Effect:** Crash or unpredictable behavior.
* **Example:**
* int\* p; // uninitialized
* \*p = 5; // ❌ garbage location

**5. Mismatched Allocation/Deallocation**

* **Cause:** Using new[] but freeing with delete (or vice versa).
* **Effect:** Undefined behavior, memory corruption.
* **Example:**
* int\* arr = new int[10];
* delete arr; // ❌ should be delete[]

**6. Memory Fragmentation**

* **Cause:** Frequent allocations/deallocations of differently sized objects.
* **Effect:** Heap becomes fragmented → allocation failures, performance drops.

**7. Complex Ownership Semantics**

* **Cause:** Multiple pointers pointing to the same resource, without clear rules of who deletes it.
* **Effect:** Double deletes or leaks.
* **Example:**
* int\* p = new int(42);
* int\* q = p; // who deletes it? p or q?

**✅ How Modern C++ Fixes These**

* **std::unique\_ptr** – avoids leaks by ensuring single ownership.
* **std::shared\_ptr** – manages shared ownership with reference counting.
* **std::weak\_ptr** – avoids cyclic references in shared\_ptr.
* **nullptr** – avoids confusion with 0 or NULL.
* **RAII (Resource Acquisition Is Initialization)** – automatic cleanup.

**1. What is RAII?**

RAII is a **C++ programming idiom** where resource management (like memory, file handles, sockets, locks, etc.) is tied to the **lifetime of objects**.

* A resource is **acquired** during object construction.
* The resource is **released** automatically when the object is destroyed (when it goes out of scope).

👉 This ensures **deterministic cleanup** and prevents resource leaks.

**2. Core Idea**

* Instead of manually calling new/delete, fopen/fclose, or lock/unlock, wrap resources inside **classes**.
* Constructor acquires the resource.
* Destructor releases it.
* Scope exit (normal return or exception) ensures cleanup.

**3. Simple Example**

#include <iostream>

#include <fstream>

using namespace std;

class FileRAII {

ifstream file;

public:

FileRAII(const string& filename) {

file.open(filename);

if (!file.is\_open()) {

throw runtime\_error("Failed to open file");

}

cout << "File opened: " << filename << endl;

}

~FileRAII() {

if (file.is\_open()) {

file.close();

cout << "File closed." << endl;

}

}

ifstream& get() { return file; }

};

int main() {

try {

FileRAII file("example.txt");

string line;

while (getline(file.get(), line)) {

cout << line << endl;

} // file automatically closed here

} catch (const exception& e) {

cerr << "Error: " << e.what() << endl;

}

}

✅ File automatically closes at scope end, even if an exception occurs.

**4. RAII in Standard C++**

C++ standard library heavily uses RAII:

* **std::unique\_ptr** → manages heap memory.
* **std::shared\_ptr** → shared ownership of memory.
* **std::lock\_guard** → manages mutex locking/unlocking.
* **std::vector / std::string** → manage dynamic arrays and strings automatically.

Example with mutex:

#include <iostream>

#include <thread>

#include <mutex>

using namespace std;

mutex mtx;

void task() {

lock\_guard<mutex> lock(mtx); // locked here

cout << "Thread " << this\_thread::get\_id() << " working..." << endl;

} // automatically unlocked here

int main() {

thread t1(task), t2(task);

t1.join(); t2.join();

}

**5. Advantages**

* ✅ No resource leaks
* ✅ Exception-safe (cleanup happens automatically)
* ✅ Encapsulation of resource handling logic
* ✅ Easier, cleaner code

**6. Real-World Impact**

Before RAII:

* Developers had to remember to call delete or close() manually.
* Easy to forget, leading to **memory leaks** or **resource leaks**.

With RAII:

* Resources are **self-managed**.
* Encouraged use of smart pointers and containers.
* Paved way for **exception-safe code** in modern C++.

**Smart Pointers in C++**

In old C++, programmers had to manage memory manually using new and delete. Forgetting to delete caused **memory leaks**, and deleting the same pointer twice caused **undefined behavior**.

Smart pointers (introduced in **C++11**) solve this by wrapping raw pointers in a class that automatically manages the pointer’s lifetime.

**1. std::unique\_ptr**

* Represents **exclusive ownership** of a resource.
* Cannot be copied, only moved.
* When the unique\_ptr goes out of scope, the resource is automatically released.

✅ **Use when a resource should only have one owner.**

**Example:**

#include <iostream>

#include <memory>

using namespace std;

class Demo {

public:

Demo() { cout << "Demo created\n"; }

~Demo() { cout << "Demo destroyed\n"; }

void show() { cout << "Hello from Demo\n"; }

};

int main() {

unique\_ptr<Demo> p1 = make\_unique<Demo>(); // safe allocation

p1->show();

// unique\_ptr cannot be copied

// unique\_ptr<Demo> p2 = p1; ❌ Error

// But can be moved

unique\_ptr<Demo> p2 = move(p1);

if (!p1) cout << "p1 is empty after move\n";

return 0;

}

**2. std::shared\_ptr**

* Represents **shared ownership** of a resource.
* Multiple shared\_ptr can point to the same object.
* Uses **reference counting**: resource is deleted only when the last shared\_ptr is destroyed.

✅ **Use when multiple owners need access to the same resource.**

**Example:**

#include <iostream>

#include <memory>

using namespace std;

class Demo {

public:

Demo() { cout << "Demo created\n"; }

~Demo() { cout << "Demo destroyed\n"; }

};

int main() {

shared\_ptr<Demo> p1 = make\_shared<Demo>();

cout << "Reference count: " << p1.use\_count() << "\n";

{

shared\_ptr<Demo> p2 = p1;

cout << "Reference count: " << p1.use\_count() << "\n";

} // p2 goes out of scope, count decreases

cout << "Reference count after p2 destroyed: " << p1.use\_count() << "\n";

return 0;

}

**3. std::weak\_ptr**

* A **non-owning reference** to an object managed by shared\_ptr.
* Does not increase reference count → prevents **circular references**.
* Must be converted to shared\_ptr before use (via lock()).

✅ **Use when you need to observe a shared resource without keeping it alive.**

**Example:**

#include <iostream>

#include <memory>

using namespace std;

class Demo {

public:

Demo() { cout << "Demo created\n"; }

~Demo() { cout << "Demo destroyed\n"; }

};

int main() {

shared\_ptr<Demo> sp = make\_shared<Demo>();

weak\_ptr<Demo> wp = sp; // weak reference

cout << "Shared count: " << sp.use\_count() << "\n";

if (auto spt = wp.lock()) { // Check if resource is still alive

cout << "Object is still alive\n";

} else {

cout << "Object already destroyed\n";

}

sp.reset(); // destroys object

if (auto spt = wp.lock()) {

cout << "Object is still alive\n";

} else {

cout << "Object already destroyed\n";

}

return 0;

}

**✅ Summary**

| **Smart Pointer** | **Ownership** | **Copyable** | **Use Case** |
| --- | --- | --- | --- |
| unique\_ptr | Exclusive | ❌ No (only movable) | Single ownership |
| shared\_ptr | Shared | ✅ Yes | Multiple ownership |
| weak\_ptr | None | ✅ Yes | Avoid circular dependency |

**Custom Deleters in Smart Pointers**

Smart pointers (std::unique\_ptr, std::shared\_ptr) allow us to define **custom deleters** — functions or functors that specify **how an object should be destroyed**.

This is especially useful when:

* Objects are allocated with a custom API (e.g., malloc/free, fopen/fclose).
* Objects require non-standard cleanup (like closing a socket, unmapping memory, or logging).

**✅ Example 1: Custom Deleter with std::unique\_ptr**

#include <iostream>

#include <memory>

#include <cstdio>

int main() {

// FILE\* must be closed with fclose, not delete

auto fileCloser = [](FILE\* f) {

if (f) {

std::cout << "Closing file...\n";

fclose(f);

}

};

std::unique\_ptr<FILE, decltype(fileCloser)> filePtr(fopen("test.txt", "w"), fileCloser);

if (filePtr) {

std::fprintf(filePtr.get(), "Hello, file!\n");

}

}

✅ Output: "Closing file..." when filePtr goes out of scope.

**✅ Example 2: Custom Deleter with std::shared\_ptr**

#include <iostream>

#include <memory>

struct Connection {

void disconnect() { std::cout << "Disconnected.\n"; }

};

int main() {

auto connDeleter = [](Connection\* c) {

std::cout << "Custom cleanup...\n";

c->disconnect();

delete c;

};

std::shared\_ptr<Connection> conn(new Connection, connDeleter);

// Shared ownership

auto conn2 = conn;

}

✅ Output:

Custom cleanup...

Disconnected.

**🔹 Ownership Semantics in Smart Pointers**

Smart pointers encode **who owns an object** and **when it should be destroyed**:

**1. std::unique\_ptr → Exclusive Ownership**

* Only **one** owner at a time.
* Cannot be copied, only moved.
* Best for **resource ownership transfer**.

std::unique\_ptr<int> p1(new int(42));

auto p2 = std::move(p1); // Ownership transferred

**2. std::shared\_ptr → Shared Ownership**

* Multiple smart pointers can share ownership.
* Reference count decides lifetime.
* Useful for **shared resources**, but slightly slower due to ref-counting overhead.

auto sp1 = std::make\_shared<int>(10);

auto sp2 = sp1; // Both share ownership

**3. std::weak\_ptr → Non-owning Reference**

* Observes a shared\_ptr without increasing ref-count.
* Prevents **cyclic references** (common in graphs, trees).

#include <iostream>

#include <memory>

struct Node {

std::shared\_ptr<Node> next;

std::weak\_ptr<Node> prev; // avoids cycle

};

int main() {

auto n1 = std::make\_shared<Node>();

auto n2 = std::make\_shared<Node>();

n1->next = n2;

n2->prev = n1; // weak\_ptr prevents memory leak

}

**🔹 Summary**

* **Custom Deleters** let you define how cleanup happens (useful for non-delete resources).
* **Ownership Semantics**:
  + unique\_ptr → single owner
  + shared\_ptr → multiple owners, ref-counted
  + weak\_ptr → observer, no ownership

**3. STL Associative Containers & Algorithms**

Overview of STL architecture  
Associative containers: map, unordered*map, set, multimap, unordered*set  
Iterators, predicates, function objects  
Using std::algorithm: find\_if, sort, transform, accumulate, etc.  
Performance trade-offs and container selection guidelines

**STL Associative Containers**

Associative containers store elements in a way that allows **fast retrieval** using keys. Unlike sequence containers (vector, list, deque), these are usually implemented as **balanced binary search trees (red-black trees)** or **hash tables (unordered variants)**.

**1. Ordered Associative Containers**

Implemented as **balanced BST (red-black trees)**:

* **std::set** → Stores unique keys in sorted order.
* **std::multiset** → Stores duplicate keys in sorted order.
* **std::map** → Stores key-value pairs (unique keys, sorted).
* **std::multimap** → Stores key-value pairs with duplicate keys.

🔹 Operations:

* Insert, erase, search in **O(log n)**.
* Iteration gives elements in **sorted order**.

#include <iostream>

#include <set>

#include <map>

using namespace std;

int main() {

set<int> s = {5, 1, 3, 5, 2}; // duplicates ignored

s.insert(4);

cout << "Set elements: ";

for (auto x : s) cout << x << " "; // 1 2 3 4 5

cout << "\n";

map<string, int> m;

m["Alice"] = 25;

m["Bob"] = 30;

m["Charlie"] = 28;

cout << "Alice age: " << m["Alice"] << endl;

}

**2. Unordered Associative Containers (C++11)**

Implemented as **hash tables**:

* **std::unordered\_set**
* **std::unordered\_multiset**
* **std::unordered\_map**
* **std::unordered\_multimap**

🔹 Operations:

* Insert, erase, search in **O(1) average** (but **O(n) worst case** due to collisions).
* Elements are **not ordered**.

#include <iostream>

#include <unordered\_map>

using namespace std;

int main() {

unordered\_map<string, int> umap;

umap["India"] = 1380;

umap["USA"] = 331;

umap["China"] = 1441;

for (auto &p : umap) {

cout << p.first << " -> " << p.second << " million\n";

}

}

**🔹 STL Algorithms with Associative Containers**

STL algorithms are generic and work with iterators. Common algorithms with associative containers:

**Search**

auto it = s.find(3);

if (it != s.end()) cout << "Found: " << \*it;

**Count**

cout << "Occurrences of 5: " << s.count(5);

**Range-based queries (ordered only)**

auto it = s.lower\_bound(3); // first >= 3

auto it2 = s.upper\_bound(3); // first > 3

**Custom comparator**

struct Desc {

bool operator()(int a, int b) const {

return a > b; // descending

}

};

set<int, Desc> descSet = {1, 4, 2, 5, 3};

**Using std::algorithm**

Since associative containers provide bidirectional iterators, you can use:

* for\_each, copy, find\_if, count\_if, etc.  
  (but not algorithms requiring random access like sort() since set/map are already sorted).

#include <algorithm>

for\_each(s.begin(), s.end(), [](int x){ cout << x << " "; });

**🔹 Real-World Example**

👉 Counting word frequency (like in search engines or log processing).

#include <iostream>

#include <map>

#include <sstream>

using namespace std;

int main() {

string text = "apple orange apple banana orange apple";

map<string, int> freq;

stringstream ss(text);

string word;

while (ss >> word) {

freq[word]++;

}

for (auto &p : freq) {

cout << p.first << " -> " << p.second << endl;

}

}

✅ Output:

apple -> 3

banana -> 1

orange -> 2

✨ **Key takeaway:**

* Use **ordered containers** when you need **sorted traversal**.
* Use **unordered containers** when you need **fast access (hashing)**.
* Combine with **STL algorithms** for powerful, efficient coding.

**1. Core Components of STL**

The STL is built on **four main components**, each interacting seamlessly with others:

1. **Containers**
   * Data structures that store collections of objects.
   * Examples: vector, list, deque, set, map, unordered\_map.
2. **Iterators**
   * Generalized pointers used to traverse containers.
   * Types: input\_iterator, output\_iterator, forward\_iterator, bidirectional\_iterator, random\_access\_iterator.
3. **Algorithms**
   * Predefined generic functions that operate on containers through iterators.
   * Examples: sort(), find(), accumulate(), for\_each().
4. **Function Objects (Functors)**
   * Classes or structs that overload the operator() to behave like functions.
   * Can be used with algorithms for flexible operations.
   * Example: greater<int>() for descending sort.

**🔹 2. How They Work Together**

Think of STL as a **lego set**:

* **Containers** store data.
* **Iterators** provide a way to access that data.
* **Algorithms** manipulate the data through iterators.
* **Functors** allow customization of algorithms.

👉 Example:

#include <iostream>

#include <vector>

#include <algorithm>

#include <functional>

int main() {

std::vector<int> v = {5, 2, 8, 1, 3};

// Sort using STL algorithm and functor

std::sort(v.begin(), v.end(), std::greater<int>());

for (auto x : v) {

std::cout << x << " ";

}

return 0;

}

**Output:**

8 5 3 2 1

Here:

* vector = container
* begin() & end() = iterators
* sort() = algorithm
* greater<int>() = function object

**🔹 3. STL Design Philosophy**

* **Generic Programming**: Write once, use for any type (template-based).
* **Efficiency**: Provides performance comparable to handwritten data structures.
* **Interoperability**: Containers, algorithms, and iterators are designed to work together.

**🔹 4. Advantages of STL**

✅ Reduces development time (reusable components).  
✅ Highly optimized and tested by the C++ community.  
✅ Type-safe due to templates.  
✅ Flexibility with custom allocators, comparators, and functors.

**Associative Containers in C++ STL**

Unlike sequence containers (vector, list, etc.), **associative containers** store elements in a way that allows fast searching, insertion, and deletion using **keys**.

They are divided into:

1. **Ordered containers (based on balanced BST, usually Red-Black Tree)**
   * map, multimap, set, multiset
   * Elements are automatically **sorted**.
   * Operations: O(log n).
2. **Unordered containers (based on Hash Tables)**
   * unordered\_map, unordered\_multimap, unordered\_set, unordered\_multiset
   * Elements are stored in **buckets** (not sorted).
   * Operations: O(1) average, O(n) worst case.

**1. std::map**

* Stores **key-value pairs** ({key, value}) in **sorted order of keys**.
* **Unique keys only**.
* Implemented as a balanced BST (usually Red-Black Tree).
* Lookup, insertion, deletion → O(log n).

**Example:**

#include <iostream>

#include <map>

using namespace std;

int main() {

map<int, string> students;

students[101] = "Alice";

students[103] = "Bob";

students[102] = "Charlie";

for (auto& [id, name] : students) {

cout << id << " -> " << name << endl;

}

}

✅ Output (sorted by key):

101 -> Alice

102 -> Charlie

103 -> Bob

**2. std::unordered\_map**

* Stores key-value pairs in **no particular order**.
* **Unique keys only**.
* Implemented as **hash table**.
* Lookup, insertion, deletion → O(1) average.

**Example:**

#include <iostream>

#include <unordered\_map>

using namespace std;

int main() {

unordered\_map<string, int> freq;

freq["apple"]++;

freq["banana"]++;

freq["apple"]++;

for (auto& [fruit, count] : freq) {

cout << fruit << " -> " << count << endl;

}

}

✅ Output (order not guaranteed):

banana -> 1

apple -> 2

**3. std::set**

* Stores **unique keys** only (no duplicates).
* Elements stored in **sorted order**.
* Lookup, insertion, deletion → O(log n).

**Example:**

#include <iostream>

#include <set>

using namespace std;

int main() {

set<int> nums = {5, 1, 3, 5, 2};

for (int x : nums) cout << x << " ";

}

✅ Output:

1 2 3 5

**4. std::multimap**

* Similar to map, but **allows duplicate keys**.
* Stores multiple values under the same key.

**Example:**

#include <iostream>

#include <map>

using namespace std;

int main() {

multimap<int, string> marks;

marks.insert({90, "Alice"});

marks.insert({85, "Bob"});

marks.insert({90, "Charlie"});

for (auto& [score, name] : marks)

cout << score << " -> " << name << endl;

}

✅ Output:

85 -> Bob

90 -> Alice

90 -> Charlie

**5. std::unordered\_set**

* Stores **unique keys**, but **unordered** (uses hash).
* Lookup, insertion, deletion → O(1) average.

**Example:**

#include <iostream>

#include <unordered\_set>

using namespace std;

int main() {

unordered\_set<string> cities = {"Paris", "London", "Paris", "Berlin"};

for (auto& city : cities) cout << city << " ";

}

✅ Output (unordered):

Berlin Paris London

**🔹 Summary Table**

| **Container** | **Ordered?** | **Unique Keys?** | **Structure** | **Complexity** |
| --- | --- | --- | --- | --- |
| map | ✅ Yes | ✅ Yes | Red-Black Tree | O(log n) |
| unordered\_map | ❌ No | ✅ Yes | Hash Table | O(1) avg |
| set | ✅ Yes | ✅ Yes | Red-Black Tree | O(log n) |
| multimap | ✅ Yes | ❌ No | Red-Black Tree | O(log n) |
| unordered\_set | ❌ No | ✅ Yes | Hash Table | O(1) avg |

**Iterators, Predicates, and Function Objects in STL**

**1. Iterators**

Iterators act like **generalized pointers** that provide a way to traverse containers without exposing their internal structure.  
They are essential for working with STL algorithms.

**Types of Iterators:**

* **Input Iterator** → read-only, forward traversal (e.g., istream\_iterator).
* **Output Iterator** → write-only (e.g., ostream\_iterator).
* **Forward Iterator** → read/write, single-pass forward traversal (e.g., forward\_list).
* **Bidirectional Iterator** → forward & backward traversal (e.g., list, set, map).
* **Random Access Iterator** → allows pointer-like arithmetic (e.g., vector, deque).

**Example:**

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<int> nums = {1, 2, 3, 4, 5};

// Using iterator

vector<int>::iterator it;

for (it = nums.begin(); it != nums.end(); ++it) {

cout << \*it << " ";

}

cout << endl;

// Using range-based for loop (syntactic sugar over iterators)

for (int x : nums) {

cout << x << " ";

}

}

**2. Predicates**

A **predicate** is a function (or lambda) that returns a **boolean value** (true / false).  
Used to test conditions inside algorithms (like find\_if, count\_if, remove\_if, sort).

**Example:**

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

bool isEven(int x) {

return x % 2 == 0;

}

int main() {

vector<int> nums = {1, 2, 3, 4, 5, 6};

// Using predicate in count\_if

int count = count\_if(nums.begin(), nums.end(), isEven);

cout << "Number of even elements: " << count << endl;

// Using lambda instead of separate function

int oddCount = count\_if(nums.begin(), nums.end(), [](int x){ return x % 2 != 0; });

cout << "Number of odd elements: " << oddCount << endl;

}

**3. Function Objects (Functors)**

A **function object** (or **functor**) is a class/struct that overloads the **function call operator ()**.  
They behave like functions but can store **state/data**, making them more powerful than plain functions.

**Example: Custom Functor**

#include <iostream>

#include <algorithm>

#include <vector>

using namespace std;

struct MultiplyBy {

int factor;

MultiplyBy(int f) : factor(f) {}

int operator()(int x) const {

return x \* factor;

}

};

int main() {

vector<int> nums = {1, 2, 3, 4, 5};

vector<int> result(nums.size());

// Applying functor with transform

transform(nums.begin(), nums.end(), result.begin(), MultiplyBy(3));

for (int x : result) cout << x << " ";

}

👉 Output: 3 6 9 12 15

**✅ Real-World Use**

* **Iterators** → allow writing generic algorithms that work with any container.
* **Predicates** → enable filtering/searching with custom conditions.
* **Function Objects** → allow optimization (inlining) and stateful behavior (better than plain function pointers).

**Using std::algorithm in C++**

The <algorithm> header provides a rich set of generic functions that operate on ranges (via iterators).  
They work with all standard containers (vector, list, map, etc.), as long as iterators support the required category (input, forward, random access).

**🔹 1. std::find\_if**

Finds the first element in a range that matches a given predicate.

#include <iostream>

#include <vector>

#include <algorithm>

int main() {

std::vector<int> nums = {1, 3, 7, 10, 14, 20};

auto it = std::find\_if(nums.begin(), nums.end(), [](int n) {

return n > 10;

});

if (it != nums.end())

std::cout << "First number > 10: " << \*it << "\n";

else

std::cout << "No number greater than 10 found\n";

}

✅ **Output:**  
First number > 10: 14

**🔹 2. std::sort**

Sorts elements in a container (requires **random-access iterators**, e.g. vector, deque, but not list).

#include <iostream>

#include <vector>

#include <algorithm>

int main() {

std::vector<int> nums = {5, 2, 8, 1, 3};

std::sort(nums.begin(), nums.end()); // Ascending

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

std::cout << "\n";

// Sort in descending order using a lambda

std::sort(nums.begin(), nums.end(), [](int a, int b) {

return a > b;

});

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

}

✅ **Output:**  
1 2 3 5 8  
8 5 3 2 1

**🔹 3. std::transform**

Applies a function to a range and stores the result in another range.

#include <iostream>

#include <vector>

#include <algorithm>

int main() {

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

std::vector<int> squared(nums.size());

std::transform(nums.begin(), nums.end(), squared.begin(),

[](int n) { return n \* n; });

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

}

✅ **Output:**  
1 4 9 16

**🔹 4. std::accumulate (from <numeric>)**

Computes the sum or other accumulation across a range.

#include <iostream>

#include <vector>

#include <numeric> // for accumulate

int main() {

std::vector<int> nums = {1, 2, 3, 4, 5};

int sum = std::accumulate(nums.begin(), nums.end(), 0);

std::cout << "Sum = " << sum << "\n";

int product = std::accumulate(nums.begin(), nums.end(), 1,

[](int a, int b) { return a \* b; });

std::cout << "Product = " << product << "\n";

}

✅ **Output:**  
Sum = 15  
Product = 120

**🌟 Real-World Benefits**

* **Cleaner code:** eliminates manual loops.
* **Safer:** reduces errors compared to writing custom loops.
* **Reusability:** generic algorithms work across containers.
* **Performance:** many STL algorithms are highly optimized.

**Performance Trade-offs and Container Selection Guidelines**

**🔹 1. Sequence Containers (vector, deque, list, forward\_list)**

| **Container** | **Strengths** | **Weaknesses** | **When to Use** |
| --- | --- | --- | --- |
| **std::vector** | - Contiguous memory → cache-friendly, fast iteration - Amortized **O(1)** push\_back - Random access **O(1)** | - Insert/erase in middle **O(n)** - Growth may cause reallocations | Default choice for dynamic arrays, when random access & iteration speed matter |
| **std::deque** | - Fast insert/remove at both ends (**O(1)**) - Random access **O(1)** | - Slightly less cache-friendly than vector - More memory overhead | Double-ended queue behavior, but still need random access |
| **std::list** | - Insert/remove anywhere in **O(1)** (with iterator) - No reallocations | - No random access (**O(n)**) - Poor cache performance | Frequent insertion/deletion in middle, but iteration dominates |
| **std::forward\_list** | - Lower memory than list - Simple **O(1)** insertion/removal | - Single-linked → forward only - No random access | Lightweight list, embedded systems, memory-constrained |

**🔹 2. Associative Containers (map, set, multimap, multiset)**

| **Container** | **Strengths** | **Weaknesses** | **When to Use** |
| --- | --- | --- | --- |
| **std::map** (RB-tree) | - Ordered by key - Logarithmic **O(log n)** operations - In-order traversal supported | - More memory overhead than unordered\_map - Slower than hash lookup | Need sorted keys, range queries, predictable order |
| **std::set** | - Like map but stores only keys - Maintains uniqueness automatically | - Same as map | Need unique sorted elements |
| **std::multimap / std::multiset** | - Allows duplicate keys | - Same overhead as map/set | When duplicates are required in sorted order |

**🔹 3. Unordered Associative Containers (unordered\_map, unordered\_set, …)**

| **Container** | **Strengths** | **Weaknesses** | **When to Use** |
| --- | --- | --- | --- |
| **std::unordered\_map** | - Average **O(1)** insert/find/erase - Very fast lookups | - Worst-case **O(n)** (bad hash) - No ordering | Best for frequent lookups, hash-based indexing |
| **std::unordered\_set** | - Like unordered\_map, only keys | - Same weaknesses | Membership tests, fast lookups without ordering |
| **std::unordered\_multimap / unordered\_multiset** | - Allows duplicates | - May require rehashing | When duplicates are allowed but fast lookup needed |

**🔹 4. Container Selection Guidelines**

✅ **Prefer std::vector by default**

* Best cache locality, usually fastest in practice.
* Even if insertions in the middle are costly, vector often still outperforms lists for small/mid-size data due to CPU caching.

✅ **Use std::unordered\_map/set when lookup speed matters**

* Hash-based, constant-time average.
* Great for implementing symbol tables, frequency counters, indexing.

✅ **Use std::map/set when sorted order is needed**

* Range queries (lower\_bound, upper\_bound) possible.
* Deterministic ordering of elements.

✅ **Use std::list only when**:

* You really need stable iterators after insert/erase.
* Frequent insertion/removal in the middle without reallocation.

✅ **Don’t over-optimize prematurely**

* For small datasets, differences are negligible.
* Benchmark before deciding, especially with modern CPUs.

**⚡ Example: Choosing Between map and unordered\_map**

#include <iostream>

#include <map>

#include <unordered\_map>

#include <chrono>

int main() {

const int N = 1e6;

std::map<int, int> ordered;

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

// Insert test

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

for(int i=0; i<N; ++i) ordered[i] = i;

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

std::cout << "std::map insert: "

<< std::chrono::duration<double>(end-start).count() << "s\n";

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

for(int i=0; i<N; ++i) unordered[i] = i;

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

std::cout << "std::unordered\_map insert: "

<< std::chrono::duration<double>(end-start).count() << "s\n";

}

👉 Typically, unordered\_map will **dominate** in insertion and lookup, but map is useful if you need **sorted traversal**.

✅ **Rule of Thumb**:

* Start with std::vector.
* If you need fast lookups → unordered\_map/set.
* If you need sorted order → map/set.
* Only use list/deque for **very specific needs**.

**4. Design Patterns and Best Practices**

Core OO patterns in C++: Strategy, Observer, Factory, Singleton, Visitor  
SOLID principles applied in modern C++  
Dependency injection, policy-based design, interface segregation  
Avoiding anti-patterns in C++  
Industry Best Practices: Writing clean and maintainable code, adhering to coding standards.  
Exception Handling: Best practices for using exceptions and error handling in C++.

**Design Patterns in C++**

Design patterns provide **reusable solutions** to common software design problems. In C++, modern features (like auto, smart pointers, lambdas, move semantics, constexpr) make implementing patterns safer and easier.

**1. Creational Patterns**

These deal with **object creation mechanisms**.

* **Singleton**
  + Ensures only one instance exists.
  + In C++11, thread-safe initialization of static local variables makes it simpler:
  + class Singleton {
  + public:
  + static Singleton& getInstance() {
  + static Singleton instance; // thread-safe in C++11+
  + return instance;
  + }
  + private:
  + Singleton() = default;
  + };
* **Factory Method**
  + Encapsulates object creation, useful for polymorphic types:
  + struct Shape { virtual void draw() = 0; };
  + struct Circle : Shape { void draw() override { std::cout << "Circle\n"; }};
  + struct Square : Shape { void draw() override { std::cout << "Square\n"; }};
  + std::unique\_ptr<Shape> makeShape(const std::string& type) {
  + if (type == "circle") return std::make\_unique<Circle>();
  + if (type == "square") return std::make\_unique<Square>();
  + return nullptr;
  + }
* **Builder**
  + Helps construct complex objects step by step.

**2. Structural Patterns**

These focus on **object composition**.

* **Adapter**
  + Converts one interface into another.
  + class OldPrinter {
  + public:
  + void oldPrint() { std::cout << "Old printing\n"; }
  + };
  + class IPrinter {
  + public:
  + virtual void print() = 0;
  + };
  + class Adapter : public IPrinter {
  + OldPrinter& old;
  + public:
  + Adapter(OldPrinter& o) : old(o) {}
  + void print() override { old.oldPrint(); }
  + };
* **Decorator**
  + Adds functionality without altering existing code.
* **Observer (Pub-Sub)**
  + Used in GUIs, messaging, and event-driven systems.
  + class Observer {
  + public:
  + virtual void update(int value) = 0;
  + };
  + class Subject {
  + std::vector<Observer\*> observers;
  + int state{};
  + public:
  + void attach(Observer\* obs) { observers.push\_back(obs); }
  + void setState(int val) {
  + state = val;
  + for (auto\* obs : observers) obs->update(state);
  + }
  + };

**3. Behavioral Patterns**

These focus on **communication between objects**.

* **Strategy**
  + Defines a family of algorithms encapsulated as interchangeable classes.
  + class Strategy {
  + public:
  + virtual int execute(int a, int b) = 0;
  + };
  + class Add : public Strategy {
  + public:
  + int execute(int a, int b) override { return a + b; }
  + };
  + class Multiply : public Strategy {
  + public:
  + int execute(int a, int b) override { return a \* b; }
  + };
* **Command**
  + Encapsulates requests as objects (useful in undo/redo systems).

**🔹 Best Practices in Modern C++**

1. **Prefer RAII over manual resource management**
   * Use smart pointers instead of new/delete.
2. **Use auto and decltype wisely**
   * Improves readability and reduces type errors.
3. **Leverage move semantics**
   * Avoid unnecessary deep copies for performance.
4. **Use range-based for loops**
   * Cleaner and safer iteration.
5. **Use constexpr for compile-time evaluation**
   * Improves performance and correctness.
6. **Prefer enum class over plain enums**
   * Provides strong typing and avoids name collisions.
7. **Avoid raw pointers for ownership**
   * Use std::unique\_ptr and std::shared\_ptr instead.
8. **Favor algorithms over manual loops**
   * e.g., std::find\_if, std::accumulate, std::transform.
9. **Use std::thread, std::async, std::future for concurrency**
   * Avoid raw OS threads unless necessary.
10. **Follow the Rule of Zero / Five**

* Prefer types that don’t require custom copy/move/destructors.

**1. Strategy Pattern**

👉 **Intent**: Define a family of algorithms, encapsulate them, and make them interchangeable at runtime.

* Promotes **composition over inheritance**.
* Useful when you want to switch between algorithms dynamically.

**Example:**

#include <iostream>

#include <memory>

// Strategy Interface

class Strategy {

public:

virtual void execute() = 0;

virtual ~Strategy() = default;

};

// Concrete Strategies

class ConcreteStrategyA : public Strategy {

public:

void execute() override {

std::cout << "Executing Strategy A\n";

}

};

class ConcreteStrategyB : public Strategy {

public:

void execute() override {

std::cout << "Executing Strategy B\n";

}

};

// Context

class Context {

std::unique\_ptr<Strategy> strategy;

public:

void setStrategy(std::unique\_ptr<Strategy> s) {

strategy = std::move(s);

}

void execute() {

if (strategy) strategy->execute();

}

};

int main() {

Context ctx;

ctx.setStrategy(std::make\_unique<ConcreteStrategyA>());

ctx.execute();

ctx.setStrategy(std::make\_unique<ConcreteStrategyB>());

ctx.execute();

}

✅ **Use case**: Sorting with different algorithms, payment methods in e-commerce.

**2. Observer Pattern**

👉 **Intent**: Define a one-to-many dependency so that when one object changes state, all dependents are notified.

* Used in **event-driven systems, GUIs, messaging**.

**Example:**

#include <iostream>

#include <vector>

#include <memory>

class Observer {

public:

virtual void update(int value) = 0;

virtual ~Observer() = default;

};

class Subject {

std::vector<Observer\*> observers;

int state = 0;

public:

void attach(Observer\* obs) { observers.push\_back(obs); }

void setState(int value) {

state = value;

notify();

}

void notify() {

for (auto obs : observers) obs->update(state);

}

};

class ConcreteObserver : public Observer {

std::string name;

public:

ConcreteObserver(std::string n) : name(std::move(n)) {}

void update(int value) override {

std::cout << name << " received update: " << value << "\n";

}

};

int main() {

Subject subject;

ConcreteObserver obs1("Observer1"), obs2("Observer2");

subject.attach(&obs1);

subject.attach(&obs2);

subject.setState(42);

}

✅ **Use case**: Event listeners, GUI frameworks, stock price updates.

**3. Factory Pattern**

👉 **Intent**: Define an interface for creating objects, but let subclasses decide which class to instantiate.

* Promotes **loose coupling** between creation and usage.

**Example:**

#include <iostream>

#include <memory>

class Product {

public:

virtual void use() = 0;

virtual ~Product() = default;

};

class ConcreteProductA : public Product {

public:

void use() override { std::cout << "Using Product A\n"; }

};

class ConcreteProductB : public Product {

public:

void use() override { std::cout << "Using Product B\n"; }

};

class Factory {

public:

static std::unique\_ptr<Product> createProduct(char type) {

if (type == 'A') return std::make\_unique<ConcreteProductA>();

else return std::make\_unique<ConcreteProductB>();

}

};

int main() {

auto p1 = Factory::createProduct('A');

auto p2 = Factory::createProduct('B');

p1->use();

p2->use();

}

✅ **Use case**: GUI elements (buttons, windows), database connectors.

**4. Singleton Pattern**

👉 **Intent**: Ensure a class has only one instance and provide a global point of access.

* Frequently used for **logging, configuration managers, thread pools**.

**Example (thread-safe C++11):**

#include <iostream>

class Singleton {

private:

Singleton() { std::cout << "Singleton created\n"; }

public:

Singleton(const Singleton&) = delete;

Singleton& operator=(const Singleton&) = delete;

static Singleton& getInstance() {

static Singleton instance; // thread-safe since C++11

return instance;

}

void showMessage() {

std::cout << "Hello from Singleton\n";

}

};

int main() {

Singleton::getInstance().showMessage();

Singleton::getInstance().showMessage();

}

✅ **Use case**: Logging system, configuration loader, global cache.

**5. Visitor Pattern**

👉 **Intent**: Represent an operation to be performed on elements of an object structure without changing the classes.

* Useful for applying operations to objects in **class hierarchies**.

**Example:**

#include <iostream>

#include <vector>

#include <memory>

class Visitor;

class Element {

public:

virtual void accept(Visitor& v) = 0;

virtual ~Element() = default;

};

class ConcreteElementA;

class ConcreteElementB;

class Visitor {

public:

virtual void visit(ConcreteElementA& e) = 0;

virtual void visit(ConcreteElementB& e) = 0;

};

class ConcreteElementA : public Element {

public:

void accept(Visitor& v) override { v.visit(\*this); }

void featureA() { std::cout << "Feature A\n"; }

};

class ConcreteElementB : public Element {

public:

void accept(Visitor& v) override { v.visit(\*this); }

void featureB() { std::cout << "Feature B\n"; }

};

class ConcreteVisitor : public Visitor {

public:

void visit(ConcreteElementA& e) override { e.featureA(); }

void visit(ConcreteElementB& e) override { e.featureB(); }

};

int main() {

std::vector<std::unique\_ptr<Element>> elements;

elements.push\_back(std::make\_unique<ConcreteElementA>());

elements.push\_back(std::make\_unique<ConcreteElementB>());

ConcreteVisitor visitor;

for (auto& el : elements) el->accept(visitor);

}

✅ **Use case**: Compilers (AST traversal), object serialization, GUI element rendering.

✨ **Summary Table**

| **Pattern** | **Purpose** | **Example Use Case** |
| --- | --- | --- |
| Strategy | Swap algorithms dynamically | Payment methods, sorting |
| Observer | Publish/subscribe | Event listeners, stock updates |
| Factory | Decouple creation logic | GUI widgets, DB connectors |
| Singleton | One global instance | Logger, config manager |
| Visitor | Add operations to hierarchy without modifying classes | Compiler AST, GUI rendering |

**SOLID Principles in Modern C++**

**1. S – Single Responsibility Principle (SRP)**

*A class should have only one reason to change.*  
➡ Prevents "God classes" that do everything.

**✅ Example (C++ with SRP)**

#include <iostream>

#include <string>

// Bad: This class handles both data & file I/O

class UserBad {

std::string name;

public:

UserBad(const std::string& n) : name(n) {}

void saveToFile() { std::cout << "Saving user to file\n"; }

};

// Good: Separate responsibilities

class User {

std::string name;

public:

User(const std::string& n) : name(n) {}

std::string getName() const { return name; }

};

class UserRepository {

public:

void save(const User& user) {

std::cout << "Saving user: " << user.getName() << " to DB\n";

}

};

👉 Each class does **one job**:

* User → holds data.
* UserRepository → persistence logic.

**2. O – Open/Closed Principle (OCP)**

*Software entities should be open for extension, but closed for modification.*  
➡ Favor **polymorphism** and **composition**.

**✅ Example**

#include <iostream>

#include <memory>

// Base Interface

class Shape {

public:

virtual double area() const = 0;

virtual ~Shape() = default;

};

class Circle : public Shape {

double r;

public:

Circle(double radius) : r(radius) {}

double area() const override { return 3.14159 \* r \* r; }

};

class Rectangle : public Shape {

double w, h;

public:

Rectangle(double width, double height) : w(width), h(height) {}

double area() const override { return w \* h; }

};

void printArea(const Shape& s) {

std::cout << "Area: " << s.area() << "\n";

}

👉 Adding a new shape means **extending** (e.g., Triangle) instead of modifying existing code.

**3. L – Liskov Substitution Principle (LSP)**

*Objects of derived classes should be replaceable for base class objects without breaking the program.*

**✅ Example**

#include <iostream>

class Bird {

public:

virtual void fly() const = 0;

virtual ~Bird() = default;

};

class Sparrow : public Bird {

public:

void fly() const override { std::cout << "Sparrow flying\n"; }

};

// Violation: Penguin can't fly!

class Penguin : public Bird {

public:

void fly() const override { throw std::logic\_error("Penguins can't fly"); }

};

👉 Penguin **breaks LSP**.  
✅ Solution: Redesign hierarchy:

class Bird { public: virtual ~Bird() = default; };

class Flyable : public Bird { public: virtual void fly() const = 0; };

class Sparrow : public Flyable {

public: void fly() const override { std::cout << "Sparrow flying\n"; }

};

class Penguin : public Bird { /\* no fly() \*/ };

**4. I – Interface Segregation Principle (ISP)**

*Clients should not be forced to depend on interfaces they do not use.*

**✅ Example**

#include <iostream>

class Printer {

public:

virtual void print() = 0;

virtual void scan() = 0; // Bad: some printers don’t scan

virtual ~Printer() = default;

};

// Better: Separate interfaces

class IPrinter { public: virtual void print() = 0; virtual ~IPrinter() = default; };

class IScanner { public: virtual void scan() = 0; virtual ~IScanner() = default; };

class BasicPrinter : public IPrinter {

public: void print() override { std::cout << "Printing...\n"; }

};

class MultiFunctionPrinter : public IPrinter, public IScanner {

public:

void print() override { std::cout << "Printing...\n"; }

void scan() override { std::cout << "Scanning...\n"; }

};

👉 Avoids "fat" interfaces.

**5. D – Dependency Inversion Principle (DIP)**

*Depend on abstractions, not on concrete implementations.*

**✅ Example**

#include <iostream>

#include <memory>

class IMessageSender {

public:

virtual void sendMessage(const std::string& msg) = 0;

virtual ~IMessageSender() = default;

};

class EmailSender : public IMessageSender {

public:

void sendMessage(const std::string& msg) override {

std::cout << "Email: " << msg << "\n";

}

};

class Notification {

std::shared\_ptr<IMessageSender> sender;

public:

Notification(std::shared\_ptr<IMessageSender> s) : sender(s) {}

void notify(const std::string& msg) { sender->sendMessage(msg); }

};

👉 Notification doesn’t depend on EmailSender directly—it depends on **abstraction (IMessageSender)**, making it easy to plug in SmsSender, PushSender, etc.

**🔹 Real-World Impact**

✅ Cleaner, modular, testable code.  
✅ Easier unit testing (mocking interfaces).  
✅ Encourages separation of concerns.  
✅ Avoids rigid, tightly coupled designs.

**1. Dependency Injection in C++**

**Definition:**  
Instead of a class creating its own dependencies, they are “injected” from outside, making code loosely coupled, testable, and flexible.

**✅ Example: Without DI (Tightly Coupled)**

#include <iostream>

using namespace std;

class EmailService {

public:

void sendMessage(const string& msg) {

cout << "Sending Email: " << msg << endl;

}

};

class Notification {

EmailService service; // tightly coupled

public:

void notify(const string& msg) {

service.sendMessage(msg);

}

};

int main() {

Notification n;

n.notify("Hello!");

}

⚠️ Problem: Notification is hardcoded to use EmailService. Changing to SMSService requires modifying the class.

**✅ With Dependency Injection**

#include <iostream>

#include <memory>

using namespace std;

class IMessageService {

public:

virtual void sendMessage(const string& msg) = 0;

virtual ~IMessageService() = default;

};

class EmailService : public IMessageService {

public:

void sendMessage(const string& msg) override {

cout << "Sending Email: " << msg << endl;

}

};

class SMSService : public IMessageService {

public:

void sendMessage(const string& msg) override {

cout << "Sending SMS: " << msg << endl;

}

};

class Notification {

shared\_ptr<IMessageService> service;

public:

Notification(shared\_ptr<IMessageService> svc) : service(move(svc)) {}

void notify(const string& msg) {

service->sendMessage(msg);

}

};

int main() {

auto emailSvc = make\_shared<EmailService>();

auto smsSvc = make\_shared<SMSService>();

Notification n1(emailSvc);

Notification n2(smsSvc);

n1.notify("Meeting at 5!");

n2.notify("Code review at 3!");

}

➡️ Notification no longer depends on a specific implementation → **easy testing, swapping services, and reusability**.

**🔹 2. Policy-Based Design**

**Definition:**  
A flexible design technique using **templates** where behavior is defined by “policies” (strategies) passed as template parameters.

**✅ Example: Policy-Based Logging**

#include <iostream>

#include <string>

using namespace std;

// Policies

struct ConsoleLogger {

void log(const string& msg) { cout << "[Console] " << msg << endl; }

};

struct FileLogger {

void log(const string& msg) { cout << "[File] " << msg << endl; } // Simulating

};

// Generic class with policy

template <typename LoggerPolicy>

class Application {

LoggerPolicy logger;

public:

void process(const string& data) {

logger.log("Processing: " + data);

}

};

int main() {

Application<ConsoleLogger> app1;

Application<FileLogger> app2;

app1.process("User request");

app2.process("System event");

}

➡️ Application doesn’t know or care how logging is done → policy decides.  
⚡ Used in **STL containers & allocators** (e.g., std::vector with custom allocators).

**🔹 3. Interface Segregation Principle (ISP)**

**Definition:**  
A class should not be forced to depend on interfaces it doesn’t use.  
👉 Split large interfaces into smaller, specific ones.

**❌ Bad Example: Fat Interface**

class Machine {

public:

virtual void print() = 0;

virtual void scan() = 0;

virtual void fax() = 0;

};

If a simple printer class implements this, it must provide dummy scan() & fax() → violates ISP.

**✅ Good Example: Segregated Interfaces**

class IPrinter {

public:

virtual void print() = 0;

virtual ~IPrinter() = default;

};

class IScanner {

public:

virtual void scan() = 0;

virtual ~IScanner() = default;

};

class Printer : public IPrinter {

public:

void print() override { cout << "Printing document..." << endl; }

};

class Scanner : public IScanner {

public:

void scan() override { cout << "Scanning document..." << endl; }

};

// Multi-function device

class MultiFunctionPrinter : public IPrinter, public IScanner {

public:

void print() override { cout << "MFP Printing..." << endl; }

void scan() override { cout << "MFP Scanning..." << endl; }

};

➡️ Now, single-purpose classes only implement what they **need**.  
➡️ Multifunction device composes the required interfaces.

**🔹 Summary**

* **Dependency Injection (DI):** Externalize dependencies for flexibility & testability.
* **Policy-Based Design:** Use templates to select behavior at compile time → very efficient.
* **Interface Segregation:** Favor smaller, focused interfaces → avoids bloated abstractions.

**🚫 Common Anti-Patterns in C++ and Their Alternatives**

**1. God Object / God Class**

* **Problem**: A single class knows and does too much (business logic, data, UI, etc.).
* **Why bad?**: Hard to maintain, tightly coupled, violates **Single Responsibility Principle (SRP)**.
* ✅ **Solution**:
  + Break into smaller classes.
  + Use **composition** and **dependency injection**.
  + Example: Split ApplicationManager into Logger, ConfigLoader, NetworkHandler, etc.

**2. Raw new / delete Everywhere**

* **Problem**: Manual memory management causes leaks, dangling pointers, and double frees.
* **Why bad?**: Exception safety issues and hard-to-track bugs.
* ✅ **Solution**:
  + Use **RAII** and **smart pointers** (std::unique\_ptr, std::shared\_ptr, std::weak\_ptr).
  + Use **containers** (std::vector, std::map) instead of raw arrays or linked lists unless you *really* need control.

// ❌ Bad

Foo\* f = new Foo();

delete f;

// ✅ Good

auto f = std::make\_unique<Foo>();

**3. Overusing Macros (#define)**

* **Problem**: Macros pollute namespaces, have no type safety, and can lead to unexpected errors.
* ✅ **Solution**:
  + Use **constexpr**, **inline functions**, or **enum class**.

// ❌ Bad

#define PI 3.14159

// ✅ Good

constexpr double PI = 3.14159;

**4. Copy-Paste Inheritance (Prefer Composition)**

* **Problem**: Using inheritance for code reuse, creating fragile hierarchies.
* **Why bad?**: Leads to tight coupling and fragile base class problem.
* ✅ **Solution**:
  + Use **composition** and **policy-based design**.
  + Inherit only for **is-a** relationships.

**5. Singleton Overuse**

* **Problem**: Makes code globally dependent, hides dependencies, hard to test.
* ✅ **Solution**:
  + Use **dependency injection** instead.
  + If you *must* use a Singleton, ensure it’s **thread-safe** and **lazy-loaded**.

**6. Using void\* and C-style Casts**

* **Problem**: Loses type safety, error-prone, bypasses compiler checks.
* ✅ **Solution**:
  + Use **templates**, **auto**, or **smart casting** (static\_cast, dynamic\_cast, reinterpret\_cast) with care.

**7. Exception-Safety Neglect**

* **Problem**: Manual resource handling causes leaks if exceptions occur.
* ✅ **Solution**:
  + Use **RAII** (std::unique\_ptr, std::lock\_guard, etc.).
  + Design exception-safe classes (destructors should never throw).

**8. Premature Optimization**

* **Problem**: Writing overly complex code for theoretical performance gains.
* ✅ **Solution**:
  + Write **clear and correct** code first.
  + Use **profilers** before optimizing.
  + Choose the right **STL container/algorithm** instead of hand-rolled structures.

**9. Not Leveraging Modern C++ Features**

* **Problem**: Sticking to C++98 style (manual loops, raw arrays, verbose syntax).
* ✅ **Solution**:
  + Use **range-based for**, **lambdas**, **auto**, **structured bindings**, **std::optional**, **std::variant**.

// ❌ Old way

for (int i = 0; i < vec.size(); ++i) {

std::cout << vec[i] << "\n";

}

// ✅ Modern way

for (auto& val : vec) {

std::cout << val << "\n";

}

**🌟 Best Practices to Avoid Anti-Patterns**

* Follow **SOLID principles**.
* Use **RAII** and **smart pointers**.
* Prefer **composition over inheritance**.
* Use **modern C++ features** for safety and readability.
* Avoid **global state** (Singletons, global variables).
* Write **unit tests** to detect design flaws early.

**Industry Best Practices in Modern C++**

**1. Code Readability & Maintainability**

* **Consistent Naming Conventions**
  + Use camelCase or snake\_case consistently.
  + Classes/structs → PascalCase, variables/functions → camelCase.
* **Comment Smartly**
  + Prefer **self-documenting code** over excessive comments.
  + Use comments for **why**, not **what** (since modern C++ is readable with descriptive naming).
* **Avoid Long Functions**
  + Keep functions small (single responsibility).
  + Helps debugging, testing, and reusability.

**2. Modern C++ Features Over Legacy**

* Prefer auto for type inference (but don’t overuse where clarity is lost).
* Use nullptr instead of NULL or 0.
* Use **range-based for loops** for readability:
* for (const auto& item : container) { ... }
* Prefer **enum class** instead of raw enums (avoids namespace pollution).
* Use **smart pointers** (unique\_ptr, shared\_ptr, weak\_ptr) over raw pointers.

**3. Error Handling**

* Use **exceptions** for errors, not return codes.
* Define custom exception types if needed.
* Prefer std::optional, std::variant, or expected<T, E> (C++23) for recoverable cases.

**4. Memory & Resource Management**

* Follow **RAII** (Resource Acquisition Is Initialization).
* Avoid manual new and delete.
* Use **smart pointers** with custom deleters if necessary.
* Use containers (std::vector, std::array) instead of raw arrays.

**5. Design Principles**

* Apply **SOLID Principles**:
  + **S**ingle Responsibility → each class does one thing well.
  + **O**pen/Closed → open for extension, closed for modification.
  + **L**iskov Substitution → derived classes should be usable via base class interface.
  + **I**nterface Segregation → prefer multiple small interfaces.
  + **D**ependency Inversion → depend on abstractions, not concretions.
* Use **Design Patterns** where applicable (Observer, Factory, Strategy, Visitor).
* Avoid **God classes**, **spaghetti inheritance**, and over-engineering.

**6. Performance & Efficiency**

* Choose the **right STL container** (vector > list in most cases).
* Avoid premature optimization — but profile when needed.
* Use move semantics (std::move, std::forward) for performance.
* Prefer **emplace\_back** over push\_back when constructing objects in place.

**7. Coding Standards & Tools**

* Follow widely accepted **style guides**:
  + **Google C++ Style Guide**
  + **C++ Core Guidelines (Bjarne Stroustrup & Herb Sutter)**
* Use **static analyzers** (clang-tidy, cppcheck) to detect issues early.
* Enable **compiler warnings** (-Wall -Wextra -Werror) and treat warnings as errors.
* Use **unit testing frameworks** (Google Test, Catch2) for maintainability.

**8. Concurrency & Safety**

* Prefer **std::thread**, std::async, std::future over raw pthreads.
* Use **std::mutex**, **std::lock\_guard**, **std::scoped\_lock** for thread safety.
* Avoid **data races** by design (favor immutability when possible).

**9. Documentation & Collaboration**

* Use **Doxygen** or similar tools for API documentation.
* Keep code reviews regular & constructive.
* Write **meaningful commit messages** and maintain version control (Git).

✅ **Key Takeaway**:  
Modern C++ emphasizes **safety, readability, maintainability, and performance**. By leveraging **RAII, smart pointers, modern STL, SOLID principles, and coding standards**, developers can write **robust, industry-grade software** that scales well.

**Exception Handling in C++: Best Practices**

**1. Use Exceptions for Exceptional Situations Only**

* Exceptions should signal **errors that cannot be handled locally** and are truly “exceptional”.
* Do **not** use exceptions for:
  + Normal control flow
  + Simple validation checks (e.g., if (x == 0) before division)

✅ Good:

int divide(int a, int b) {

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

return a / b;

}

❌ Bad:

try {

throw 1; // using exceptions for simple checks

} catch (int) {

// do something trivial

}

**2. Prefer RAII for Resource Management**

* Use **RAII** (Resource Acquisition Is Initialization) so that resources (memory, files, sockets) are released automatically when exceptions are thrown.

✅ Example:

void processFile(const std::string& filename) {

std::ifstream file(filename); // RAII: closes automatically

if (!file) throw std::runtime\_error("File not found");

std::string line;

while (std::getline(file, line)) {

// process

}

} // file closes automatically here

**3. Throw Exceptions by Value, Catch by Reference**

* Always throw **objects** (usually derived from std::exception).
* Catch by **const reference** to avoid slicing and unnecessary copies.

✅ Good:

throw std::runtime\_error("Something went wrong");

try {

// risky code

} catch (const std::runtime\_error& e) {

std::cerr << e.what() << '\n';

}

❌ Bad:

throw "Error"; // throwing raw string (bad)

catch (std::exception e) { ... } // slicing

**4. Use Standard Exception Types When Possible**

* Prefer std::runtime\_error, std::logic\_error, std::invalid\_argument, etc., over custom exceptions.
* Create custom exceptions only when they carry extra info.

**5. Avoid throw() and Prefer noexcept**

* throw() (old-style exception specifier) is deprecated.
* Use noexcept for functions that **guarantee not to throw**.

✅ Example:

void logMessage(const std::string& msg) noexcept {

std::cout << msg << '\n';

}

**6. Don’t Throw in Destructors**

* Throwing inside a destructor during stack unwinding (when another exception is active) calls std::terminate().
* Instead, **catch exceptions inside destructors** and log them.

✅ Example:

class Resource {

public:

~Resource() {

try {

// cleanup that may throw

} catch (...) {

// log and swallow

}

}

};

**7. Keep Exception Safety Guarantees**

Three levels of **exception safety**:

1. **Basic Guarantee** → No resource leaks, object in valid state.
2. **Strong Guarantee** → Operation is either completed successfully or has no effect.
3. **No-Throw Guarantee** → Function never throws (use noexcept).

✅ Example: Using copy-and-swap idiom for strong exception safety.

class Buffer {

std::vector<int> data;

public:

void setData(std::vector<int> newData) {

std::swap(data, newData); // strong guarantee

}

};

**8. Document Exception Behavior**

* Clearly state which functions may throw and under what conditions.
* For public APIs, specify exception guarantees in comments or docs.

**9. Use std::optional or std::expected (C++17/23) for Expected Failures**

* If failure is common and expected, don’t use exceptions—use std::optional or std::expected.

✅ Example with std::optional:

std::optional<int> findValue(const std::vector<int>& v, int x) {

auto it = std::find(v.begin(), v.end(), x);

if (it != v.end()) return \*it;

return std::nullopt; // expected failure

}

**🔑 Key Takeaways**

✔ Use exceptions for rare, unexpected errors.  
✔ RAII + smart pointers make exception handling safer.  
✔ Throw by value, catch by const reference.  
✔ Never throw from destructors.  
✔ Prefer standard exceptions and noexcept.  
✔ Use std::optional/std::expected for expected failures.

**5. Concurrency and Parallelism**

Thread management with std::thread  
Mutexes, condition variables, locks (std::mutex, std::lock*guard, std::unique*lock)  
Thread-safe data sharing  
Task-based concurrency: std::async, std::future, std::promise

**Concurrency and Parallelism in C++**

Modern C++ (from **C++11 onward**) provides standardized libraries and constructs for writing **multi-threaded and parallel programs** safely and efficiently.

**1. Concurrency vs Parallelism**

* **Concurrency** → Multiple tasks **logically progressing** at the same time (e.g., multitasking on a single CPU core by context switching).
* **Parallelism** → Multiple tasks **physically executing simultaneously** (e.g., using multiple CPU cores or SIMD instructions).

👉 Concurrency = structure of programs.  
👉 Parallelism = performance improvement.

**2. Concurrency in C++**

C++ provides primitives for multi-threading through <thread>, <mutex>, <future>, etc.

**a) Threads**

#include <iostream>

#include <thread>

void task(int id) {

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

}

int main() {

std::thread t1(task, 1);

std::thread t2(task, 2);

t1.join(); // Wait for t1

t2.join(); // Wait for t2

}

* std::thread runs a function concurrently.
* join() waits for a thread to finish.
* detach() runs a thread independently (not recommended unless necessary).

**b) Synchronization with Mutex**

To prevent **data races**:

#include <iostream>

#include <thread>

#include <mutex>

std::mutex mtx;

int counter = 0;

void increment() {

for(int i = 0; i < 1000; ++i) {

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

++counter;

}

}

int main() {

std::thread t1(increment), t2(increment);

t1.join(); t2.join();

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

}

✅ std::lock\_guard ensures RAII-based lock/unlock.  
✅ Prevents race conditions.

**c) Condition Variables**

Used for **thread communication**:

#include <iostream>

#include <thread>

#include <condition\_variable>

#include <mutex>

std::mutex m;

std::condition\_variable cv;

bool ready = false;

void worker() {

std::unique\_lock<std::mutex> lock(m);

cv.wait(lock, []{ return ready; });

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

}

int main() {

std::thread t(worker);

{

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

ready = true;

}

cv.notify\_one(); // wake worker

t.join();

}

**d) Futures, Promises, and async**

For **safe asynchronous execution**:

#include <future>

#include <iostream>

int compute() {

return 42;

}

int main() {

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

std::cout << "Answer: " << result.get() << "\n"; // waits for compute()

}

* std::async launches tasks asynchronously.
* std::future retrieves the result.
* std::promise can be used for **manual communication** between threads.

**3. Parallelism in C++17/20**

**a) Parallel STL Algorithms (C++17)**

C++17 introduced execution policies:

#include <vector>

#include <algorithm>

#include <execution>

#include <iostream>

int main() {

std::vector<int> v(1e6, 1);

// Parallel reduction (sum)

long long sum = std::reduce(std::execution::par, v.begin(), v.end(), 0LL);

std::cout << "Sum: " << sum << "\n";

}

* std::execution::seq → sequential
* std::execution::par → parallel
* std::execution::par\_unseq → parallel + vectorized

**b) Coroutines (C++20)**

Enable **asynchronous programming** with co\_await:

#include <iostream>

#include <coroutine>

#include <thread>

struct Task {

struct promise\_type {

Task get\_return\_object() { return {}; }

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

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

void return\_void() {}

void unhandled\_exception() {}

};

};

Task foo() {

std::cout << "Inside coroutine\n";

co\_return;

}

int main() {

foo(); // Coroutine execution

}

**4. Real-World Impact**

* **Servers** → Concurrent request handling (HTTP servers, DB engines).
* **Games** → Parallel AI, rendering, physics.
* **Finance** → Low-latency trading, concurrent computations.
* **Big Data / AI** → Parallel numeric computations.

✅ **Key Takeaways**

* Use **std::thread** for direct threading, but prefer **std::async** for safer async tasks.
* Use **mutexes & locks** to prevent race conditions.
* Use **condition variables** for inter-thread communication.
* Use **Parallel STL (C++17+)** for high-performance data processing.
* Use **Coroutines (C++20)** for lightweight async programming.

**Thread Management with std::thread**

C++11 introduced the **std::thread** class in the <thread> header to allow creating and managing threads directly in a portable way.

**✅ 1. Creating and Running Threads**

You can launch a thread by passing a **function**, **lambda**, or **callable object**:

#include <iostream>

#include <thread>

void task(int x) {

std::cout << "Task running with value: " << x << "\n";

}

int main() {

std::thread t1(task, 5); // Start thread with function

std::thread t2([] { // Start thread with lambda

std::cout << "Lambda task running\n";

});

t1.join(); // Wait for thread t1 to finish

t2.join(); // Wait for thread t2 to finish

return 0;

}

**✅ 2. Joining vs Detaching Threads**

* **join()**: Main thread waits until the thread finishes.
* **detach()**: Thread runs independently in the background.

std::thread t([]{

std::cout << "Detached thread running...\n";

});

t.detach(); // Thread runs in background, main won't wait

⚠️ Detached threads must not access resources that may go out of scope (danger of dangling references).

**✅ 3. Passing Arguments to Threads**

Arguments are copied by default, but can be passed by reference using std::ref:

#include <functional> // for std::ref

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

std::cout << msg << "\n";

}

int main() {

std::string text = "Hello from thread!";

std::thread t(printMessage, std::ref(text)); // Pass by reference

t.join();

}

**✅ 4. Thread Identification**

Each thread has an **ID**:

std::cout << "Thread ID: " << std::this\_thread::get\_id() << "\n";

**✅ 5. Hardware Concurrency**

Query number of supported concurrent threads:

unsigned int cores = std::thread::hardware\_concurrency();

std::cout << "System supports " << cores << " threads\n";

**✅ 6. Thread Synchronization (Preview)**

Using multiple threads often requires synchronization to avoid race conditions. C++ provides:

* std::mutex, std::lock\_guard, std::unique\_lock
* std::condition\_variable
* std::future, std::promise
* Atomic operations (std::atomic)

Example with mutex:

#include <mutex>

std::mutex mtx;

int counter = 0;

void increment() {

for (int i = 0; i < 1000; i++) {

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

counter++;

}

}

**✅ 7. Real-World Example: Parallel Computation**

#include <iostream>

#include <thread>

#include <vector>

void worker(int id) {

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

}

int main() {

std::vector<std::thread> threads;

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

threads.emplace\_back(worker, i);

}

for (auto &t : threads) {

t.join();

}

return 0;

}

**🔹 Best Practices**

* Always **join or detach** threads before the main program exits.
* Prefer **RAII-style thread management** (e.g., use wrapper classes).
* Avoid **data races** → use mutexes, atomics, or message-passing.
* Consider **higher-level abstractions** (std::async, thread pools) for scalability.

**Mutexes, Condition Variables, and Locks in C++**

Modern C++ (C++11 and beyond) provides a robust threading and synchronization library in <thread> and <mutex>. These features help prevent **data races** and coordinate execution between threads.

**1. std::mutex**

A **mutex (mutual exclusion)** ensures that only one thread at a time can access a shared resource.

#include <iostream>

#include <thread>

#include <mutex>

std::mutex mtx;

int counter = 0;

void increment(int id) {

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

mtx.lock(); // acquire lock

++counter;

std::cout << "Thread " << id << " incremented counter to " << counter << "\n";

mtx.unlock(); // release lock

}

}

int main() {

std::thread t1(increment, 1);

std::thread t2(increment, 2);

t1.join();

t2.join();

}

⚠️ **Pitfall:** Forgetting unlock() can cause deadlocks. That’s why RAII-based **lock wrappers** (std::lock\_guard, std::unique\_lock) are preferred.

**2. std::lock\_guard**

RAII wrapper for std::mutex. Automatically locks in constructor and unlocks in destructor.

void safe\_increment(int id) {

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

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

++counter;

std::cout << "Thread " << id << " safely incremented counter to " << counter << "\n";

}

}

✅ Guarantees unlock even if exceptions occur.  
🚫 No manual unlock possible before destruction.

**3. std::unique\_lock**

More flexible than lock\_guard:

* Can be locked/unlocked multiple times.
* Supports deferred locking (std::defer\_lock).
* Works with **condition variables**.

void safe\_increment\_with\_unique(int id) {

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

std::unique\_lock<std::mutex> lock(mtx);

++counter;

std::cout << "Thread " << id << " using unique\_lock, counter = " << counter << "\n";

lock.unlock(); // allows other work in same scope

}

}

**4. Condition Variables**

Used for **thread communication** (e.g., producer-consumer).  
Threads can **wait** until a condition is met, avoiding busy-waiting.

#include <condition\_variable>

std::condition\_variable cv;

std::mutex cv\_mtx;

bool ready = false;

void worker() {

std::unique\_lock<std::mutex> lock(cv\_mtx);

cv.wait(lock, []{ return ready; }); // wait until "ready" becomes true

std::cout << "Worker thread proceeding...\n";

}

int main() {

std::thread t(worker);

{

std::lock\_guard<std::mutex> lock(cv\_mtx);

ready = true;

}

cv.notify\_one(); // wake one waiting thread

t.join();

}

**Key Notes:**

* cv.wait(lock, predicate) releases the lock while waiting and reacquires it before resuming.
* notify\_one() wakes a single waiting thread.
* notify\_all() wakes all waiting threads.

**5. Summary & Best Practices**

✅ Use std::lock\_guard for simple locking.  
✅ Use std::unique\_lock for advanced cases (deferred locking, multiple unlock/lock, condition variables).  
✅ Always avoid manual mutex.lock()/mutex.unlock() unless absolutely necessary.  
✅ Prefer cv.wait(lock, predicate) to avoid **spurious wakeups**.  
✅ Keep critical sections **small** to reduce contention.

**Thread-Safe Data Sharing in C++**

When multiple threads share data, **race conditions** can occur if access is not properly synchronized. C++11 and beyond introduced a set of primitives to ensure **safe concurrent access**.

**1. The Problem: Race Conditions**

A **race condition** happens when multiple threads read/write shared data simultaneously without synchronization.

**Example (Problematic Code)**

#include <iostream>

#include <thread>

#include <vector>

int counter = 0; // shared resource

void increment(int n) {

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

counter++; // Not thread-safe

}

}

int main() {

std::thread t1(increment, 1000000);

std::thread t2(increment, 1000000);

t1.join();

t2.join();

std::cout << "Counter: " << counter << std::endl;

}

👉 Expected: 2000000, but actual output varies due to race conditions.

**2. Fix: Using std::mutex**

A **mutex** ensures only one thread modifies shared data at a time.

#include <iostream>

#include <thread>

#include <mutex>

int counter = 0;

std::mutex mtx;

void increment(int n) {

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

std::lock\_guard<std::mutex> lock(mtx); // RAII-style locking

counter++;

}

}

int main() {

std::thread t1(increment, 1000000);

std::thread t2(increment, 1000000);

t1.join();

t2.join();

std::cout << "Counter: " << counter << std::endl; // Always 2000000

}

✅ Correct result guaranteed.

**3. Reader-Writer Pattern**

If multiple threads **read** more often than **write**, use **std::shared\_mutex** (C++17).

#include <iostream>

#include <shared\_mutex>

#include <thread>

#include <vector>

std::shared\_mutex rw\_mtx;

int sharedData = 0;

void reader(int id) {

std::shared\_lock<std::shared\_mutex> lock(rw\_mtx);

std::cout << "Reader " << id << " sees: " << sharedData << "\n";

}

void writer(int value) {

std::unique\_lock<std::shared\_mutex> lock(rw\_mtx);

sharedData = value;

std::cout << "Writer updated value to " << sharedData << "\n";

}

int main() {

std::thread t1(reader, 1);

std::thread t2(writer, 42);

std::thread t3(reader, 2);

t1.join(); t2.join(); t3.join();

}

👉 Readers can access simultaneously, writers get exclusive access.

**4. Thread-Safe Queues**

Common in producer-consumer systems. Using std::mutex + std::condition\_variable:

#include <iostream>

#include <thread>

#include <queue>

#include <mutex>

#include <condition\_variable>

std::queue<int> q;

std::mutex mtx;

std::condition\_variable cv;

void producer() {

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

{

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

q.push(i);

}

cv.notify\_one(); // wake up one waiting consumer

}

}

void consumer() {

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

std::unique\_lock<std::mutex> lock(mtx);

cv.wait(lock, [] { return !q.empty(); }); // waits until q not empty

int value = q.front(); q.pop();

lock.unlock();

std::cout << "Consumed: " << value << "\n";

}

}

int main() {

std::thread t1(producer);

std::thread t2(consumer);

t1.join();

t2.join();

}

**5. When to Use Atomics Instead of Locks**

If shared data is simple (like counters), **std::atomic** is faster than mutex.

#include <iostream>

#include <thread>

#include <atomic>

std::atomic<int> counter(0);

void increment(int n) {

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

counter.fetch\_add(1, std::memory\_order\_relaxed);

}

}

int main() {

std::thread t1(increment, 1000000);

std::thread t2(increment, 1000000);

t1.join();

t2.join();

std::cout << "Counter: " << counter << std::endl; // Always 2000000

}

**✅ Guidelines for Thread-Safe Data Sharing**

1. **Prefer immutability** → share read-only data whenever possible.
2. **Use std::mutex** for complex shared state.
3. **Use std::shared\_mutex** for reader-heavy workloads.
4. **Use std::atomic** for counters/flags.
5. **Avoid data races** at all costs – undefined behavior is disastrous.
6. **Don’t overuse locks** → can cause deadlocks, use lock hierarchy or std::scoped\_lock.

**Task-Based Concurrency in C++**

Unlike **manual thread management** (std::thread), task-based concurrency allows you to express *what* should run concurrently without worrying too much about *how* it’s scheduled.

**1. std::async**

* Launches a task asynchronously (in a separate thread) or lazily (on demand).
* Returns a std::future that holds the result of the computation.

**Example:**

#include <iostream>

#include <future>

#include <chrono>

int computeSquare(int x) {

std::this\_thread::sleep\_for(std::chrono::seconds(2)); // simulate heavy work

return x \* x;

}

int main() {

std::cout << "Launching async task...\n";

std::future<int> result = std::async(std::launch::async, computeSquare, 10);

std::cout << "Doing something else while task runs...\n";

// Wait and get the result

int value = result.get();

std::cout << "Result: " << value << "\n";

}

✅ Key point: std::async + future makes concurrency *declarative* and *safe*.

**2. std::future**

* Represents a value that will be available later (from async task or promise).
* future.get() → blocks until the result is ready (can throw exceptions if task failed).

std::future<int> f = std::async(std::launch::async, []{ return 42; });

int result = f.get(); // blocks until ready

**3. std::promise**

* Lets you explicitly set a value or exception from one thread, to be consumed in another via std::future.

**Example:**

#include <iostream>

#include <thread>

#include <future>

void producer(std::promise<int> p) {

std::this\_thread::sleep\_for(std::chrono::seconds(2));

p.set\_value(100); // send data to consumer

}

void consumer(std::future<int> f) {

std::cout << "Waiting for value...\n";

int val = f.get(); // blocks until producer sets value

std::cout << "Got value: " << val << "\n";

}

int main() {

std::promise<int> p;

std::future<int> f = p.get\_future();

std::thread t1(producer, std::move(p));

std::thread t2(consumer, std::move(f));

t1.join();

t2.join();

}

**🔑 Comparison**

| **Feature** | **std::async** | **std::future** | **std::promise** |
| --- | --- | --- | --- |
| Launches tasks | ✅ | ❌ | ❌ |
| Holds results | ✅ (from async) | ✅ | ✅ (via promise) |
| Sets results | ❌ | ❌ | ✅ |
| Exception transfer | ✅ | ✅ | ✅ |

**4. Real-World Usage**

* std::async → Use when you just want to run a function in background and get result later.
* std::promise → Use when producer-consumer pattern is needed across threads.
* std::future → Unified handle to retrieve results (from both async & promise).

⚡ In **C++20**, you also get std::shared\_future (allowing multiple consumers) and std::when\_all, std::when\_any (via concurrency TS / libraries).

**6. Templates and Metaprogramming**

Template basics: function, class, and variadic templates  
SFINAE, decltype, auto, and type traits - ***Substitution Failure Is Not An Error***  
constexpr and compile-time computation  
Modern alternatives: Concepts, type deduction

**Templates and Metaprogramming in Modern C++**

**1. Introduction to Templates**

Templates allow writing **generic code** that works with any data type, avoiding duplication.

**Function Template**

#include <iostream>

using namespace std;

template <typename T>

T add(T a, T b) {

return a + b;

}

int main() {

cout << add(3, 5) << endl; // int

cout << add(2.5, 4.7) << endl; // double

}

👉 The compiler generates type-specific versions (int add, double add, etc.) during compilation (called **template instantiation**).

**2. Class Templates**

Class templates allow data structures and classes to be type-independent.

template <typename T>

class Box {

T value;

public:

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

T get() const { return value; }

};

int main() {

Box<int> b1(10);

Box<string> b2("Hello");

cout << b1.get() << " " << b2.get() << endl;

}

**3. Template Specialization**

Sometimes we want different behavior for specific types.

template <typename T>

class Printer {

public:

void print(T value) {

cout << value << endl;

}

};

// Specialization for bool

template <>

class Printer<bool> {

public:

void print(bool value) {

cout << (value ? "true" : "false") << endl;

}

};

int main() {

Printer<int> p1;

p1.print(42);

Printer<bool> p2;

p2.print(true); // Prints "true" instead of "1"

}

**4. Variadic Templates (C++11+)**

Allow functions/classes to take a variable number of template arguments.

#include <iostream>

using namespace std;

void print() { } // Base case

template <typename T, typename... Args>

void print(T first, Args... rest) {

cout << first << " ";

print(rest...);

}

int main() {

print(1, 2.5, "Hello", 'A');

}

👉 Expands recursively → print(1, 2.5, "Hello", 'A');

**5. SFINAE (Substitution Failure Is Not An Error)**

Allows **function overloading based on template type validity**.

#include <type\_traits>

#include <iostream>

using namespace std;

template <typename T>

typename enable\_if<is\_integral<T>::value>::type

process(T val) {

cout << "Integral: " << val << endl;

}

template <typename T>

typename enable\_if<is\_floating\_point<T>::value>::type

process(T val) {

cout << "Floating: " << val << endl;

}

int main() {

process(42); // Integral

process(3.14); // Floating

}

**6. Template Metaprogramming (TMP)**

Using templates to perform **compile-time computations**.

**Example: Factorial at Compile-Time**

#include <iostream>

using namespace std;

template <int N>

struct Factorial {

static const int value = N \* Factorial<N-1>::value;

};

template <>

struct Factorial<0> {

static const int value = 1;

};

int main() {

cout << Factorial<5>::value << endl; // 120

}

👉 Computed **at compile-time**, no runtime overhead.

**7. constexpr and Template Metaprogramming (C++11+)**

C++11+ introduced constexpr, making TMP easier and more readable.

constexpr int factorial(int n) {

return n <= 1 ? 1 : n \* factorial(n-1);

}

int main() {

constexpr int val = factorial(5); // Evaluated at compile-time

std::cout << val << std::endl;

}

**8. Concepts (C++20)**

Replaces complex SFINAE with cleaner syntax.

#include <concepts>

#include <iostream>

using namespace std;

template <std::integral T>

T add(T a, T b) {

return a + b;

}

int main() {

cout << add(3, 4) << endl; // OK

// cout << add(3.5, 4.2) << endl; // ERROR: not integral

}

**✅ Real-World Applications**

1. **STL Containers** → vector<int>, map<string, double> are template-based.
2. **Smart Pointers** → std::unique\_ptr<T> and std::shared\_ptr<T>.
3. **Generic Algorithms** → std::sort, std::find\_if.
4. **Compile-time Optimizations** → metaprogramming avoids runtime overhead.
5. **Policy-based Design** → flexible, reusable libraries (Boost, Eigen, etc.).

**1. Function Templates**

Function templates allow writing a function **independent of data type**.

🔹 Example:

#include <iostream>

using namespace std;

// Function template

template <typename T>

T add(T a, T b) {

return a + b;

}

int main() {

cout << add<int>(3, 4) << endl; // Explicit instantiation

cout << add(2.5, 3.5) << endl; // Implicit type deduction

}

✅ Compiler generates versions (add<int>, add<double>) automatically.  
✅ Eliminates code duplication.

**2. Class Templates**

Class templates allow creating **type-independent classes**.

🔹 Example: A generic stack

#include <iostream>

#include <vector>

using namespace std;

template <typename T>

class Stack {

private:

vector<T> elements;

public:

void push(const T& item) { elements.push\_back(item); }

void pop() { elements.pop\_back(); }

T top() const { return elements.back(); }

bool empty() const { return elements.empty(); }

};

int main() {

Stack<int> intStack;

intStack.push(10);

intStack.push(20);

cout << intStack.top() << endl; // 20

Stack<string> strStack;

strStack.push("Hello");

strStack.push("World");

cout << strStack.top() << endl; // World

}

✅ Same class works for **any type**.  
✅ STL containers (like vector, map, set) are class templates.

**3. Variadic Templates (C++11 onwards)**

They allow functions/classes to accept **arbitrary number of arguments**.

🔹 Example: Print multiple arguments

#include <iostream>

using namespace std;

// Base case

void print() { cout << endl; }

// Variadic template

template<typename T, typename... Args>

void print(T first, Args... args) {

cout << first << " ";

print(args...); // Recursive expansion

}

int main() {

print(1, 2.5, "Hello", 'A');

// Output: 1 2.5 Hello A

}

🔹 Example: Variadic constructor

template<typename... Args>

class MyClass {

public:

MyClass(Args... args) {

// do something with args...

cout << "Created with " << sizeof...(args) << " arguments\n";

}

};

int main() {

MyClass<int, double, string> obj(10, 3.14, "test");

}

✅ Reduces boilerplate.  
✅ Core to libraries like std::tuple, std::make\_shared, etc.

**🚀 Summary**

* **Function Templates** → type-independent functions.
* **Class Templates** → generic data structures/classes.
* **Variadic Templates** → handle **arbitrary arguments**, essential for modern metaprogramming.

**1. 🔹 SFINAE (Substitution Failure Is Not An Error)**

* A **core principle** in C++ templates.
* When the compiler substitutes template parameters and finds an invalid type/expression → **it doesn’t cause a hard error**, instead the compiler simply removes that template from the overload set.
* Used heavily in **template specialization** and **type trait libraries**.

**Example:**

#include <iostream>

#include <type\_traits>

// Function only enabled if T is integral

template<typename T>

typename std::enable\_if<std::is\_integral<T>::value, void>::type

print(T t) {

std::cout << "Integral: " << t << "\n";

}

// Function only enabled if T is floating point

template<typename T>

typename std::enable\_if<std::is\_floating\_point<T>::value, void>::type

print(T t) {

std::cout << "Floating-point: " << t << "\n";

}

int main() {

print(42); // calls integral version

print(3.14); // calls floating-point version

// print("Hello"); // error, no matching overload

}

✅ **Impact**: Enables **concepts-like behavior** (before C++20 introduced actual concepts).

**2. 🔹 decltype**

* Introduced in **C++11**.
* Used to **deduce the type** of an expression at compile time.
* Often used in templates to determine return types.

**Example:**

#include <iostream>

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

int main() {

decltype(add(1, 2)) x = 10; // deduces to int

std::cout << "x is of type int with value: " << x << "\n";

}

✅ Useful when working with **generic code** where type deduction is tricky.

**3. 🔹 auto**

* Introduced in **C++11** → allows **type inference**.
* Compiler deduces type from initializer.
* Works seamlessly with templates.

**Example:**

#include <iostream>

#include <vector>

int main() {

auto x = 42; // int

auto y = 3.14; // double

auto z = "Hello"; // const char\*

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

for (auto val : v) { // range-based for with auto

std::cout << val << " ";

}

}

✅ Makes code **cleaner** and reduces duplication.

**4. 🔹 Type Traits (std::type\_traits)**

* Introduced in **C++11**.
* A **compile-time toolkit** to query and modify types.
* Often used with **SFINAE** and **metaprogramming**.

**Common Type Traits:**

* **Type checks**: std::is\_integral<T>, std::is\_floating\_point<T>
* **Type modifications**: std::remove\_reference<T>, std::add\_const<T>
* **Compile-time constants**: std::integral\_constant, std::true\_type, std::false\_type

**Example:**

#include <iostream>

#include <type\_traits>

template<typename T>

void checkType(T) {

if constexpr (std::is\_integral<T>::value) {

std::cout << "Integral type\n";

} else if constexpr (std::is\_floating\_point<T>::value) {

std::cout << "Floating-point type\n";

} else {

std::cout << "Other type\n";

}

}

int main() {

checkType(10); // Integral

checkType(3.14); // Floating-point

checkType("Hi"); // Other type

}

✅ Enables **static polymorphism** and **compile-time optimizations**.

**🚀 Real-World Impact**

1. **SFINAE + Type Traits** → Safer generic libraries (used in STL & Boost).
2. **decltype + auto** → Cleaner APIs, less error-prone code, modern style.
3. Enabled **compile-time introspection**, forming the foundation for **C++20 concepts**.
4. Huge boost to **generic programming, metaprogramming, and template libraries**.

**1. What is constexpr?**

* constexpr means **the expression can be evaluated at compile time**, provided all inputs are known at compile time.
* Functions and variables marked constexpr are eligible for compile-time evaluation, but they **can still execute at runtime** if needed.

constexpr int square(int x) {

return x \* x;

}

int main() {

constexpr int result = square(5); // ✅ evaluated at compile time

int y = 10;

int runtime\_result = square(y); // ✅ computed at runtime

}

**🔹 2. constexpr Variables**

* A constexpr variable must be initialized with a compile-time constant.
* Safer alternative to macros (#define).

constexpr int maxSize = 100; // ✅ compile-time constant

const int minSize = 10; // could be runtime constant

**🔹 3. constexpr Functions**

* Must contain **only one return statement** (C++11).
* Since C++14, they can have **loops, if-statements, and multiple returns**.
* Since C++20, even **try/catch blocks** are allowed in constexpr.

Example: factorial at compile time

constexpr int factorial(int n) {

return (n <= 1) ? 1 : (n \* factorial(n - 1));

}

int main() {

constexpr int fact5 = factorial(5); // ✅ computed at compile time (120)

}

**🔹 4. constexpr with User-Defined Types**

You can create classes with constexpr constructors:

struct Point {

int x, y;

constexpr Point(int a, int b) : x(a), y(b) {}

};

int main() {

constexpr Point p1(10, 20); // ✅ compile-time object

}

**🔹 5. Compile-Time Computation with constexpr**

Examples where constexpr shines:

* Array sizes known at compile-time.
* Optimizing expensive math functions.
* Safe metaprogramming.

constexpr int fibonacci(int n) {

return (n <= 1) ? n : fibonacci(n - 1) + fibonacci(n - 2);

}

int main() {

constexpr int fib10 = fibonacci(10); // ✅ 55 at compile time

int fib\_n = fibonacci(20); // ✅ runtime if n not constexpr

}

**🔹 6. consteval vs constexpr (C++20)**

* constexpr → may be evaluated at compile time (if possible).
* consteval → **must** be evaluated at compile time.

consteval int cube(int x) {

return x \* x \* x;

}

int main() {

constexpr int val = cube(3); // ✅ ok

// int n = 5;

// int res = cube(n); // ❌ error (must be compile time)

}

**🔹 7. Real-World Impact**

* **Performance**: No runtime overhead for repeated computations.
* **Safety**: Errors caught during compilation.
* **Template metaprogramming**: Enables advanced compile-time logic without ugly tricks.
* **Embedded systems**: Critical for resource-constrained environments.

✅ In summary:

* Use constexpr for functions/variables you want evaluated at compile time.
* Use consteval when it **must** be compile-time.
* Combine with templates for **powerful compile-time programming**.

**1. Concepts**

Concepts allow you to **constrain templates** — making them easier to read, debug, and maintain.

**Without Concepts (Pre-C++20)**

#include <iostream>

#include <type\_traits>

template <typename T>

typename std::enable\_if<std::is\_integral<T>::value, bool>::type

is\_even(T n) {

return n % 2 == 0;

}

* Hard to read.
* Errors show long SFINAE messages.

**With Concepts (C++20)**

#include <iostream>

#include <concepts>

bool is\_even(std::integral auto n) { // requires integral types

return n % 2 == 0;

}

int main() {

std::cout << is\_even(4) << "\n"; // ✅ works

// std::cout << is\_even(3.14); // ❌ compile-time error

}

✨ Benefits:

* Cleaner syntax.
* Compiler errors are **human-readable**.
* Expressive constraints like std::integral, std::floating\_point, or custom concepts.

**🔹 2. Custom Concepts**

You can define your own constraints:

#include <concepts>

#include <iostream>

template<typename T>

concept Addable = requires(T a, T b) {

{ a + b } -> std::convertible\_to<T>; // must support +

};

template<Addable T>

T add(T a, T b) {

return a + b;

}

int main() {

std::cout << add(2, 3) << "\n"; // ✅ int works

// std::cout << add("hi", "bye"); // ❌ strings don't work

}

**🔹 3. Type Deduction**

Modern C++ simplifies code using **auto** and template argument deduction.

**Example: Old Style**

std::pair<int, double> p(10, 3.14);

**With Deduction (C++17+)**

std::pair p(10, 3.14); // type deduced as pair<int, double>

**Deduction with Functions**

template<typename T, typename U>

auto multiply(T a, U b) {

return a \* b; // return type deduced

}

auto result = multiply(3, 2.5); // deduces double

**🔹 4. Concepts + Deduction Together**

#include <concepts>

#include <iostream>

auto divide(std::integral auto a, std::integral auto b) {

return static\_cast<double>(a) / b;

}

int main() {

std::cout << divide(10, 2) << "\n"; // ✅ works

// std::cout << divide(10, 2.5); // ❌ compile-time error

}

* Here std::integral auto restricts parameters to integers.
* auto ensures **type deduction** for return type.

**🔹 5. Why Concepts & Deduction Matter in Real-World C++**

✅ Makes templates **self-documenting** (no need for verbose SFINAE).  
✅ **Better compiler errors** — quicker debugging.  
✅ Encourages **generic programming** (algorithms reusable across types).  
✅ Improves **code safety** by preventing misuse (e.g., dividing strings).  
✅ Matches **industry best practices** in libraries like C++20 STL, ranges, and coroutines.

**7. Optimizing Code Performance**

Identifying performance bottlenecks  
Value vs reference semantics, copy elision, move semantics  
Inlining, loop unrolling, memory alignment  
Cache-awareness and memory access patterns  
Compiler optimization flags and profiling

**1. General Performance Guidelines**

* **Write clean code first, optimize later** – readability > premature micro-optimizations.
* **Measure performance before optimizing** → Use profilers (e.g., gprof, perf, valgrind, Visual Studio Profiler).
* **Prefer algorithmic improvements** → O(n log n) → O(n) gives far more benefit than minor tweaks.

**🔹 2. Compiler-Level Optimizations**

* Enable optimizations:
  + GCC/Clang → -O2, -O3, -Ofast
  + MSVC → /O2
* Use **link-time optimization (LTO)**: -flto (GCC/Clang)
* Inline small functions with inline or let the compiler auto-inline.

**🔹 3. Memory Management**

* **Stack vs Heap**
  + Prefer **stack allocation** (faster, automatic cleanup).
  + Use heap only for large/long-lived objects.
* **Avoid memory leaks** → use smart pointers (std::unique\_ptr, std::shared\_ptr).
* **Cache locality**
  + Use **contiguous containers** (std::vector) over std::list when possible.
  + Minimize pointer chasing.
  + Example: Iterating a vector<int> is much faster than a list<int> due to CPU caching.

**🔹 4. Efficient Use of STL**

* Prefer **std::vector** as the default container.
* Use std::unordered\_map for average O(1) lookups vs std::map (O(log n)) when ordering is not needed.
* Use **reserve()** to preallocate capacity in vector to avoid repeated reallocations.
* Use emplace\_back() instead of push\_back() to avoid unnecessary copies.

**🔹 5. Move Semantics & Copy Elision**

* Implement **move constructors/assignment** for classes with resources.
* Use std::move when you no longer need an object.
* Example:
* std::vector<std::string> v;
* std::string s = "Hello";
* v.push\_back(std::move(s)); // moves instead of copies
* Compiler optimizations like **RVO (Return Value Optimization)** eliminate unnecessary copies.

**🔹 6. Avoid Expensive Operations**

* Minimize **virtual calls** in performance-critical loops (use CRTP or function objects).
* Avoid unnecessary **string concatenations** → use std::ostringstream or reserve.
* Use **references** instead of passing large objects by value.

**🔹 7. Parallelism & Concurrency**

* Use **multi-threading** (std::thread, std::async) to utilize multiple cores.
* Use **parallel STL algorithms** (std::for\_each(std::execution::par, ...) in C++17).
* Minimize **false sharing** by aligning thread-local data.

**🔹 8. Compile-Time Computation**

* Use **constexpr** functions for values known at compile-time.
* Prefer **templates and type traits** for metaprogramming over runtime computations.
* Example:
* constexpr int factorial(int n) {
* return (n <= 1) ? 1 : (n \* factorial(n - 1));
* }
* int arr[factorial(5)]; // computed at compile time

**🔹 9. Real-World Performance Tips**

✅ Profile before optimizing – don’t guess.  
✅ Prefer STL containers & algorithms – they’re highly optimized.  
✅ Favor **data-oriented design** (pack data for cache).  
✅ Use **RAII & smart pointers** – eliminates leaks while keeping performance.  
✅ Apply **parallelism** only where workload is CPU-heavy.

⚡ **Example: Optimized vs Non-Optimized**

❌ Bad:

std::vector<int> v;

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

v.push\_back(i); // reallocates many times

✅ Good:

std::vector<int> v;

v.reserve(100000); // allocate once

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

v.push\_back(i);

**1. What Are Performance Bottlenecks?**

A **bottleneck** is a part of your code that limits overall performance. Even if most of the program runs fast, one inefficient section can drastically slow everything down.

For example:

* A loop with unnecessary computations.
* Excessive memory allocations/deallocations.
* Synchronization contention in multithreading.
* Inefficient algorithms (O(n²) instead of O(n log n)).

**🛠️ 2. Common Areas Where Bottlenecks Occur**

1. **CPU-bound issues**
   * Heavy computations (e.g., nested loops, unoptimized math).
   * Inefficient algorithms/data structures.
   * Excessive branching or function calls.
2. **Memory-bound issues**
   * Cache misses (poor data locality).
   * Too many allocations/deallocations (new/delete, malloc/free).
   * Fragmentation of heap memory.
3. **I/O-bound issues**
   * Disk or network latency.
   * Blocking system calls.
   * Inefficient file/stream handling.
4. **Concurrency issues**
   * Lock contention (std::mutex overuse).
   * False sharing (two threads modifying different variables in the same cache line).
   * Poor task granularity in parallel execution.

**🧩 3. Methods to Identify Bottlenecks**

**✅ Profiling Tools**

* **Linux**: gprof, perf, valgrind (cachegrind, callgrind).
* **Windows**: Visual Studio Profiler.
* **Cross-platform**: Google Perf Tools, Intel VTune, Tracy, Instruments (macOS).

These tools give:

* CPU hotspots (which functions consume most time).
* Memory usage patterns.
* Cache performance.

**✅ Code Instrumentation**

* Add **timing measurements** using:
* #include <chrono>
* auto start = std::chrono::high\_resolution\_clock::now();
* // function or loop
* auto end = std::chrono::high\_resolution\_clock::now();
* std::cout << "Elapsed: "
* << std::chrono::duration\_cast<std::chrono::microseconds>(end - start).count()
* << " µs\n";
* Use **logging** to track frequency of expensive operations.

**✅ Algorithmic Complexity Analysis**

* Before writing code, analyze:
  + Is your algorithm O(n), O(n log n), or worse?
  + Can it be replaced with STL algorithms (std::sort, std::unordered\_map)?

**✅ Memory Tracing**

* Use tools like **Valgrind (Massif)** or **AddressSanitizer** for:
  + Heap usage.
  + Memory leaks.
  + Allocations per second.

**🚀 4. Guidelines to Fix Bottlenecks**

* Profile first, **don’t guess**.
* Fix the **largest bottleneck first** (80/20 rule: 20% of code often causes 80% slowdown).
* Replace naive algorithms with STL or well-known efficient ones.
* Improve **data locality** (use contiguous containers like std::vector over std::list).
* Minimize synchronization in multithreaded code.
* Cache results of expensive computations.

✅ **Summary:**  
To identify bottlenecks in C++, use **profiling tools, manual timing, and algorithmic analysis**. Always measure first, fix what matters, and prefer STL or proven libraries for optimized performance.

**Value vs Reference Semantics, Copy Elision, Move Semantics in C++**

**1. Value Semantics**

* **Definition:** When an object is passed/assigned by **value**, a copy of the object is made.
* **Characteristics:**
  + Each object has its own copy of the data.
  + Modifying one does not affect another.
  + Safe but may be expensive if the object is large.
* **Example:**
* struct Data { int x; };
* void foo(Data d) { d.x = 20; } // Copy occurs
* int main() {
* Data a{10};
* foo(a); // Pass by value (copies `a`)
* std::cout << a.x; // Prints 10 (unchanged)
* }

**2. Reference Semantics**

* **Definition:** When an object is passed/assigned by **reference**, no copy is made. Instead, multiple references point to the same underlying object.
* **Characteristics:**
  + More efficient (no copy).
  + Mutations affect all references.
* **Example:**
* void foo(Data& d) { d.x = 20; }
* int main() {
* Data a{10};
* foo(a); // Pass by reference
* std::cout << a.x; // Prints 20 (changed!)
* }
* **Const Reference:** Often used for efficiency when you don’t want modification.
* void printData(const Data& d) { std::cout << d.x; }

**3. Copy Elision**

* **Definition:** Compiler optimization where temporary copies are omitted.
* **Key Situations:**
  1. **Return Value Optimization (RVO):**
  2. Data createData() {
  3. return Data{42}; // Compiler may construct directly in caller’s space
  4. }
  5. **Named Return Value Optimization (NRVO):**
  6. Data createData() {
  7. Data d{42};
  8. return d; // Copy can be eliminated
  9. }
* **C++17:** Guarantees copy elision in many return cases (mandatory RVO).
* **Benefit:** Avoids unnecessary copies → big performance gains.

**4. Move Semantics**

* **Introduced in C++11** for efficiency when dealing with temporary resources.
* **Definition:** Transfer ownership of resources from one object to another instead of copying.
* **Key Mechanism:** T(T&&) (move constructor), operator=(T&&) (move assignment).
* **Example:**
* #include <iostream>
* #include <vector>
* struct MyData {
* std::vector<int> arr;
* // Move constructor
* MyData(MyData&& other) noexcept : arr(std::move(other.arr)) {
* std::cout << "Move constructor\n";
* }
* };
* int main() {
* MyData a{{1,2,3,4}};
* MyData b = std::move(a); // Move instead of copy
* // `a.arr` is now empty, `b.arr` owns the data
* }
* **Why Move?**
  + Copying large containers (like std::vector) is costly.
  + Move avoids deep copies by transferring ownership (pointers, file handles, sockets, etc.).

**✅ Summary Table**

| **Concept** | **Behavior** | **Cost** | **C++ Feature** |
| --- | --- | --- | --- |
| **Value Semantics** | Copy data into a new object | Expensive | Default copy constructor |
| **Reference Semantics** | Share access to the same object | Cheap | References, pointers |
| **Copy Elision** | Skip unnecessary temporary copies | Free | RVO, NRVO, mandatory in C++17 |
| **Move Semantics** | Transfer ownership instead of copying | Very Cheap | Move constructor, std::move |

👉 Together, these mechanisms give you fine-grained **control over performance vs safety** in modern C++.

**1. Inlining**

Inlining is the process of replacing a function call with the actual function body.  
This avoids the **function call overhead** (stack push/pop, jumps, etc.) and can enable **further optimizations** like constant folding and dead code elimination.

**Example:**

inline int square(int x) {

return x \* x;

}

int main() {

int result = square(5); // Compiler may replace with: int result = 5 \* 5;

}

✔️ **Pros**:

* Eliminates function call overhead.
* Can improve cache locality if small functions are inlined.
* Allows better compiler optimizations.

❌ **Cons**:

* Code bloat if large functions are inlined many times.
* Can negatively affect instruction cache.

👉 Rule of Thumb: Inline **small, frequently called functions** (getters, setters, mathematical ops). Let the compiler handle aggressive inlining with optimizations (-O2, -O3).

**🔹 2. Loop Unrolling**

Loop unrolling reduces the overhead of loop control (increment, compare, jump) by executing multiple iterations per loop body.

**Example (manual unrolling):**

// Normal loop

for (int i = 0; i < 8; i++) {

arr[i] \*= 2;

}

// Unrolled loop

for (int i = 0; i < 8; i += 4) {

arr[i] \*= 2;

arr[i+1] \*= 2;

arr[i+2] \*= 2;

arr[i+3] \*= 2;

}

✔️ **Pros**:

* Fewer branch instructions → better CPU pipeline efficiency.
* Can improve vectorization (SIMD).

❌ **Cons**:

* Code size increases (hurts cache).
* Diminishing returns for very large unrolling.

👉 Modern compilers (-O3) automatically unroll small loops, but manual unrolling may still help in performance-critical code.

**🔹 3. Memory Alignment**

Memory alignment ensures data is stored at addresses that match the **CPU’s word boundaries** (e.g., 4-byte, 8-byte). Misaligned data causes **extra CPU cycles** or even hardware exceptions.

**Example:**

#include <iostream>

#include <cstddef>

#include <new> // for alignas

struct alignas(16) AlignedData {

float x, y, z, w; // 16 bytes total, aligned to 16 bytes

};

int main() {

AlignedData data;

std::cout << "Alignment: " << alignof(AlignedData) << "\n";

}

✔️ **Pros**:

* Improves performance with SIMD instructions (SSE/AVX expect aligned data).
* Reduces false sharing in multithreading.

❌ **Cons**:

* Uses more memory due to padding.
* Not always beneficial if access patterns are irregular.

👉 Guidelines:

* Use alignas() for SIMD data structures.
* Use std::aligned\_alloc or std::pmr::monotonic\_buffer\_resource for aligned allocations.

**✅ Summary**

* **Inlining**: Reduce function call overhead, use for small hot functions.
* **Loop Unrolling**: Reduce branch overhead, improve vectorization.
* **Memory Alignment**: Ensure efficient memory access, especially for SIMD and parallel code.

🚀 Together, these techniques can drastically improve performance in high-performance computing, game engines, and embedded systems.

**Cache Awareness and Memory Access Patterns in C++**

Modern CPUs rely heavily on **cache memory** (L1, L2, L3) to bridge the speed gap between CPU and RAM. Writing cache-friendly C++ code can massively improve performance.

**1. 🔹 Why Cache Matters?**

* CPU clock cycle: ~0.5–1 ns
* L1 cache access: ~1 ns
* L2 cache access: ~3–4 ns
* L3 cache access: ~10–20 ns
* RAM access: **100–200 ns**
* Disk/SSD: ms scale

👉 If data fits in cache, it’s **100x faster** than main memory access.

**2. 🔹 Memory Access Patterns**

**(a) Spatial Locality**

* Accessing memory locations close to each other.
* Example: Iterating arrays sequentially is cache-friendly.

✅ Good (cache-friendly):

int arr[100000];

long long sum = 0;

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

sum += arr[i]; // Sequential → Prefetch-friendly

}

❌ Bad (cache-unfriendly):

int arr[100000];

long long sum = 0;

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

sum += arr[(i \* 64) % 100000]; // Jumps around → Cache misses

}

**(b) Temporal Locality**

* Reusing recently accessed data.
* Example: Caching results in a lookup table.

**(c) Data Layout**

* **AoS (Array of Structures)** vs. **SoA (Structure of Arrays)**.

🔴 Array of Structures (AoS):

struct Point { float x, y, z; };

Point points[100000];

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

points[i].x += 1.0f; // Accessing scattered memory

🟢 Structure of Arrays (SoA):

struct Points {

float x[100000], y[100000], z[100000];

};

Points points;

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

points.x[i] += 1.0f; // Continuous memory → cache-friendly

SoA is better when you work on only one field at a time.

**3. 🔹 Techniques for Cache-Friendly C++ Code**

✅ **Use contiguous containers**

* std::vector is better than std::list for cache locality.
* Linked lists scatter nodes across memory → many cache misses.

✅ **Minimize false sharing**

* Multiple threads writing to adjacent memory locations can thrash caches.
* Solution: **align memory** with alignas(64).

Example:

struct alignas(64) Counter {

std::atomic<int> value;

};

✅ **Loop tiling / blocking**

* Break large loops into smaller blocks to fit data into cache.
* Common in matrix multiplication.

for (int i = 0; i < N; i += block)

for (int j = 0; j < N; j += block)

for (int k = 0; k < N; k += block)

// Work on a block that fits into cache

✅ **Prefetching**

* Compilers and CPUs often prefetch automatically.
* In performance-critical code, you can use \_\_builtin\_prefetch() (GCC/Clang).

**4. 🔹 Example: Vector vs List**

#include <iostream>

#include <vector>

#include <list>

#include <chrono>

int main() {

const int N = 1'000'000;

std::vector<int> vec(N, 1);

std::list<int> lst(N, 1);

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

long long sum = 0;

for (int v : vec) sum += v; // cache-friendly

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

std::cout << "Vector time: "

<< std::chrono::duration<double>(end - start).count() << "s\n";

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

sum = 0;

for (int v : lst) sum += v; // many cache misses

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

std::cout << "List time: "

<< std::chrono::duration<double>(end - start).count() << "s\n";

}

👉 std::vector is usually **10x+ faster** than std::list for traversal.

**5. 🔹 Guidelines**

* Prefer **contiguous memory** (std::vector, raw arrays).
* Use **SoA** if operating on one field frequently.
* Avoid linked lists for high-performance code.
* Align data to **cache line size (64 bytes)**.
* Consider **loop tiling** for matrix-heavy workloads.

**1. Compiler Optimization Flags**

Modern C++ compilers (GCC, Clang, MSVC) provide optimization levels that balance **compile-time vs run-time speed**.

**GCC / Clang**

* **-O0** → No optimization (default in debug). Faster compilation, easy debugging.
* **-O1** → Basic optimizations (dead code elimination, simple inlining).
* **-O2** → Aggressive optimizations without long compile times. (Loop unrolling, constant folding, better register allocation).
* **-O3** → Maximum optimizations, including vectorization and function inlining. May increase binary size.
* **-Ofast** → Like -O3, but disregards strict standards compliance (may break floating-point precision).
* **-Os** → Optimize for **size**.
* **-march=native** → Optimize for the local CPU (uses SIMD, AVX, SSE, etc.).

Example:

g++ -O3 -march=native -flto program.cpp -o program

* -flto = Link-time optimization (LTO) → whole-program optimization across compilation units.

**MSVC**

* **/Od** → Disable optimizations (debug).
* **/O1** → Minimize size.
* **/O2** → Maximize speed (common in release builds).
* **/Ox** → Full optimization.
* **/GL** → Whole-program optimization (similar to -flto).

**2. Profiling and Performance Tools**

Once optimized, you must **measure performance** to find bottlenecks.

**Linux / GCC / Clang**

* **gprof** → Classic profiler (basic).
* **perf** → Linux performance analysis tool (tracks CPU cycles, cache misses).
* **valgrind (callgrind)** → Detects memory leaks and profiles cache/memory usage.
* **google-perftools** → Sampling profiler from Google.

Example with gprof:

g++ -pg program.cpp -o program

./program

gprof program gmon.out > analysis.txt

**Windows / MSVC**

* **Visual Studio Profiler** → Built-in, graphical performance analysis.
* **Windows Performance Analyzer (WPA)** → For system-level bottlenecks.

**Cross-platform / Advanced**

* **Intel VTune** → Industry-grade profiler for HPC apps.
* **gperftools** → CPU + heap profiler.
* **Clang Sanitizers** → -fsanitize=address, -fsanitize=thread help debug issues.

**3. Best Practices with Flags & Profiling**

* Always build with **-O2** for production (safe, efficient).
* Use **-O3 only for performance-critical sections**, and test correctness.
* Use **-march=native** if distributing binaries is not required (it locks to your CPU).
* Profile first: Don’t guess the bottlenecks. Optimize only after locating hot spots.
* Use **constexpr** and **templates** to move work to compile-time when possible.
* Keep a **debug build** (-O0 -g) and a **release build** (-O2 -DNDEBUG).

✅ Example workflow:

1. Write clean, maintainable C++ code.
2. Compile with -O2 (or -O3 if profiling shows improvement).
3. Run profiler (perf, valgrind, VS Profiler).
4. Fix bottlenecks (e.g., cache misses, bad memory layout).
5. Re-test and validate correctness.

**8. Advanced Language Features**

Rvalue references and move semantics in depth  
decltype, noexcept, override, final, explicit  
Lambda captures and closures  
Coroutines  
Modules  
Structured bindings and deconstruction

**Advanced C++ Language Features**

**1. Rvalue References and Move Semantics (C++11)**

* **Rvalue references (T&&)** allow you to bind temporaries.
* Enables **move semantics**, avoiding deep copies for efficiency.
* Example:
* class Buffer {
* int\* data;
* size\_t size;
* public:
* Buffer(size\_t n) : size(n), data(new int[n]) {}
* ~Buffer() { delete[] data; }
* // Move constructor
* Buffer(Buffer&& other) noexcept
* : data(other.data), size(other.size) {
* other.data = nullptr;
* other.size = 0;
* }
* };

✅ **Benefit:** Performance optimization when working with large data.

**2. Perfect Forwarding & std::forward**

* Forwarding preserves **value category** (lvalue/rvalue).
* Used in **factory functions and template wrappers**.
* template <typename T, typename... Args>
* std::unique\_ptr<T> make\_object(Args&&... args) {
* return std::make\_unique<T>(std::forward<Args>(args)...);
* }

✅ **Benefit:** No redundant copies when forwarding arguments.

**3. Lambdas and Captures**

* Anonymous functions with captures.
* Captures: [=] by value, [&] by reference, [this], or explicit [x, &y].
* Example:
* auto add = [](int a, int b) { return a + b; };
* int offset = 5;
* auto f = [offset](int x) { return x + offset; };

✅ **Benefit:** Inline functional programming, predicates in STL algorithms.

**4. Ranges and Views (C++20)**

* Introduces std::ranges and **lazy evaluation pipelines**.
* #include <ranges>
* #include <iostream>
* #include <vector>
* int main() {
* std::vector<int> v = {1,2,3,4,5,6};
* auto evenSquares = v
* | std::ranges::views::filter([](int n){ return n % 2 == 0; })
* | std::ranges::views::transform([](int n){ return n\*n; });
* for (int n : evenSquares)
* std::cout << n << " ";
* }

✅ **Benefit:** Cleaner code, avoids unnecessary copies.

**5. Coroutines (C++20)**

* Support for **asynchronous programming**.
* co\_await, co\_yield, co\_return.
* Example:
* #include <coroutine>
* #include <iostream>
* struct Generator {
* struct promise\_type {
* int current;
* auto get\_return\_object() { return Generator{this}; }
* std::suspend\_always initial\_suspend() { return {}; }
* std::suspend\_always final\_suspend() noexcept { return {}; }
* std::suspend\_always yield\_value(int value) { current = value; return {}; }
* void return\_void() {}
* void unhandled\_exception() {}
* };
* promise\_type\* p;
* Generator(promise\_type\* p) : p(p) {}
* int next() { return p->current; }
* };
* Generator counter() {
* for (int i=0; i<3; i++) co\_yield i;
* }

✅ **Benefit:** Efficient async tasks (event loops, generators, networking).

**6. Modules (C++20)**

* Alternative to headers, avoiding **long compile times** & ODR issues.
* Example:
* // 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); }

✅ **Benefit:** Faster builds, better encapsulation.

**7. Structured Bindings (C++17)**

* Unpack tuples, pairs, structs.
* std::map<int, std::string> m = {{1,"A"}, {2,"B"}};
* for (auto& [key, value] : m) {
* std::cout << key << " => " << value << "\n";
* }

✅ **Benefit:** Cleaner, more readable code.

**8. Concepts & Constraints (C++20)**

* Restrict templates with **compile-time conditions**.
* template <typename T>
* concept Number = std::is\_arithmetic\_v<T>;
* template <Number T>
* T add(T a, T b) { return a + b; }

✅ **Benefit:** Clearer error messages, safer template code.

**9. Smart Pointers Enhancements**

* std::unique\_ptr, std::shared\_ptr, std::weak\_ptr.
* make\_shared and make\_unique prevent memory leaks.
* Example:
* auto ptr = std::make\_shared<int>(42);

✅ **Benefit:** Safer ownership semantics, avoid manual delete.

**10. Custom Allocators**

* For **performance-sensitive applications** (games, embedded systems).
* Example: std::pmr::polymorphic\_allocator in C++17.

✅ **Benefit:** Optimized memory usage.

**📌 Summary**

Advanced features give **power, safety, and performance**:

* **Efficiency:** Move semantics, coroutines, custom allocators.
* **Expressiveness:** Lambdas, ranges, structured bindings.
* **Safety:** Concepts, smart pointers, modules.

**Rvalue References and Move Semantics in Depth**

**1. What are Rvalues?**

* **Lvalue** → Has a persistent memory address, can appear on the left-hand side of =.
* int x = 10; // x is an lvalue
* **Rvalue** → A temporary object or literal, usually destroyed at the end of the expression.
* int y = x + 5; // (x+5) is an rvalue (a temporary)

Before C++11, you couldn’t bind temporaries (rvalues) to non-const references.

**2. Rvalue References (T&&)**

C++11 introduced **rvalue references** (&&) to allow functions to **accept temporaries** directly.

void process(int&& x) { // rvalue reference

std::cout << "Got temporary: " << x << "\n";

}

int main() {

process(5); // OK: 5 is an rvalue

int a = 10;

// process(a); // ERROR: lvalue cannot bind to int&&

}

👉 T&& binds **only to rvalues**.

**3. Motivation: Copying vs Moving**

* Copying large objects (like vectors, strings) is **expensive**.
* Often we just need to **transfer ownership** of resources (e.g., heap memory).

**Example**

#include <vector>

#include <iostream>

int main() {

std::vector<int> v1 = {1, 2, 3, 4, 5};

// Copy

std::vector<int> v2 = v1; // Deep copy O(n)

// Move

std::vector<int> v3 = std::move(v1); // O(1) - transfers ownership

}

After move, v1 is **valid but unspecified** (usually empty).

**4. Move Constructor & Move Assignment**

You define these when your class manages resources (Rule of Five).

class Buffer {

int\* data;

size\_t size;

public:

Buffer(size\_t n) : size(n), data(new int[n]) {}

~Buffer() { delete[] data; }

// Copy Constructor

Buffer(const Buffer& other) : size(other.size), data(new int[other.size]) {

std::copy(other.data, other.data + size, data);

}

// Move Constructor

Buffer(Buffer&& other) noexcept : size(other.size), data(other.data) {

other.data = nullptr; // release ownership

other.size = 0;

}

// Move Assignment

Buffer& operator=(Buffer&& other) noexcept {

if (this != &other) {

delete[] data; // clean up old

size = other.size;

data = other.data;

other.data = nullptr;

other.size = 0;

}

return \*this;

}

};

**5. std::move**

* Converts an **lvalue → rvalue reference**.
* It doesn’t actually move anything; it just tells the compiler you’re done with the object.

Buffer b1(100);

Buffer b2 = std::move(b1); // invokes move constructor

⚠️ After std::move(b1), b1 is in a valid but unspecified state.

**6. Perfect Forwarding & Universal References**

If you write a template with T&&, it can bind to both lvalues and rvalues (**forwarding reference**).

template<typename T>

void wrapper(T&& arg) {

process(std::forward<T>(arg)); // preserves value category

}

**7. Best Practices**

✅ Implement **move semantics** when your class manages dynamic resources.  
✅ Always =delete copy/move functions you don’t want.  
✅ Use noexcept in move constructors/assignments for optimal STL usage.  
✅ Use std::move when you’re done with an object and want to transfer ownership.  
❌ Don’t use std::move on objects you still need — it leaves them in unspecified states.

**1. decltype**

* **Purpose**: Used to query the **type** of an expression at compile time.
* Helps in **generic programming** where you want to deduce types.

✅ Example:

int x = 10;

decltype(x) y = 20; // y is int

const int& ref = x;

decltype(ref) anotherRef = x; // anotherRef is const int&

⚡ With functions:

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

decltype(add(1, 2.5)) result = 5; // result is int

**2. noexcept**

* Specifies that a function **does not throw exceptions**.
* Helps the compiler **optimize** (especially move operations).
* Makes intent clear.

✅ Example:

void foo() noexcept { // promises no exceptions

// safe code

}

void risky() { throw std::runtime\_error("Oops"); }

⚠️ noexcept can be **conditional**:

template <typename T>

void safeMove(T&& obj) noexcept(std::is\_nothrow\_move\_constructible<T>::value) {

T copy = std::move(obj);

}

**3. override**

* Ensures a function **overrides a virtual function** from a base class.
* Prevents bugs from signature mismatches.

❌ Without override (silent bug):

struct Base {

virtual void foo(int) {}

};

struct Derived : Base {

void foo(double) {} // not overriding, just overloading!

};

✅ With override:

struct Derived : Base {

void foo(int) override {} // compiler error if mismatched

};

**4. final**

* Prevents **further overriding** of a virtual function or **inheritance** of a class.

✅ Example with functions:

struct Base {

virtual void foo() final; // cannot override in derived classes

};

✅ Example with classes:

struct Sealed final {}; // cannot inherit from Sealed

**5. explicit**

* Prevents unintended **implicit conversions** in constructors and conversion operators.
* Always use explicit for **single-argument constructors** unless you want implicit conversion.

❌ Without explicit:

struct Foo {

Foo(int x) {} // implicit conversion allowed

};

Foo f = 10; // works, but may be unintended

✅ With explicit:

struct Foo {

explicit Foo(int x) {}

};

Foo f = 10; // ERROR

Foo f(10); // OK

**🔑 Summary**

| **Feature** | **Purpose** |
| --- | --- |
| decltype | Deduces expression type at compile-time |
| noexcept | Declares function won't throw (optimization + clarity) |
| override | Ensures correct overriding of virtual functions |
| final | Prevents further overriding/inheritance |
| explicit | Prevents unintended implicit conversions |

**Lambda Expressions Recap**

A **lambda** in C++ is an anonymous function (a function without a name).  
Syntax:

[capture\_list](parameters) -> return\_type {

// function body

};

* **[capture\_list]** → tells the lambda which variables from the surrounding scope it can use.
* **parameters** → like a normal function parameter list.
* **return\_type** → (optional, usually deduced).
* **body** → code executed when lambda is called.

**🔹 Closures**

A **closure** is the object generated by a lambda expression.

* In C++, every lambda creates a **unique class type** (closure type) with an operator() overload.
* This means lambdas are actually syntactic sugar for **functors (function objects)**.

Example:

auto f = [](int x) { return x \* 2; };

std::cout << f(5); // prints 10

Here, f is a closure (an object with an operator()).

**🔹 Capture Modes**

Lambdas can capture variables from their surrounding scope in different ways:

**1. Capture by Value (=)**

* Makes a copy of the variable inside the lambda.
* Safe from modifications outside the lambda.
* Default: [=] means capture all used variables by value.

int a = 10;

auto f = [=]() { return a + 5; };

std::cout << f(); // 15

a = 20;

std::cout << f(); // still 15 (captured by value)

**2. Capture by Reference (&)**

* Captures variable as a reference.
* Reflects changes in the outer variable.
* Default: [&] means capture all used variables by reference.

int a = 10;

auto f = [&]() { return a + 5; };

std::cout << f(); // 15

a = 20;

std::cout << f(); // 25 (changes reflected)

**3. Mixed Captures**

* You can specify individually:

int x = 10, y = 20;

auto f = [x, &y]() { return x + y; };

y = 30;

std::cout << f(); // 40 (x captured by value, y by reference)

**4. Capture by Move (C++14 and beyond)**

* Useful for **non-copyable types** like std::unique\_ptr.

auto ptr = std::make\_unique<int>(42);

auto f = [p = std::move(ptr)]() { return \*p; };

std::cout << f(); // 42

**5. Capture this**

* Captures the this pointer of a class.
* Since C++17, we can capture by value ([\*this]), meaning **copy of the object state**.

struct MyClass {

int val = 5;

auto getLambda() {

return [this]() { return val; }; // captures `this` pointer

}

};

MyClass obj;

auto f = obj.getLambda();

std::cout << f(); // 5

**🔹 Closures in Action**

Example: Using lambdas in **std::sort** with captures:

std::vector<int> v = {5, 2, 8, 1, 3};

int pivot = 4;

std::sort(v.begin(), v.end(), [pivot](int a, int b) {

return std::abs(a - pivot) < std::abs(b - pivot);

});

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

// Output: 3 5 2 1 8

Here:

* The lambda captured pivot by value.
* Sorting was done relative to pivot.

**🔹 When to Use Captures**

✅ Use **by value** for immutability & thread safety.  
✅ Use **by reference** for efficiency or when you need updates.  
✅ Use **by move** for resource-owning objects.  
✅ Use \**[this]* (C++17+) when you want a copy of the object inside the lambda.

⚡ So in short: **lambda captures let you bind context, and closures let you treat code as first-class objects.**

**What Are Coroutines?**

* A **coroutine** is a function that can **suspend execution** and later **resume** from where it left off.
* Unlike normal functions (which run to completion), coroutines support:
  + **co\_return** → returns a value (or ends the coroutine).
  + **co\_yield** → produces a value and suspends (like a generator).
  + **co\_await** → suspends until a task completes (async/await style).

**🔹 Coroutine Keywords**

1. **co\_return** → Ends the coroutine, optionally returning a value.
2. **co\_yield** → Produces values lazily (generator).
3. **co\_await** → Suspends coroutine until awaited task completes.

**🔹 Example 1: Generator (using co\_yield)**

#include <coroutine>

#include <iostream>

#include <optional>

template <typename T>

struct Generator {

struct promise\_type {

std::optional<T> current\_value;

Generator get\_return\_object() {

return Generator{std::coroutine\_handle<promise\_type>::from\_promise(\*this)};

}

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

std::suspend\_always yield\_value(T value) {

current\_value = value;

return {};

}

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

void return\_void() {}

void unhandled\_exception() { throw; }

};

std::coroutine\_handle<promise\_type> handle;

Generator(std::coroutine\_handle<promise\_type> h) : handle(h) {}

~Generator() { if (handle) handle.destroy(); }

bool next() {

if (!handle.done()) {

handle.resume();

}

return !handle.done();

}

T value() { return \*handle.promise().current\_value; }

};

Generator<int> count(int n) {

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

co\_yield i;

}

}

int main() {

auto gen = count(5);

while (gen.next()) {

std::cout << gen.value() << " ";

}

// Output: 1 2 3 4 5

}

✅ Works like a **Python generator** — producing values lazily.

**🔹 Example 2: Asynchronous Task (co\_await)**

#include <iostream>

#include <coroutine>

#include <thread>

#include <chrono>

struct Task {

struct promise\_type {

Task get\_return\_object() {

return Task{std::coroutine\_handle<promise\_type>::from\_promise(\*this)};

}

std::suspend\_never initial\_suspend() noexcept { return {}; }

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

void return\_void() {}

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

};

std::coroutine\_handle<promise\_type> handle;

Task(std::coroutine\_handle<promise\_type> h) : handle(h) {}

~Task() { if (handle) handle.destroy(); }

};

Task asyncWork() {

std::cout << "Task started...\n";

std::this\_thread::sleep\_for(std::chrono::seconds(2));

std::cout << "Task finished!\n";

co\_return;

}

int main() {

auto t = asyncWork();

std::cout << "Main continues while task runs.\n";

}

**🔹 Why Use Coroutines?**

* **Generators:** Produce sequences without storing all values in memory.
* **Async Programming:** Natural co\_await syntax (similar to C# / Python async).
* **Performance:** Avoids heap allocations in many cases, low overhead.

**🔹 Real-World Uses**

* **Networking:** Asynchronous I/O (Boost.Asio, cppcoro).
* **Game development:** Coroutine-driven event loops.
* **Pipelines:** Data stream processing.

**C++20 Modules**

**1. Why Modules?**

Traditionally, C++ uses **headers (.h)** and **source files (.cpp)**:

* The **preprocessor** copies headers into every .cpp file using #include.
* This leads to **slow compilation**, **redundant parsing**, and **fragile dependencies** (macro collisions, ODR violations).

👉 **Modules replace headers** with a modern, explicit mechanism.

**2. Key Benefits**

* **Faster compilation** – parsed once, reused across TUs.
* **Better encapsulation** – can hide implementation details.
* **No textual inclusion** – avoids macro pollution.
* **Cleaner interfaces** – explicitly export what you want.

**3. Module Basics**

A module has:

1. **Module Interface Unit** – defines what is exported.
2. **Module Implementation Unit** – defines hidden details.
3. **Import** – replaces #include.

**✅ Example: Simple Module**

**math.ixx** (Module Interface Unit):

export module math; // declare module name

export int add(int a, int b) {

return a + b;

}

export int mul(int a, int b) {

return a \* b;

}

**main.cpp**:

import math; // use module

#include <iostream>

int main() {

std::cout << add(2, 3) << "\n"; // 5

std::cout << mul(4, 5) << "\n"; // 20

}

✅ No #include "math.h".  
✅ Only explicitly **exported** symbols are visible.

**4. Splitting Interface & Implementation**

**math.ixx**:

export module math;

export int add(int a, int b);

export int mul(int a, int b);

**math\_impl.cpp**:

module math; // implementation part

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

int mul(int a, int b) { return a \* b; }

👉 Keeps interface separate from implementation (like .h vs .cpp).

**5. Private Module Partitions**

You can split code into **partitions**:

**math\_add.ixx**:

export module math:add;

export int add(int a, int b);

**math\_mul.ixx**:

export module math:mul;

export int mul(int a, int b);

**math.ixx**:

export module math;

export import :add; // export partition

export import :mul;

👉 User just imports math, not each subpart.

**6. Importing Standard Library**

Some compilers already support **std library modules**.

import std; // import all of the standard library

import <vector>; // or selectively

Much faster than including <vector>.

**7. Restrictions**

* No macros are exported (better hygiene).
* Templates **can be exported**.
* Inline variables/functions **work fine**.
* Requires **compiler support** (MSVC, GCC ≥ 11, Clang ≥ 15).

**8. Compiling Modules**

Compilation is slightly different:

# Compile module interface

g++ -std=c++20 -c math.ixx -o math.o

# Compile implementation

g++ -std=c++20 -c main.cpp -o main.o

# Link

g++ math.o main.o -o main

MSVC (Visual Studio 2022) automates this with /interface.

✅ **Summary**:

* **Modules replace headers**.
* They provide **faster builds**, **better encapsulation**, and **explicit exports**.
* **Widely adopted in modern compilers**, and C++23 extends them further (e.g., std modularization).

**Structured Bindings in C++**

Structured bindings allow you to **unpack multiple values** from an object (like std::tuple, std::pair, or even a struct) into separate variables in one statement.

**✅ Syntax**

auto [a, b, c] = some\_tuple\_or\_struct;

* The variables (a, b, c) are automatically deduced.
* They act like **references** to the original values unless explicitly made auto (copy).

**🔹 Example 1: Unpacking a std::pair**

#include <iostream>

#include <map>

int main() {

std::map<int, std::string> m{{1, "Alice"}, {2, "Bob"}};

for (auto& [id, name] : m) { // Structured binding in for loop

std::cout << id << " -> " << name << "\n";

}

}

🔸 Instead of:

for (auto& kv : m) {

std::cout << kv.first << " -> " << kv.second << "\n";

}

**🔹 Example 2: Unpacking a std::tuple**

#include <tuple>

#include <iostream>

std::tuple<int, std::string, double> getPerson() {

return {25, "Alice", 55.7};

}

int main() {

auto [age, name, weight] = getPerson(); // Deconstruction

std::cout << name << " is " << age << " years old, weighs " << weight << "kg\n";

}

**🔹 Example 3: With a struct**

#include <iostream>

struct Point {

int x;

int y;

};

int main() {

Point p{10, 20};

auto [a, b] = p; // Deconstruction

std::cout << "x=" << a << ", y=" << b << "\n";

}

**🔹 Reference vs Copy**

Point p{10, 20};

auto [a, b] = p; // Copies values

auto& [x, y] = p; // References (changes modify original)

x = 50; // Modifies p.x

**🔹 Example 4: Returning multiple values without out params**

#include <tuple>

#include <iostream>

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

return {a / b, a % b}; // quotient, remainder

}

int main() {

auto [q, r] = divide(10, 3);

std::cout << "Quotient: " << q << ", Remainder: " << r << "\n";

}

**🔹 Benefits**

✔ Cleaner, more readable code.  
✔ Eliminates boilerplate (first, second, get<0>...).  
✔ Great with STL algorithms, maps, and tuples.  
✔ Works with structured data (maps, tuples, structs).

⚡ Pro tip: Combine **structured bindings + if initializer (C++17)** for powerful idioms:

if (auto [it, inserted] = m.insert({1, "new"}); inserted) {

std::cout << "Inserted!\n";

} else {

std::cout << "Already exists: " << it->second << "\n";

}

**9. Linking and Binary Structure**

Lib vs DLL: Differences and use-cases  
Compiling/Loading/Linking Static Libraries (LIB)  
Compiling/Loading/Linking Dynamic Libraries (DLL)  
Best practices for cross-platform binary compatibility

**Linking and Binary Structure in C++**

**1. Compilation Stages Recap**

When you compile a C++ program:

1. **Preprocessing**  
   Expands macros, includes headers (#include), removes comments.
2. g++ -E file.cpp -o file.i
3. **Compilation**  
   Converts preprocessed C++ into assembly code.
4. g++ -S file.i -o file.s
5. **Assembly**  
   Assembler turns .s into machine code object files (.o).
6. g++ -c file.s -o file.o
7. **Linking**  
   Linker (ld) combines object files and libraries into final **executable** or **shared library**.
8. g++ file.o -o program

**2. Types of Linking**

**✅ Static Linking**

* Libraries (e.g., .a in Linux, .lib in Windows) are copied into the binary at link time.
* Pros: Single self-contained executable, no runtime dependencies.
* Cons: Larger binaries, harder updates.

g++ main.cpp libmath.a -o app

**✅ Dynamic Linking**

* Uses shared libraries (.so in Linux, .dll in Windows).
* Symbols resolved at runtime.
* Pros: Smaller binaries, updates don’t require recompilation.
* Cons: Requires shared library installed on system.

g++ main.cpp -lm # links to libm.so (math library)

**3. Symbol Resolution**

* Each function/variable is a **symbol** in object files.
* **Linker** matches **declarations** with **definitions**:
  + extern int x; → must resolve to a defined int x; in another TU.
* **One Definition Rule (ODR)**: Only one definition per program.

👉 Tools:

* nm file.o → shows symbol table.
* objdump -t file.o → detailed symbol info.

**4. Binary Structure (Executable Format)**

Different OSes have different formats:

* **ELF (Linux)**
* **PE (Windows)**
* **Mach-O (macOS)**

Typical binary sections:

* .text → executable code
* .data → initialized global/static data
* .bss → uninitialized global/static data
* .rodata → read-only data (e.g., string literals)
* .heap → dynamic allocations (new, malloc)
* .stack → function call frames, local vars

**5. Relocation**

* Object files can’t assume where symbols live in memory.
* **Relocation tables** tell linker how to patch addresses.
* Example: A function foo() call in one object is patched with its final address after linking.

**6. Dynamic Linking Internals**

When you run an executable with .so dependencies:

1. **Loader (ld.so / Windows loader)** loads shared libs.
2. **PLT (Procedure Linkage Table)** + **GOT (Global Offset Table)** are used for lazy binding.
   * First call → jumps to loader → resolves symbol → caches in GOT → later calls are direct.

**7. Common Linking Errors**

* **Undefined reference**  
  Forgot to link object/library.
* undefined reference to `foo(int)'
* **Multiple definitions**  
  Same symbol defined in multiple TUs. Fix with inline or static.
* **ABI mismatch**  
  Compiled with different compiler versions/settings.

**8. Best Practices**

* Prefer **dynamic linking** for large frameworks (Qt, Boost).
* Use **static linking** for small utilities or embedded.
* Organize libraries properly:
  + Public headers → include/
  + Object/library → lib/
* Always use **include guards / #pragma once**.
* Follow **ODR** to avoid surprises.

**Static Library (.lib / .a) vs Dynamic Library (.dll / .so)**

| **Feature** | **Static Library** | **Dynamic Library** |
| --- | --- | --- |
| **File Extension** | .lib (Windows), .a (Linux) | .dll (Windows), .so (Linux) |
| **Linking Type** | Linked **at compile/link time** | Linked **at runtime** |
| **Code Inclusion** | Code copied into executable | Code **loaded on demand** at runtime |
| **Binary Size** | Larger executable | Smaller executable |
| **Memory Usage** | Each program has its own copy of library code | Multiple programs can share the same library in memory |
| **Updates** | Requires recompiling program to update library | Update library without recompiling dependent programs |
| **Performance** | Slightly faster (no indirection) | Slight overhead due to dynamic lookup (PLT/GOT) |
| **Distribution** | Single executable contains everything | Must distribute executable + DLL/so files |
| **Visibility of Symbols** | All symbols linked into exe | Only exported symbols are accessible |

**🔹 Static Library Use-Cases**

* Small utilities or command-line tools.
* Embedded systems where **all code must be self-contained**.
* Situations where **deployment simplicity** is important (no external DLLs).

✅ Example:

g++ -c utils.cpp -o utils.o

ar rcs libutils.a utils.o

g++ main.cpp -L. -lutils -o app

**🔹 Dynamic Library Use-Cases**

* Large frameworks (Qt, Boost, OpenCV).
* Plugins, extensions, or modular architectures.
* Situations where **updates** or **memory sharing** are important.
* Games and GUI apps that load modules at runtime.

✅ Example (Linux .so):

g++ -fPIC -c utils.cpp -o utils.o

g++ -shared -o libutils.so utils.o

g++ main.cpp -L. -lutils -o app

LD\_LIBRARY\_PATH=. ./app

✅ Example (Windows .dll):

* Export functions using \_\_declspec(dllexport) in DLL.
* Import functions using \_\_declspec(dllimport) in client program.

**🔹 Key Differences in Practice**

1. **Compilation**
   * Static: Full linking happens at compile time → larger exe.
   * Dynamic: Only headers needed at compile time → runtime resolves addresses.
2. **Versioning**
   * Static: Changing the library requires recompilation of all dependent code.
   * Dynamic: Can upgrade the library independently, no recompilation needed.
3. **Memory**
   * Static: Each program has its own copy → higher memory usage.
   * Dynamic: Library code shared across programs → lower memory footprint.
4. **Symbol Visibility**
   * Static: All symbols linked in → potential conflicts.
   * Dynamic: Only exported symbols accessible → better encapsulation.

**🔹 When to Use What**

| **Scenario** | **Recommended** |
| --- | --- |
| Small utility / embedded tool | Static |
| Large framework / plugin architecture | Dynamic |
| Frequent updates needed | Dynamic |
| Performance-critical single program | Static |

**🔹 Quick Windows DLL Example**

// mathlib.h

#ifdef MATHLIB\_EXPORTS

#define MATHLIB\_API \_\_declspec(dllexport)

#else

#define MATHLIB\_API \_\_declspec(dllimport)

#endif

extern "C" MATHLIB\_API int add(int a, int b);

// mathlib.cpp

#include "mathlib.h"

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

// main.cpp

#include <iostream>

#include "mathlib.h"

int main() {

std::cout << add(3, 4); // 7

}

Compile:

cl /LD mathlib.cpp # produces mathlib.dll

cl main.cpp mathlib.lib # links against import library

✅ **Summary**

* **Static library:** fast, self-contained, larger exe, simpler distribution.
* **Dynamic library:** smaller exe, runtime flexibility, shared memory, upgradeable, requires library distribution.

**Static Libraries in C++**

A **static library** is a collection of **object files** (.obj or .o) bundled into a single file (.lib on Windows, .a on Linux).

* At **link time**, the linker copies only the used object code into the executable.
* No runtime dependency — everything becomes part of the final .exe.

**1. Step 1: Write Library Source Files**

Example: mathlib.cpp

// mathlib.cpp

#include "mathlib.h"

int add(int a, int b) {

return a + b;

}

int mul(int a, int b) {

return a \* b;

}

Header: mathlib.h

// mathlib.h

#pragma once

int add(int a, int b);

int mul(int a, int b);

**2. Step 2: Compile Object Files**

* Compile source files into object files **without linking**.

**Windows (MSVC)**

cl /c mathlib.cpp # produces mathlib.obj

**Linux (GCC/Clang)**

g++ -c mathlib.cpp -o mathlib.o

**3. Step 3: Create Static Library**

* Bundle object files into a library file.

**Windows (MSVC)**

lib /OUT:mathlib.lib mathlib.obj

**Linux (GCC/Clang)**

ar rcs libmathlib.a mathlib.o

* ar rcs → r: insert/replace, c: create, s: index table.

**4. Step 4: Link Static Library to Your Program**

* Include headers **at compile time**.
* Link the library **at link time**.

**Example: main.cpp**

#include <iostream>

#include "mathlib.h"

int main() {

std::cout << add(3, 5) << "\n"; // 8

std::cout << mul(4, 6) << "\n"; // 24

}

**Linking**

**Windows (MSVC)**

cl main.cpp mathlib.lib

**Linux (GCC/Clang)**

g++ main.cpp -L. -lmathlib -o app

* -L. → search library in current directory
* -lmathlib → link libmathlib.a

**5. Step 5: Run Executable**

./app # Linux

app.exe # Windows

✅ Output:

8

24

**6. Tips & Best Practices**

1. **Use #pragma once** or include guards in headers.
2. **Organize library**:
3. project/
4. ├─ include/ # headers
5. ├─ src/ # cpp files
6. ├─ lib/ # compiled .lib/.a
7. **Static library naming**:
   * Linux → libname.a
   * Windows → name.lib
8. **Recompile library** only when its source changes.
9. **Avoid exporting unnecessary symbols** to reduce binary size.

**7. Advanced Notes**

* Supports multiple object files:
* ar rcs libmathlib.a mathlib.o utils.o ...
* **Link order matters** in GCC: libraries that depend on others should appear **after** the dependent objects.
* Debugging symbols can be included or stripped using compiler flags (-g / -s).

**Dynamic Libraries in C++**

A **dynamic library** is a collection of object code that is **loaded at runtime** rather than being copied into the executable at compile time.

* Windows: .dll (Dynamic Link Library)
* Linux: .so (Shared Object)
* macOS: .dylib

Advantages:

* Smaller executable size
* Shared memory between programs
* Can update library without recompiling programs

**1. Step 1: Write Library Source**

**Example: mathlib.cpp**

#include "mathlib.h"

int add(int a, int b) {

return a + b;

}

int mul(int a, int b) {

return a \* b;

}

**Header: mathlib.h**

#pragma once

#ifdef \_WIN32

#ifdef MATHLIB\_EXPORTS

#define MATHLIB\_API \_\_declspec(dllexport)

#else

#define MATHLIB\_API \_\_declspec(dllimport)

#endif

#else

#define MATHLIB\_API

#endif

extern "C" MATHLIB\_API int add(int a, int b);

extern "C" MATHLIB\_API int mul(int a, int b);

* \_\_declspec(dllexport) → marks function for export in DLL
* \_\_declspec(dllimport) → tells compiler to import symbol
* extern "C" avoids C++ name mangling (important for linking)

**2. Step 2: Compile Dynamic Library**

**Windows (MSVC)**

cl /LD mathlib.cpp /FEmathlib.dll /DMATHLIB\_EXPORTS

* /LD → create DLL
* /DMATHLIB\_EXPORTS → define macro to export functions

**Linux (GCC)**

g++ -fPIC -shared mathlib.cpp -o libmathlib.so

* -fPIC → position-independent code
* -shared → create shared object

**3. Step 3: Use DLL in Client Program**

**Client Code: main.cpp**

#include <iostream>

#include "mathlib.h"

int main() {

std::cout << add(3, 5) << "\n"; // 8

std::cout << mul(4, 6) << "\n"; // 24

}

**4. Step 4: Link Against Library**

**Windows (MSVC)**

cl main.cpp mathlib.lib

* Compiler uses mathlib.lib (import library) to resolve symbols
* At runtime, mathlib.dll must be in the same folder or PATH

**Linux (GCC)**

g++ main.cpp -L. -lmathlib -o app

export LD\_LIBRARY\_PATH=. # runtime linker finds libmathlib.so

./app

**5. Step 5: Run Program**

✅ Output:

8

24

* Windows: app.exe + mathlib.dll
* Linux: ./app + libmathlib.so in LD\_LIBRARY\_PATH

**6. Dynamic Loading at Runtime (Optional)**

* **Windows:** LoadLibrary, GetProcAddress
* **Linux:** dlopen, dlsym

**Windows Example**

#include <windows.h>

#include <iostream>

typedef int(\*FuncPtr)(int, int);

int main() {

HINSTANCE hDll = LoadLibrary("mathlib.dll");

FuncPtr add = (FuncPtr)GetProcAddress(hDll, "add");

std::cout << add(2, 3) << "\n"; // 5

FreeLibrary(hDll);

}

**Linux Example**

#include <dlfcn.h>

#include <iostream>

int main() {

void\* handle = dlopen("./libmathlib.so", RTLD\_LAZY);

auto add = (int(\*)(int,int))dlsym(handle, "add");

std::cout << add(2,3) << "\n"; // 5

dlclose(handle);

}

**7. Best Practices**

* Use extern "C" to avoid C++ name mangling when exporting to DLLs.
* Use macros (MATHLIB\_API) for cross-platform builds.
* Keep DLL interface **stable**; internal changes won’t break programs if interface unchanged.
* Avoid global/static variables in DLL for thread safety.
* Always distribute DLL along with executable.

✅ Summary:

* **Static (.lib / .a):** linked at compile time, no runtime dependency.
* **Dynamic (.dll / .so):** linked at runtime, shared across processes, smaller executables, flexible updates.

**Best Practices for Cross-Platform Binary Compatibility**

**1. Use Standard Types**

* Avoid int, long, short when binary layout matters.
* Use **fixed-width types** from <cstdint>:
* #include <cstdint>
* int32\_t a; // guaranteed 32-bit
* uint64\_t b; // guaranteed 64-bit
* Ensures same size across platforms and compilers.

**2. Avoid Compiler-Specific Extensions**

* Minimize use of:
  + MSVC-specific keywords (\_\_declspec, \_\_forceinline)
  + GCC/Clang-specific attributes (\_\_attribute\_\_((packed)))
* Use **portable macros** for conditional compilation:

#ifdef \_WIN32

#define EXPORT \_\_declspec(dllexport)

#else

#define EXPORT \_\_attribute\_\_((visibility("default")))

#endif

**3. Use extern "C" for Library Interfaces**

* C++ name mangling varies between compilers → break ABI.
* Use extern "C" for exported functions to maintain stable symbols:

extern "C" EXPORT int add(int a, int b);

**4. Define Clear API / ABI Boundaries**

* Keep **class layout stable**:
  + Avoid virtual functions in interfaces exposed to other binaries unless necessary.
  + Avoid exposing STL containers (std::string, std::vector) across DLL boundaries.
* Prefer **Pimpl (Pointer to Implementation) idiom** to hide implementation details:

class MyClass {

private:

struct Impl;

Impl\* pImpl;

public:

MyClass();

~MyClass();

void doSomething();

};

**5. Be Careful With Packing & Alignment**

* Memory layout may differ:
* struct \_\_attribute\_\_((packed)) Data { int a; char b; };
* Use #pragma pack(push,1) / #pragma pack(pop) carefully; better to **serialize explicitly** for cross-platform communication.

**6. Use Portable Calling Conventions**

* Function calling conventions differ (Windows: \_\_stdcall, \_\_cdecl).
* Stick to **cdecl** by default or wrap in macros:

#ifdef \_WIN32

#define CALL\_CONV \_\_cdecl

#else

#define CALL\_CONV

#endif

extern "C" EXPORT int CALL\_CONV add(int a, int b);

**7. Avoid Platform-Specific Types**

* Pointers may have different sizes on 32-bit vs 64-bit systems.
* Avoid assumptions like sizeof(int) == sizeof(void\*).
* Use uintptr\_t / intptr\_t when storing pointers as integers.

**8. Version Your Libraries**

* Maintain **semantic versioning** to ensure clients know when ABI changes:
  + Breaking ABI → increment **major version**.
  + Minor additions → **minor version**.
* Use **symbol versioning** on Linux (\_\_attribute\_\_((visibility))) for shared libraries.

**9. Testing Across Compilers / Architectures**

* Test your library with:
  + GCC, Clang, MSVC
  + x86\_64, ARM
  + Debug vs Release builds
* Catch **alignment, padding, endianness** issues early.

**10. Serialization for Persistent Data**

* Never assume binary layout of structs is portable.
* Use explicit serialization (e.g., protobuf, flatbuffers) for:
  + Saving files
  + Network protocols

**11. Use CMake or Cross-Platform Build Systems**

* CMake can handle:
  + Compiler flags
  + Platform-specific defines
  + Library suffixes (.dll, .so, .dylib) automatically

Example:

add\_library(myLib SHARED mathlib.cpp)

target\_include\_directories(myLib PUBLIC ${CMAKE\_CURRENT\_SOURCE\_DIR}/include)

**✅ Summary**

* Stick to **standard types**, **C linkage**, and **explicit API boundaries**.
* Avoid exposing compiler-dependent features across binary boundaries.
* Always **test on all target platforms and architectures**.
* Consider **serialization or Pimpl** for cross-platform ABI stability.