Complete C++ Language Reference Guide

Table of Contents

Language Fundamentals

Basic Program Structure

Philosophy: C++ follows a structured approach where every program needs a main entry point and follows specific syntax rules.

Structure:

```cpp
#include <iostream>  // Preprocessor directive

int main() {          // Entry point
    std::cout << "Hello, World!" << std::endl;
    return 0;         // Exit status
}
```

Best Practices:

- Always return 0 from main() for successful execution
- Use meaningful names for functions and variables
- Include necessary headers only

Preprocessor Directives

Philosophy: Text processing before compilation begins, enabling conditional compilation and code inclusion.

Common Directives:

- `#include`: Include header files
- `#define`: Define macros
- `#ifdef/#ifndef`: Conditional compilation
- `#pragma`: Compiler-specific directives

Best Practices:

- Prefer const variables over #define for constants
- Use include guards or #pragma once to prevent multiple inclusions
- Minimize macro usage in favor of inline functions or templates

Namespaces

Philosophy: Organize code into logical groups and avoid name conflicts.

```
namespace MyNamespace {
    int value = 42;
    void function() { /* ... */ }
}

// Usage
using namespace MyNamespace;  // Import entire namespace
using MyNamespace::value;     // Import specific symbol
```

Best Practices:

- Avoid `using namespace std;` in header files
- Create meaningful namespace hierarchies
- Use anonymous namespaces for internal linkage

Data Types and Variables

Fundamental Types

Philosophy: C++ provides a rich set of built-in types optimized for different use cases.

Integer Types:

- `char`: Usually 8 bits, for characters
- `short`: At least 16 bits
- `int`: Natural word size, at least 16 bits
- `long`: At least 32 bits
- `long long`: At least 64 bits (C++11)

Floating Point Types:

- `float`: Single precision (typically 32 bits)
- `double`: Double precision (typically 64 bits)
- `long double`: Extended precision

Boolean Type:

- `bool`: true or false

Best Practices:

- Use `int` for general integer arithmetic
- Use `std::size_t` for array indices and sizes
- Use `std::int32_t` when exact size matters (from `<cstdint>`)

Type Modifiers

Signed/Unsigned:

```
signed int x;     // Can hold negative values
unsigned int y;   // Only non-negative values, larger positive range
```

Const Qualifier:

```
const int MAX_SIZE = 100;  // Cannot be modified
const int* ptr;           // Pointer to const int
int* const ptr2;          // Const pointer to int
const int* const ptr3;    // Const pointer to const int
```

Philosophy: Immutability by default improves code safety and optimization opportunities.

Best Practices:

- Use `const` whenever possible
- Prefer `const` references for function parameters
- Use `constexpr` for compile-time constants (C++11)

Variables and Initialization

Declaration vs Definition:

```
extern int x;         // Declaration (no storage allocated)
int x = 42;          // Definition (storage allocated and initialized)
```

Initialization Methods:

```
int a = 5;           // Copy initialization
int b(5);            // Direct initialization
int c{5};            // Uniform initialization (C++11)
int d{};             // Zero initialization
auto e = 5;          // Auto type deduction (C++11)
```

Best Practices:

- Always initialize variables
- Prefer uniform initialization `{}` for consistency
- Use `auto` when type is obvious from context

Storage Classes

Auto: Default storage class (local variables) **Static**:

- Local static: Retains value between function calls
- Global static: Internal linkage (file scope)

Extern: External linkage (can be accessed from other files) **Register**: Hint to store in CPU register (deprecated in C++17)

Thread_local: Each thread has its own copy (C++11)

Operators

Arithmetic Operators

Philosophy: Provide natural mathematical operations with expected precedence.

```
+ - * / %           // Basic arithmetic
++ --               // Increment/decrement (prefix/postfix)
+= -= *= /= %=      // Compound assignment
```

Best Practices:

- Prefer prefix increment/decrement for iterators
- Be aware of integer division truncation
- Use parentheses to clarify precedence

Comparison Operators

```
== != < > <= >=     // Comparison operators
<=> (C++20)         // Three-way comparison (spaceship operator)
```

Logical Operators

```
&& || !             // Logical AND, OR, NOT
& | ^ ~ << >>       // Bitwise operations
```

Best Practices:

- Use logical operators for boolean expressions
- Use bitwise operators for bit manipulation
- Understand short-circuit evaluation of && and ||

Assignment and Memory Operators

```
=                     // Assignment
&                     // Address-of
*                     // Dereference
new / delete          // Dynamic memory allocation/deallocation
new[] / delete[]      // Array versions
```

Operator Overloading

Philosophy: Allow user-defined types to work naturally with built-in operators.

```
class Vector {
public:
    Vector operator+(const Vector& other) const;
    Vector& operator+=(const Vector& other);
    bool operator==(const Vector& other) const;
    friend std::ostream& operator<<(std::ostream& os, const Vector& v);
};
```

Best Practices:

- Follow expected semantics (+ should not modify operands)
- Implement related operators consistently
- Use member functions for unary operators, non-member for binary when possible

---

Control Flow

Conditional Statements

If Statement:

```
if (condition) {
    // code
} else if (another_condition) {
    // code
} else {
    // code
}
```

Switch Statement:

```
switch (expression) {
    case value1:
        // code
        break;
    case value2:
        // code
        [[fallthrough]];  // C++17 attribute
    case value3:
        // code
        break;
    default:
        // code
}
```

Conditional Operator:

```
result = condition ? value_if_true : value_if_false;
```

Best Practices:

- Always use braces for if statements, even single statements
- Use switch for multiple discrete values
- Explicitly mark intentional fallthrough in switch statements

Loops

For Loop:

```
// Traditional for loop
for (int i = 0; i < n; ++i) {
    // code
}

// Range-based for loop (C++11)
for (const auto& element : container) {
    // code
}
```

While Loop:

```
while (condition) {
    // code
}
```

```
do {
    // code
} while (condition);
```

Best Practices:

- Use range-based for loops when possible
- Prefer prefix increment in loops
- Avoid infinite loops without clear exit conditions

Jump Statements

```
break;       // Exit loop or switch
continue;    // Skip to next iteration
return;      // Exit function
goto label; // Jump to label (generally discouraged)
```

Philosophy: Provide mechanism for exceptional control flow while maintaining structured programming principles.

Functions

Function Declaration and Definition

Philosophy: Functions encapsulate reusable code and provide abstraction boundaries.

```
// Declaration
int add(int a, int b);

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

Parameter Passing

By Value:

```
void func(int x) {  // Copy of argument
    x = 10;  // Original unchanged
}
```

By Reference:

```cpp
void func(int& x) {  // Reference to argument
    x = 10;  // Original modified
}
```

By Pointer:

```cpp
void func(int* x) {  // Pointer to argument
    if (x) *x = 10;  // Original modified if not null
}
```

Best Practices:

- Pass large objects by const reference
- Use references for output parameters
- Prefer references over pointers when null is not valid

Function Overloading

Philosophy: Same operation on different types should use the same name.

```cpp
int max(int a, int b);
double max(double a, double b);
std::string max(const std::string& a, const std::string& b);
```

Default Arguments

```cpp
void func(int x, int y = 0, int z = 1);
// Can be called as: func(5), func(5, 3), func(5, 3, 7)
```

Best Practices:

- Default arguments only in declarations, not definitions
- Place default arguments at the end of parameter list

Inline Functions

```cpp
inline int square(int x) {
    return x * x;
}
```

Philosophy: Hint to compiler to replace function call with function body for performance.

Lambda Expressions (C++11)

Philosophy: Anonymous functions for local use, especially with algorithms.

```
auto lambda = [capture](parameters) -> return_type {
    // body
};

// Examples
auto add = [](int a, int b) { return a + b; };
auto counter = [count = 0]() mutable { return ++count; };
```

Capture Modes:

- `[]`: No capture
- `[=]`: Capture by value
- `[&]`: Capture by reference
- `[x, &y]`: Capture x by value, y by reference

Object-Oriented Programming

Classes and Objects

Philosophy: Encapsulate data and behavior together, modeling real-world entities.

```
class Rectangle {
private:
    double width, height;

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

    // Member functions
    double area() const { return width * height; }
    double perimeter() const { return 2 * (width + height); }

    // Getters and setters
    double getWidth() const { return width; }
    void setWidth(double w) { if (w > 0) width = w; }
};
```

Access Specifiers

- `private`: Accessible only within the class
- `protected`: Accessible within class and derived classes
- `public`: Accessible from anywhere

Best Practices:

- Keep data members private
- Provide public interface through member functions
- Use protected for inheritance hierarchies

Constructors and Destructors

Default Constructor:

```
class MyClass {
public:
    MyClass() = default;  // Compiler-generated
    MyClass() : member(0) {}  // Custom
};
```

Parameterized Constructor:

```
MyClass(int value) : member(value) {}
```

Copy Constructor:

```
MyClass(const MyClass& other) : member(other.member) {}
```

Move Constructor (C++11):

```
MyClass(MyClass&& other) noexcept : member(std::move(other.member)) {}
```

Destructor:

```
~MyClass() {
    // Cleanup resources
}
```

Best Practices:

- Use member initialization lists
- Follow Rule of Zero/Three/Five
- Make destructors virtual for base classes

Inheritance

Philosophy: Model "is-a" relationships and enable code reuse.

```
class Animal {
protected:
    std::string name;
public:
    Animal(const std::string& n) : name(n) {}
    virtual void speak() const = 0;  // Pure virtual
    virtual ~Animal() = default;
};

class Dog : public Animal {
public:
    Dog(const std::string& n) : Animal(n) {}
    void speak() const override {
        std::cout << name << " says Woof!" << std::endl;
    }
};
```

Types of Inheritance:

- `public`: "is-a" relationship
- `protected`: Rarely used
- `private`: "implemented-in-terms-of" relationship

Polymorphism

Virtual Functions:

```
class Base {
public:
    virtual void func() { std::cout << "Base" << std::endl; }
    virtual ~Base() = default;
};

class Derived : public Base {
public:
    void func() override { std::cout << "Derived" << std::endl; }
};
```

Abstract Classes:

```
class Shape {
public:
    virtual double area() const = 0;  // Pure virtual
    virtual ~Shape() = default;
};
```

Best Practices:

- Use virtual destructors in base classes
- Use `override` keyword for clarity (C++11)
- Prefer composition over inheritance when possible

Operator Overloading in Classes

```
class Complex {
private:
    double real, imag;
public:
    Complex operator+(const Complex& other) const;
    Complex& operator+=(const Complex& other);
    bool operator==(const Complex& other) const;

    // Stream operators as friends
    friend std::ostream& operator<<(std::ostream& os, const Complex& c);
    friend std::istream& operator>>(std::istream& is, Complex& c);
};
```

Memory Management

Stack vs Heap

Stack Memory:

- Automatic allocation/deallocation
- Fast access
- Limited size
- LIFO order

Heap Memory:

- Manual allocation/deallocation
- Slower access
- Large capacity
- Random access

Dynamic Memory Allocation

Raw Pointers:

```
int* ptr = new int(42);          // Allocate single int
int* arr = new int[10];          // Allocate array
delete ptr;                      // Deallocate single object
delete[] arr;                    // Deallocate array
```

Problems with Raw Pointers:

- Memory leaks
- Double deletion
- Use after free
- Exception safety issues

Smart Pointers (C++11)

Philosophy: Automatic resource management through RAII (Resource Acquisition Is Initialization).

unique_ptr:

```cpp
std::unique_ptr<int> ptr = std::make_unique<int>(42);
std::unique_ptr<int[]> arr = std::make_unique<int[]>(10);
// Automatic cleanup, no copy, move only
```

shared_ptr:

```cpp
std::shared_ptr<int> ptr1 = std::make_shared<int>(42);
std::shared_ptr<int> ptr2 = ptr1;  // Reference count = 2
// Automatic cleanup when last reference goes out of scope
```

weak_ptr:

```cpp
std::weak_ptr<int> weak = ptr1;  // Non-owning reference
if (auto locked = weak.lock()) {  // Check if still valid
    // Use locked like shared_ptr
}
```

Best Practices:

- Prefer smart pointers over raw pointers for ownership
- Use `make_unique` and `make_shared`
- Use `weak_ptr` to break circular references

Memory Management Best Practices

- Follow RAII principles
- Use stack allocation when possible
- Prefer containers over raw arrays
- Use smart pointers for dynamic allocation
- Avoid naked new/delete

Templates

Function Templates

Philosophy: Write generic code that works with multiple types without sacrificing performance.

```cpp
template<typename T>
T max(const T& a, const T& b) {
    return (a > b) ? a : b;
}

// Usage
int x = max(5, 10);
double y = max(3.14, 2.71);
std::string z = max(std::string("hello"), std::string("world"));
```

Class Templates

```cpp
template<typename T>
class Stack {
private:
    std::vector<T> elements;

public:
    void push(const T& elem) {
        elements.push_back(elem);
    }

    T pop() {
        if (elements.empty()) {
            throw std::runtime_error("Stack is empty");
        }
        T elem = elements.back();
        elements.pop_back();
        return elem;
    }

    bool empty() const {
        std::lock_guard<std::mutex> lock(mtx);
        return queue.empty();
    }
};
```

Best Practices:

- Use RAII with lock guards
- Avoid deadlocks by acquiring locks in consistent order
- Prefer high-level abstractions over low-level primitives
- Use atomic operations for simple shared data
- Be aware of memory ordering and race conditions

Advanced Features

RAII (Resource Acquisition Is Initialization)

Philosophy: Tie resource lifetime to object lifetime, ensuring automatic cleanup.

```
class FileHandler {
private:
    std::FILE* file;

public:
    FileHandler(const std::string& filename) {
        file = std::fopen(filename.c_str(), "r");
        if (!file) {
            throw std::runtime_error("Failed to open file");
        }
    }

    ~FileHandler() {
        if (file) {
            std::fclose(file);
        }
    }

    // Delete copy operations to prevent double-close
    FileHandler(const FileHandler&) = delete;
    FileHandler& operator=(const FileHandler&) = delete;

    // Move operations
    FileHandler(FileHandler&& other) noexcept : file(other.file) {
        other.file = nullptr;
    }

    FileHandler& operator=(FileHandler&& other) noexcept {
        if (this != &other) {
            if (file) std::fclose(file);
            file = other.file;
            other.file = nullptr;
        }
        return *this;
    }

    std::FILE* get() const { return file; }
};
```

SFINAE (Substitution Failure Is Not An Error)

Philosophy: Enable conditional template instantiation based on type properties.

```cpp
#include <type_traits>

// Enable only for integral types
template<typename T>
typename std::enable_if<std::is_integral<T>::value, T>::type
process_integral(T value) {
    return value * 2;
}

// Enable only for floating point types
template<typename T>
typename std::enable_if<std::is_floating_point<T>::value, T>::type
process_floating(T value) {
    return value * 0.5;
}

// C++17 version using if constexpr
template<typename T>
auto process_modern(T value) {
    if constexpr (std::is_integral_v<T>) {
        return value * 2;
    } else if constexpr (std::is_floating_point_v<T>) {
        return value * 0.5;
    }
}
```

Template Metaprogramming

Compile-time Computation:

```cpp
// Recursive template for factorial
template<int N>
struct Factorial {
    static constexpr int value = N * Factorial<N-1>::value;
};

template<>
struct Factorial<0> {
    static constexpr int value = 1;
};

constexpr int fact5 = Factorial<5>::value;  // Computed at compile time

// C++11 constexpr version
constexpr int factorial(int n) {
    return (n <= 1) ? 1 : n * factorial(n - 1);
}

constexpr int fact6 = factorial(6);
```

Type Traits:

```cpp
template<typename T>
struct IsPointer {
    static constexpr bool value = false;
};

template<typename T>
struct IsPointer<T*> {
    static constexpr bool value = true;
};

// Usage
static_assert(IsPointer<int*>::value, "Should be true");
static_assert(!IsPointer<int>::value, "Should be false");
```

Perfect Forwarding

Philosophy: Preserve value categories when passing arguments to other functions.

```cpp
#include <utility>

template<typename T>
void wrapper(T&& arg) {
    // Perfect forwarding preserves lvalue/rvalue nature
    actual_function(std::forward<T>(arg));
}

// Variadic template version
template<typename Func, typename... Args>
auto call_function(Func&& func, Args&&... args) {
    return func(std::forward<Args>(args)...);
}
```

Custom Allocators

```cpp
#include <memory>

template<typename T>
class CustomAllocator {
public:
    using value_type = T;

    CustomAllocator() = default;

    template<typename U>
    CustomAllocator(const CustomAllocator<U>&) {}
```

```cpp
    T* allocate(std::size_t n) {
        std::cout << "Allocating " << n << " objects\n";
        return static_cast<T*>(std::malloc(n * sizeof(T)));
    }

    void deallocate(T* p, std::size_t n) {
        std::cout << "Deallocating " << n << " objects\n";
        std::free(p);
    }

    template<typename U>
    bool operator==(const CustomAllocator<U>&) const { return true; }

    template<typename U>
    bool operator!=(const CustomAllocator<U>&) const { return false; }
};

// Usage
std::vector<int, CustomAllocator<int>> vec;
```

Function Pointers and std::function

Function Pointers:

```cpp
// Function pointer declaration
int (*operation)(int, int);

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

// Assignment and usage
operation = add;
int result1 = operation(5, 3);  // 8

operation = multiply;
int result2 = operation(5, 3);  // 15
```

std::function (Type Erasure):

```cpp
#include <functional>

std::function<int(int, int)> func;

func = add;
int result1 = func(5, 3);

func = [](int a, int b) { return a - b; };
int result2 = func(5, 3);
```

```cpp
// Can store member functions with std::bind
class Calculator {
public:
    int divide(int a, int b) { return b != 0 ? a / b : 0; }
};

Calculator calc;
func = std::bind(&Calculator::divide, &calc, std::placeholders::_1,
std::placeholders::_2);
int result3 = func(10, 2);
```

Placement New

Philosophy: Construct objects at specific memory locations.

```cpp
#include <new>

char buffer[sizeof(MyClass)];

// Construct object in buffer
MyClass* obj = new(buffer) MyClass(args);

// Must manually call destructor
obj->~MyClass();
```

Custom Iterators

```cpp
template<typename T>
class SimpleVector {
private:
    T* data;
    size_t size;
    size_t capacity;

public:
    class Iterator {
    private:
        T* ptr;
    public:
        using iterator_category = std::random_access_iterator_tag;
        using value_type = T;
        using difference_type = std::ptrdiff_t;
        using pointer = T*;
        using reference = T&;

        Iterator(T* p) : ptr(p) {}

        reference operator*() const { return *ptr; }
        pointer operator->() const { return ptr; }
```

```
        Iterator& operator++() { ++ptr; return *this; }
        Iterator operator++(int) { Iterator temp = *this; ++ptr; return temp; }
        Iterator& operator--() { --ptr; return *this; }
        Iterator operator--(int) { Iterator temp = *this; --ptr; return temp; }

        Iterator operator+(difference_type n) const { return Iterator(ptr + n); }
        Iterator operator-(difference_type n) const { return Iterator(ptr - n); }
        difference_type operator-(const Iterator& other) const { return ptr - other.ptr; }

        bool operator==(const Iterator& other) const { return ptr == other.ptr; }
        bool operator!=(const Iterator& other) const { return ptr != other.ptr; }
        bool operator<(const Iterator& other) const { return ptr < other.ptr; }
    };

    Iterator begin() { return Iterator(data); }
    Iterator end() { return Iterator(data + size); }
};
```

Best Practices and Design Principles

SOLID Principles in C++

Single Responsibility Principle:

```cpp
// Bad: Class does too much
class User {
    std::string name;
    std::string email;
public:
    void setName(const std::string& n) { name = n; }
    void setEmail(const std::string& e) { email = e; }
    void saveToDatabase() { /* database logic */ }
    void sendEmail() { /* email logic */ }
};

// Good: Separate responsibilities
class User {
    std::string name;
    std::string email;
public:
    void setName(const std::string& n) { name = n; }
    void setEmail(const std::string& e) { email = e; }
```

```
    const std::string& getName() const { return name; }
    const std::string& getEmail() const { return email; }
};

class UserRepository {
public:
    void save(const User& user) { /* database logic */ }
};

class EmailService {
public:
    void sendEmail(const User& user) { /* email logic */ }
};
```

Open/Closed Principle:

```
// Open for extension, closed for modification
class Shape {
public:
    virtual double area() const = 0;
    virtual ~Shape() = default;
};

class Rectangle : public Shape {
private:
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double area() const override { return width * height; }
};

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

// Can add new shapes without modifying existing code
class Triangle : public Shape {
private:
    double base, height;
public:
    Triangle(double b, double h) : base(b), height(h) {}
    double area() const override { return 0.5 * base * height; }
};
```

Performance Guidelines

Prefer Stack to Heap:

```
// Good: Stack allocation
{
    MyClass obj;  // Automatic cleanup
}  // obj destroyed here

// Avoid: Unnecessary heap allocation
{
    std::unique_ptr<MyClass> obj = std::make_unique<MyClass>();
}
```

Use const and constexpr:

```
// Compile-time constants
constexpr double PI = 3.14159265359;
constexpr int factorial(int n) { return n <= 1 ? 1 : n * factorial(n-1); }

// Runtime constants
const std::string getMessage() {
    return "Hello, World!";
}
```

Avoid Unnecessary Copies:

```
// Bad: Unnecessary copies
std::vector<std::string> process(std::vector<std::string> input) {
    for (std::string str : input) {  // Copy each string
        // Process str
    }
    return input;  // Another copy
}

// Good: Use references and move
std::vector<std::string> process(std::vector<std::string> input) {
    for (const auto& str : input) {  // No copy
        // Process str
    }
    return std::move(input);  // Move instead of copy
}
```

Error Handling Guidelines

Use Exceptions for Exceptional Cases:

```cpp
class BankAccount {
private:
    double balance;
public:
    void withdraw(double amount) {
        if (amount < 0) {
            throw std::invalid_argument("Amount cannot be negative");
        }
        if (amount > balance) {
            throw std::runtime_error("Insufficient funds");
        }
        balance -= amount;
    }
};
```

RAII for Resource Management:

```cpp
class DatabaseConnection {
private:
    void* connection;
public:
    DatabaseConnection() : connection(connect_to_database()) {
        if (!connection) {
            throw std::runtime_error("Failed to connect to database");
        }
    }

    ~DatabaseConnection() {
        if (connection) {
            disconnect_from_database(connection);
        }
    }

    // Move-only semantics
    DatabaseConnection(const DatabaseConnection&) = delete;
    DatabaseConnection& operator=(const DatabaseConnection&) = delete;

    DatabaseConnection(DatabaseConnection&& other) noexcept
        : connection(other.connection) {
        other.connection = nullptr;
    }

    DatabaseConnection& operator=(DatabaseConnection&& other) noexcept {
        if (this != &other) {
            if (connection) disconnect_from_database(connection);
            connection = other.connection;
            other.connection = nullptr;
        }
        return *this;
    }
};
```

Code Organization

Header Files (.h/.hpp):

```cpp
#ifndef MYCLASS_H
#define MYCLASS_H

#include <string>
#include <vector>

class MyClass {
private:
    std::string name;
    std::vector<int> data;

public:
    // Constructor
    explicit MyClass(const std::string& n);

    // Destructor
    ~MyClass();

    // Copy operations
    MyClass(const MyClass& other);
    MyClass& operator=(const MyClass& other);

    // Move operations
    MyClass(MyClass&& other) noexcept;
    MyClass& operator=(MyClass&& other) noexcept;

    // Member functions
    void addData(int value);
    const std::vector<int>& getData() const;

    // Static functions
    static MyClass createDefault();
};

#endif // MYCLASS_H
```

Implementation Files (.cpp):

```cpp
#include "MyClass.h"
#include <algorithm>
#include <iostream>

MyClass::MyClass(const std::string& n) : name(n) {
    // Constructor implementation
```

```
}

MyClass::~MyClass() {
    // Destructor implementation
}

// ... other implementations
```

Conclusion

C++ is a powerful, multi-paradigm programming language that supports:

- **Procedural Programming**: Functions and structured programming
- **Object-Oriented Programming**: Classes, inheritance, polymorphism
- **Generic Programming**: Templates and compile-time computation
- **Functional Programming**: Lambda expressions, higher-order functions

Key Strengths:

- Zero-cost abstractions
- Deterministic resource management (RAII)
- Excellent performance
- Rich standard library
- Strong type system
- Multi-paradigm support

Best Practices Summary:

1. Follow RAII principles
2. Use smart pointers for dynamic memory
3. Prefer const and constexpr
4. Use uniform initialization
5. Embrace modern C++ features
6. Write exception-safe code
7. Use STL algorithms and containers
8. Follow the Rule of Zero/Three/Five
9. Make interfaces easy to use correctly and hard to use incorrectly
10. Optimize for readability first, performance second (unless in hot paths)

Common Pitfalls to Avoid:

- Manual memory management with new/delete
- Ignoring const-correctness
- Using C-style casts instead of C++ casts
- Not following RAII
- Premature optimization
- Using raw pointers for ownership
- Not using the standard library effectively

This reference covers the major features of C++ from basic syntax to advanced concepts. The language continues to evolve with new standards (C++23, C++26), but the fundamental principles and best practices remain consistent. return elements.empty(); } };

````

### Template Specialization

**Full Specialization**:
```cpp
template<>
class Stack<bool> {
    // Specialized implementation for bool
};
````

Partial Specialization:

```cpp
template<typename T>
class Stack<T*> {
    // Specialized implementation for pointer types
};
```

Template Parameters

Type Parameters:

```cpp
template<typename T>  // or template<class T>
```

Non-type Parameters:

```cpp
template<int N>
class Array {
    int data[N];
};
```

Template Template Parameters:

```cpp
template<template<typename> class Container>
class Wrapper {
    Container<int> container;
};
```

Variadic Templates (C++11)
```

```
template<typename... Args>
void print(Args... args) {
    ((std::cout << args << " "), ...);  // Fold expression (C++17)
}
```

Best Practices:

- Use meaningful template parameter names
- Provide good error messages with concepts (C++20) or SFINAE
- Avoid excessive template instantiation

Standard Template Library (STL)

Containers

Philosophy: Provide efficient, generic data structures with consistent interfaces.

Sequence Containers:

vector:

```
std::vector<int> v = {1, 2, 3, 4, 5};
v.push_back(6);
v[0] = 10;  // Random access
```

- Dynamic array
- Random access
- Efficient insertion/deletion at end

deque:

```
std::deque<int> d = {1, 2, 3};
d.push_front(0);  // Efficient insertion at both ends
d.push_back(4);
```

list:

```
std::list<int> l = {1, 2, 3};
l.insert(l.begin(), 0);  // Efficient insertion anywhere
```

- Doubly linked list
- No random access
- Efficient insertion/deletion anywhere

Associative Containers:

set/multiset:

```
std::set<int> s = {3, 1, 4, 1, 5};  // Sorted, unique elements
// s contains {1, 3, 4, 5}
```

map/multimap:

```
std::map<std::string, int> ages;
ages["Alice"] = 25;
ages["Bob"] = 30;
```

Unordered Containers (C++11):

unordered_set/unordered_multiset:

```
std::unordered_set<int> us = {1, 2, 3, 4, 5};
// Hash table implementation, average O(1) operations
```

unordered_map/unordered_multimap:

```
std::unordered_map<std::string, int> um;
um["key"] = 42;  // Average O(1) access
```

Iterators

Philosophy: Provide uniform interface to traverse containers.

Types:

- Input Iterator: Read-only, forward only
- Output Iterator: Write-only, forward only
- Forward Iterator: Read/write, forward only
- Bidirectional Iterator: Forward + backward
- Random Access Iterator: Jump to any position

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

// Iterator usage
for (auto it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << " ";
}
```

```
// Range-based for loop (preferred)
for (const auto& element : v) {
    std::cout << element << " ";
}
```

Algorithms

Philosophy: Separate data structures from algorithms for maximum reusability.

Common Algorithms:

find:

```
auto it = std::find(v.begin(), v.end(), 3);
if (it != v.end()) {
    std::cout << "Found at position: " << it - v.begin() << std::endl;
}
```

sort:

```
std::sort(v.begin(), v.end());  // Ascending
std::sort(v.begin(), v.end(), std::greater<int>());  // Descending
```

transform:

```
std::vector<int> result;
std::transform(v.begin(), v.end(), std::back_inserter(result),
               [](int x) { return x * x; });
```

for_each:

```
std::for_each(v.begin(), v.end(), [](int x) {
    std::cout << x << " ";
});
```

Function Objects and Lambda

Function Objects (Functors):

```
struct Multiply {
    int operator()(int x, int y) const {
        return x * y;
    }
```

```cpp
};

Multiply mult;
int result = mult(5, 3);  // result = 15
```

Lambda with STL:

```cpp
std::vector<int> v = {1, 2, 3, 4, 5};
auto count = std::count_if(v.begin(), v.end(), [](int x) { return x % 2 == 0; });
```

Exception Handling

Philosophy

Exception handling provides a mechanism to handle runtime errors gracefully, separating error handling code from normal program flow.

Basic Syntax

```cpp
try {
    // Code that might throw
    throw std::runtime_error("Something went wrong");
} catch (const std::exception& e) {
    std::cerr << "Exception: " << e.what() << std::endl;
} catch (...) {
    std::cerr << "Unknown exception" << std::endl;
}
```

Standard Exception Hierarchy

```
std::exception
├── std::runtime_error
│   ├── std::range_error
│   ├── std::overflow_error
│   └── std::underflow_error
├── std::logic_error
│   ├── std::domain_error
│   ├── std::invalid_argument
│   ├── std::length_error
│   └── std::out_of_range
├── std::bad_alloc
├── std::bad_cast
└── std::bad_typeid
```

Custom Exceptions

```
class MyException : public std::exception {
private:
    std::string message;
public:
    MyException(const std::string& msg) : message(msg) {}
    const char* what() const noexcept override {
        return message.c_str();
    }
};
```

Exception Safety

Levels:

1. **No-throw guarantee**: Never throws exceptions
2. **Strong exception safety**: Operations are atomic
3. **Basic exception safety**: No resource leaks, objects in valid state
4. **No exception safety**: Anything can happen

RAII and Exception Safety:

```
class File {
    std::FILE* file;
public:
    File(const std::string& filename)
        : file(std::fopen(filename.c_str(), "r")) {
        if (!file) throw std::runtime_error("Cannot open file");
    }

    ~File() {
        if (file) std::fclose(file);  // Always cleanup
    }

    // Prevent copying for simplicity
    File(const File&) = delete;
    File& operator=(const File&) = delete;
};
```

Best Practices:

- Catch exceptions by const reference
- Throw by value, catch by reference
- Use RAII for resource management
- Don't throw in destructors
- Use `noexcept` specification when appropriate

Modern C++ Features

C++11 Features

Auto Type Deduction:

```
auto x = 42;         // int
auto y = 3.14;       // double
auto z = "hello";    // const char*
```

Range-based For Loops:

```
std::vector<int> v = {1, 2, 3, 4, 5};
for (const auto& element : v) {
    std::cout << element << std::endl;
}
```

nullptr:

```
int* ptr = nullptr;  // Better than NULL
```

Strongly Typed Enums:

```
enum class Color : int {
    RED = 1,
    GREEN = 2,
    BLUE = 3
};

Color c = Color::RED;  // Must qualify with scope
```

Initializer Lists:

```
std::vector<int> v = {1, 2, 3, 4, 5};
std::map<std::string, int> m = {{"key1", 1}, {"key2", 2}};
```

Move Semantics:

```
class MyClass {
    std::string data;
public:
    // Move constructor
```

```
    MyClass(MyClass&& other) noexcept : data(std::move(other.data)) {}

    // Move assignment
    MyClass& operator=(MyClass&& other) noexcept {
        if (this != &other) {
            data = std::move(other.data);
        }
        return *this;
    }
};
```

C++14 Features

Generic Lambdas:

```
auto lambda = [](auto x, auto y) { return x + y; };
```

Return Type Deduction:

```
auto add(int a, int b) {
    return a + b;  // Return type deduced as int
}
```

Variable Templates:

```
template<typename T>
constexpr T pi = T(3.1415926535897932385);
```

C++17 Features

Structured Bindings:

```
std::map<std::string, int> m = {{"key", 42}};
auto [iterator, inserted] = m.insert({"new_key", 100});
```

if constexpr:

```
template<typename T>
void process(T value) {
    if constexpr (std::is_integral_v<T>) {
        // Handle integer types
    } else {
        // Handle other types
```

```
    }
}
```

Optional:

```cpp
std::optional<int> divide(int a, int b) {
    if (b != 0) return a / b;
    return std::nullopt;
}
```

C++20 Features

Concepts:

```cpp
template<typename T>
concept Printable = requires(T t) {
    std::cout << t;
};

template<Printable T>
void print(const T& value) {
    std::cout << value << std::endl;
}
```

Ranges:

```cpp
#include <ranges>

std::vector<int> v = {1, 2, 3, 4, 5, 6};
auto even = v | std::views::filter([](int x) { return x % 2 == 0; });
```

Coroutines:

```cpp
#include <coroutine>

generator<int> fibonacci() {
    int a = 0, b = 1;
    while (true) {
        co_yield a;
        auto temp = a;
        a = b;
        b = temp + b;
    }
}
```

Concurrency

Philosophy

Modern C++ provides high-level abstractions for concurrent programming, making it safer and more portable than platform-specific threading APIs.

std::thread

```cpp
#include <thread>
#include <iostream>

void worker_function(int id) {
    std::cout << "Worker " << id << " is running\n";
}

int main() {
    std::thread t1(worker_function, 1);
    std::thread t2(worker_function, 2);

    t1.join();  // Wait for t1 to complete
    t2.join();  // Wait for t2 to complete

    return 0;
}
```

Synchronization Primitives

Mutex:

```cpp
#include <mutex>

std::mutex mtx;
int shared_counter = 0;

void increment() {
    std::lock_guard<std::mutex> lock(mtx);
    ++shared_counter;  // Thread-safe increment
}
```

Recursive Mutex:

```cpp
std::recursive_mutex rec_mtx;

void recursive_function(int n) {
    std::lock_guard<std::recursive_mutex> lock(rec_mtx);
    if (n > 0) {
```

```
            recursive_function(n - 1);  // Can lock again
        }
    }
```

Condition Variable:

```
#include <condition_variable>

std::mutex mtx;
std::condition_variable cv;
bool ready = false;

void worker() {
    std::unique_lock<std::mutex> lock(mtx);
    cv.wait(lock, [] { return ready; });  // Wait until ready
    // Do work
}

void signal() {
    {
        std::lock_guard<std::mutex> lock(mtx);
        ready = true;
    }
    cv.notify_all();  // Wake up all waiting threads
}
```

Atomic Operations

```
#include <atomic>

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

void increment_atomic() {
    counter++;  // Atomic increment
}

void compare_and_swap() {
    int expected = 5;
    int desired = 10;
    bool success = counter.compare_exchange_strong(expected, desired);
}
```

async and future

```
#include <future>

int compute_something(int x) {
```

```
    return x * x;
}

int main() {
    // Launch async task
    std::future<int> result = std::async(std::launch::async,
                                         compute_something, 42);

    // Do other work...

    // Get result (blocks if not ready)
    int value = result.get();
    std::cout << "Result: " << value << std::endl;

    return 0;
}
```

Thread-Safe Patterns

Producer-Consumer with Queue:

```
#include <queue>
#include <mutex>
#include <condition_variable>

template<typename T>
class ThreadSafeQueue {
private:
    std::queue<T> queue;
    mutable std::mutex mtx;
    std::condition_variable cv;

public:
    void push(const T& item) {
        std::lock_guard<std::mutex> lock(mtx);
        queue.push(item);
        cv.notify_one();
    }

    T pop() {
        std::unique_lock<std::mutex> lock(mtx);
        cv.wait(lock, [this] { return !queue.empty(); });
        T item = queue.front();
        queue.pop();
        return item;
    }

    bool empty() const {
```