C++ Quick Reference

Classes, templates, STL, smart pointers, modern C++ essentials

Basics

Hello World

```
#include <iostream>
int main() {
    std::cout << "Hello, World!" << std::endl;
    return 0;
}
```

Compile & Run

```
g++ -std=c++20 -Wall -o app main.cpp
./app
clang++ -std=c++20 -o app main.cpp
```

Variables & Constants

```
int x = 42;
auto y = 3.14;             // type deduction
const int MAX = 100;
constexpr int SIZE = 256;  // compile-time constant
```

Namespaces

```
namespace math {
    double pi = 3.14159;
}
using namespace std;       // use sparingly
using std::cout;           // prefer selective
```

Classes

Class Definition

```
class Rectangle {
    double w_, h_;
public:
    Rectangle(double w, double h) : w_(w), h_(h) {}
    double area() const { return w_ * h_; }  };
```

Inheritance

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

Access Specifiers

`public`	Accessible from anywhere
`protected`	Accessible in class and derived classes
`private`	Accessible only within the class
`friend`	Grant access to specific function or class

Special Members

`Constructor`	**`MyClass(args)`** — initialize object
`Destructor`	**`~MyClass()`** — clean up resources
`Copy ctor`	**`MyClass(const MyClass&)`**
`Move ctor`	**`MyClass(MyClass&&)`** — transfer ownership
`Copy assign`	**`operator=(const MyClass&)`**
`Move assign`	**`operator=(MyClass&&)`**

Templates

Function Template

```
template <typename T>
T max_val(T a, T b) {
    return (a > b) ? a : b;
}
auto result = max_val(3, 7);  // deduced as int
```

Class Template

```
template <typename T>
class Stack {
    std::vector<T> data_;
public:
    void push(const T& v) { data_.push_back(v); }  };
```

Concepts (C++20)

```
template <typename T>
concept Numeric = std::integral<T> || std::floating_point<T>;
template <Numeric T>
T add(T a, T b) { return a + b; }
```

STL Containers

Sequence Containers

`vector<T>`	Dynamic array, fast random access
`deque<T>`	Double-ended queue
`list<T>`	Doubly-linked list
`array<T,N>`	Fixed-size array (compile-time size)
`forward_list<T>`	Singly-linked list

Associative Containers

`map<K,V>`	Ordered key-value pairs (red-black tree)
`set<T>`	Ordered unique elements
`unordered_map<K,V>`	Hash map, O(1) average lookup
`unordered_set<T>`	Hash set, O(1) average lookup
`multimap<K,V>`	Ordered, allows duplicate keys

Vector Operations

```
std::vector<int> v = {1, 2, 3};
v.push_back(4);
v.emplace_back(5);         // construct in place
v.size();  v.empty();
v[0];  v.at(0);            // at() has bounds check
```

Iterators & Algorithms

Iterator Usage

```
std::vector<int> v = {3, 1, 4, 1, 5};
for (auto it = v.begin(); it != v.end(); ++it) {
    std::cout << *it << " ";
}
for (const auto& val : v) { }  // range-based for
```

Common Algorithms

`sort(begin, end)`	Sort elements in ascending order
`find(begin, end, val)`	Find first occurrence of value
`count(begin, end, val)`	Count occurrences of value
`transform(b, e, out, fn)`	Apply function to each element
`accumulate(b, e, init)`	Reduce elements (sum by default)
`reverse(begin, end)`	Reverse element order
`unique(begin, end)`	Remove consecutive duplicates

Ranges (C++20)

```
namespace rv = std::views;
auto evens = v | rv::filter([](int n){ return n % 2 == 0; })
               | rv::transform([](int n){ return n * n; });
```

Smart Pointers

unique_ptr

```
auto p = std::make_unique<int>(42);
std::cout << *p << std::endl;
// auto-deleted when out of scope
// cannot be copied, only moved
```

shared_ptr

```
auto sp = std::make_shared<std::string>("hello");
auto sp2 = sp;               // reference count: 2
std::cout << sp.use_count(); // 2
```

Comparison

`unique_ptr<T>`	Exclusive ownership, zero overhead
`shared_ptr<T>`	Shared ownership via reference counting
`weak_ptr<T>`	Non-owning observer of **`shared_ptr`**
`make_unique<T>()`	Preferred way to create **`unique_ptr`**
`make_shared<T>()`	Preferred way to create **`shared_ptr`**

Lambdas

Lambda Syntax

```
auto add = [](int a, int b) { return a + b; };
int sum = add(3, 4);         // 7
```

Capture Modes

`[x]`	Capture **x** by value (copy)
`[&x]`	Capture **x** by reference
`[=]`	Capture all used variables by value
`[&]`	Capture all used variables by reference
`[=, &x]`	All by value, **x** by reference
`[this]`	Capture enclosing object pointer

Lambda with STL

```
std::vector<int> v = {5, 2, 8, 1};
std::sort(v.begin(), v.end(),
    [](int a, int b) { return a > b; });  // descending
auto it = std::find_if(v.begin(), v.end(),
    [](int n) { return n > 3; });
```

Strings & I/O

std::string

```
std::string s = "hello";
s += " world";               // concatenation
s.substr(0, 5);              // "hello"
s.find("world");             // 6 (position)
s.length();  s.empty();
```

String Conversions

`std::to_string(42)`	Number to string
`std::stoi(s)`	String to **`int`**
`std::stod(s)`	String to **`double`**
`std::stol(s)`	String to **`long`**

I/O Streams

```
std::cout << "output" << std::endl;
std::cin >> variable;
std::getline(std::cin, line);
```

File I/O

```
std::ofstream out("file.txt");
out << "hello" << std::endl;
std::ifstream in("file.txt");
std::string line;
while (std::getline(in, line)) { }
```

C++ Quick Reference

Error Handling

Exceptions

```
try {
    throw std::runtime_error("something failed");
} catch (const std::exception& e) {
    std::cerr << e.what() << std::endl;
} catch (...) { /* unknown error */ }
```

Standard Exceptions

`std::exception`	Base class for all standard exceptions
`std::runtime_error`	Runtime error with message
`std::logic_error`	Logic error (pre-condition violation)
`std::out_of_range`	Index or iterator out of range
`std::invalid_argument`	Invalid function argument
`std::bad_alloc`	Memory allocation failure

noexcept

```
void safe_func() noexcept {
    // guaranteed not to throw
}
bool can_throw = noexcept(safe_func());  // true
```

Modern C++ (17/20)

Structured Bindings (C++17)

```
std::map<std::string, int> m = {{"a", 1}, {"b", 2}};
for (auto& [key, value] : m) {
    std::cout << key << ": " << value << "\n";
}
```

std::optional (C++17)

```
std::optional<int> find(int id) {
    if (id > 0) return id * 10;
    return std::nullopt;
}
auto val = find(3);             // has_value() == true
```

std::variant & std::any (C++17)

```
std::variant<int, std::string> v = "hello";
std::cout << std::get<std::string>(v);
std::any a = 42;
int n = std::any_cast<int>(a);
```

Key Modern Features

`auto`	Type deduction for variables and return types
`constexpr`	Compile-time evaluation
`if constexpr`	Compile-time conditional (C++17)
`std::span<T>`	Non-owning view over contiguous data (C++20)
`std::format()`	Type-safe formatting (C++20)
`co_await`	Coroutine support (C++20)