Referência Rápida de C++

Classes, templates, STL, smart pointers, essenciais do C++ moderno

Básico

Hello World

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

Compilar & Executar

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

Variáveis & Constantes

```
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

Definição de Classe

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

Herança

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

Especificadores de Acesso

public	Acessível de qualquer lugar
protected	Acessível na classe e em classes derivadas
private	Acessível apenas dentro da classe
friend	Conceder acesso a função ou classe específica

Membros Especiais

Constructor	**MyClass(args)** — inicializar objeto
Destructor	**~MyClass()** — liberar recursos
Copy ctor	**MyClass(const MyClass&)**
Move ctor	**MyClass(MyClass&&)** — transferir ownership
Copy assign	**operator=(const MyClass&)**
Move assign	**operator=(MyClass&&)**

Templates

Template de Função

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

Template de Classe

```
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; }
```

Containers STL

Containers de Sequência

vector<T>	Array dinâmico, acesso aleatório rápido
deque<T>	Fila de duas pontas
list<T>	Lista duplamente encadeada
array<T,N>	Array de tamanho fixo (tamanho em compile-time)
forward_list<T>	Lista simplesmente encadeada

Containers Associativos

map<K,V>	Pares chave-valor ordenados (árvore red-black)
set<T>	Elementos únicos ordenados
unordered_map<K,V>	Hash map, busca O(1) médio
unordered_set<T>	Hash set, busca O(1) médio
multimap<K,V>	Ordenado, permite chaves duplicadas

Operações com Vector

```
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
```

Iteradores & Algoritmos

Uso de Iteradores

```
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
```

Algoritmos Comuns

sort(begin, end)	Ordenar elementos em ordem crescente
find(begin, end, val)	Encontrar primeira ocorrência do valor
count(begin, end, val)	Contar ocorrências do valor
transform(b, e, out, fn)	Aplicar função a cada elemento
accumulate(b, e, init)	Reduzir elementos (soma por padrão)
reverse(begin, end)	Inverter ordem dos elementos
unique(begin, end)	Remover duplicatas consecutivas

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
```

Comparação

unique_ptr<T>	Propriedade exclusiva, overhead zero
shared_ptr<T>	Propriedade compartilhada via contagem de referência
weak_ptr<T>	Observador sem propriedade de **shared_ptr**
make_unique<T>()	Forma preferida de criar **unique_ptr**
make_shared<T>()	Forma preferida de criar **shared_ptr**

Lambdas

Sintaxe Lambda

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

Modos de Captura

[x]	Capturar **x** por valor (cópia)
[&x]	Capturar **x** por referência
[=]	Capturar todas as variáveis por valor
[&]	Capturar todas as variáveis por referência
[=, &x]	Todas por valor, **x** por referência
[this]	Capturar ponteiro do objeto atual

Lambda com 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();
```

Conversões de String

std::to_string(42)	Número para string
std::stoi(s)	String para **int**
std::stod(s)	String para **double**
std::stol(s)	String para **long**

Streams de I/O

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

I/O de Arquivo

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

Referência Rápida de C++

Tratamento de Erros

Exceções

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

Exceções Padrão

`std::exception`	Classe base para todas as exceções padrão
`std::runtime_error`	Erro em tempo de execução com mensagem
`std::logic_error`	Erro lógico (violação de pré-condição)
`std::out_of_range`	Índice ou iterador fora do intervalo
`std::invalid_argument`	Argumento de função inválido
`std::bad_alloc`	Falha de alocação de memória

noexcept

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

C++ Moderno (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);
```

Recursos Modernos Chave

`auto`	Dedução de tipo para variáveis e retornos
`constexpr`	Avaliação em tempo de compilação
`if constexpr`	Condicional em compile-time (C++17)
`std::span<T>`	Visão não-proprietária sobre dados contíguos (C++20)
`std::format()`	Formatação type-safe (C++20)
`co_await`	Suporte a coroutines (C++20)