REFERENCIA RÁPIDA DE C++

Clases, plantillas, STL, smart pointers, esenciales de C++ moderno

Fundamentos

Hola Mundo

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

Compilar y Ejecutar

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

Variables y Constantes

```
int x = 42;
auto y = 3.14;              // deducción de tipo
const int MAX = 100;
constexpr int SIZE = 256;  // constante en tiempo de compilación
```

Namespaces

```
namespace math {
    double pi = 3.14159;
}
using namespace std;        // usar con moderación
using std::cout;            // preferir selectivo
```

Clases

Definición de Clase

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

Herencia

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

Especificadores de Acceso

`public`	Accesible desde cualquier lugar
`protected`	Accesible en la clase y clases derivadas
`private`	Accesible solo dentro de la clase
`friend`	Otorgar acceso a función o clase específica

Miembros Especiales

`Constructor`	`MyClass(args)` — inicializar objeto
`Destructor`	`~MyClass()` — liberar recursos
`Constructor de copia`	`MyClass(const MyClass&)`
`Constructor de movimiento`	`MyClass(MyClass&&)` — transferir propiedad
`Asignación de copia`	`operator=(const MyClass&)`
`Asignación de movimiento`	`operator=(MyClass&&)`

Plantillas

Plantilla de Función

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

Plantilla de Clase

```
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; }
```

Contenedores STL

Contenedores de Secuencia

`vector<T>`	Arreglo dinámico, acceso aleatorio rápido
`deque<T>`	Cola de doble extremo
`list<T>`	Lista doblemente enlazada
`array<T,N>`	Arreglo de tamaño fijo (tamaño en compilación)
`forward_list<T>`	Lista simplemente enlazada

Contenedores Asociativos

`map<K,V>`	Pares clave-valor ordenados (árbol rojo-negro)
`set<T>`	Elementos únicos ordenados
`unordered_map<K,V>`	Hash map, búsqueda promedio O(1)
`unordered_set<T>`	Hash set, búsqueda promedio O(1)
`multimap<K,V>`	Ordenado, permite claves duplicadas

Operaciones con Vector

```
std::vector<int> v = {1, 2, 3};
v.push_back(4);
v.emplace_back(5);          // construir en sitio
v.size();   v.empty();
v[0];  v.at(0);             // at() verifica límites
```

Iteradores y 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) { }  // for basado en rango
```

Algoritmos Comunes

`sort(begin, end)`	Ordenar elementos en forma ascendente
`find(begin, end, val)`	Encontrar primera ocurrencia del valor
`count(begin, end, val)`	Contar ocurrencias del valor
`transform(b, e, out, fn)`	Aplicar función a cada elemento
`accumulate(b, e, init)`	Reducir elementos (suma por defecto)
`reverse(begin, end)`	Invertir orden de elementos
`unique(begin, end)`	Eliminar duplicados consecutivos

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;
// eliminado automáticamente al salir del scope
// no se puede copiar, solo mover
```

shared_ptr

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

Comparación

`unique_ptr<T>`	Propiedad exclusiva, sin overhead
`shared_ptr<T>`	Propiedad compartida via conteo de referencias
`weak_ptr<T>`	Observador no propietario de `shared_ptr`
`make_unique<T>()`	Forma preferida de crear `unique_ptr`
`make_shared<T>()`	Forma preferida de crear `shared_ptr`

Lambdas

Sintaxis Lambda

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

Modos de Captura

`[x]`	Capturar `x` por valor (copia)
`[&x]`	Capturar `x` por referencia
`[=]`	Capturar todas las variables usadas por valor
`[&]`	Capturar todas las variables usadas por referencia
`[=, &x]`	Todas por valor, `x` por referencia
`[this]`	Capturar puntero al objeto envolvente

Lambda con STL

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

Cadenas y E/S

std::string

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

Conversiones de Cadena

`std::to_string(42)`	Número a cadena
`std::stoi(s)`	Cadena a `int`
`std::stod(s)`	Cadena a `double`
`std::stol(s)`	Cadena a `long`

Streams de E/S

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

E/S de Archivos

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

Manejo de Errores

Excepciones

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

Excepciones Estándar

`std::exception`	Clase base para todas las excepciones estándar
`std::runtime_error`	Error en tiempo de ejecución con mensaje
`std::logic_error`	Error lógico (violación de precondición)
`std::out_of_range`	Índice o iterador fuera de rango
`std::invalid_argument`	Argumento inválido de función
`std::bad_alloc`	Fallo en la asignación de memoria

noexcept

```
void safe_func() noexcept {
    // garantizado que no lanza excepciones
}
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 y 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);
```

Características Modernas Clave

`auto`	Deducción de tipo para variables y retornos
`constexpr`	Evaluación en tiempo de compilación
`if constexpr`	Condicional en tiempo de compilación (C++17)
`std::span<T>`	Vista no propietaria sobre datos contiguos (C++20)
`std::format()`	Formateo con seguridad de tipos (C++20)
`co_await`	Soporte para coroutines (C++20)