

RIFERIMENTO RAPIDO C++

Classi, template, STL, smart pointer, essenziali del C++ moderno

Basi

Hello World

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

Compilazione ed Esecuzione

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

Variabili e Costanti

```
int x = 42;
auto y = 3.14; // deduzione del tipo
const int MAX = 100;
constexpr int SIZE = 256; // costante a compile-time
```

Namespace

```
namespace math {
    double pi = 3.14159;
}
using namespace std; // usare con parsimonia
using std::cout; // preferire selettivo
```

Classi

Definizione di Classe

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

Ereditarietà

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

Specificatori di Accesso

public Accessibile da ovunque
protected Accessibile nella classe e nelle derivate
private Accessibile solo all'interno della classe
friend Concede accesso a una specifica funzione o classe

Membri Speciali

Costruttore `MyClass(args)` — inializza l'oggetto
Distruttore `~MyClass()` — libera le risorse
Costruttore copia `MyClass(const MyClass&)`
Costruttore move `MyClass(MyClass&&)` — trasferisce proprietà
Assegnazione copia `operator=(const MyClass&)`
Assegnazione move `operator=(MyClass&&)`

Template

Template di Funzione

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

Template di Classe

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

Concept (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; }
```

Contenitori STL

Contenitori Sequenziali

vector<T> Array dinamico, accesso casuale veloce
deque<T> Coda a doppia estremità
list<T> Lista doppiamente concatenata
array<T,N> Array di dimensione fissa (dimensione a compile-time)
forward_list<T> Lista singolarmente concatenata

Contenitori Associativi

map<K,V> Coppie chiave-valore ordinate (albero red-black)
set<T> Elementi unici ordinati
unordered_map<K,V> Hash map, ricerca O(1) media
unordered_set<T> Hash set, ricerca O(1) media
multimap<K,V> Ordinato, permette chiavi duplicate

Operazioni su Vector

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

Iterator e Algoritmi

Uso degli Iterator

```
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 range-based
```

Algoritmi Comuni

sort(begin, end) Ordina gli elementi in ordine crescente
find(begin, end, val) Trova la prima occorrenza del valore

count(begin, end, val) Conta le occorrenze del valore
transform(b, e, out, fn) Applica la funzione a ogni elemento
accumulate(b, e, init) Riduce gli elementi (somma per default)

reverse(begin, end) Inverte l'ordine degli elementi
unique(begin, end) Rimuove i duplicati consecutivi

Range (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 Pointer

unique_ptr

```
auto p = std::make_unique<int>(42);
std::cout << *p << std::endl;
// eliminato automaticamente uscendo dallo scope
// non può essere copiato, solo spostato
```

shared_ptr

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

Confronto

unique_ptr<T> Proprietà esclusiva, overhead zero
shazed_ptr<T> Proprietà condivisa tramite reference counting
weak_ptr<T> Osservatore non proprietario di `shared_ptr`
make_unique<T>() Modo preferito per creare `unique_ptr`
make_shared<T>() Modo preferito per creare `shared_ptr`

Lambda

Sintassi Lambda

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

Modalità di Cattura

[x] Cattura ``x`` per valore (copia)
[&x] Cattura ``x`` per riferimento
[=] Cattura tutte le variabili usate per valore
[&] Cattura tutte le variabili usate per riferimento
[=, &x] Tutte per valore, ``x`` per riferimento
[this] Cattura il puntatore all'oggetto contenitore

Lambda con STL

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

Stringhe e I/O

std::string

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

Conversioni Stringa

std::to_string(42) Numero a stringa
std::stoi(s) Stringa a `int`
std::stod(s) Stringa a `double`
std::stol(s) Stringa a `long`

Stream I/O

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

I/O su File

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

Gestione Errori

Eccezioni

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

Eccezioni Standard

std::exception Classe base per tutte le eccezioni standard
std::runtime_error Errore runtime con messaggio
std::logic_error Errore logico (violazione preconditione)
std::out_of_range Indice o iteratore fuori intervallo
std::invalid_argument Argomento di funzione non valido
std::bad_alloc Errore di allocazione memoria

noexcept

```
void safe_func() noexcept {
    // garantita a non lanciare
}
bool can_throw = noexcept(safe_func()); // true
```

C++ Moderno (17/20)

Structured Binding (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);
```

Funzionalità Moderne Principali

auto Deduzione del tipo per variabili e valori di ritorno
constexpr Valutazione a compile-time
if constexpr Condizionale a compile-time (C++17)
std::span<T> Vista non proprietaria su dati contigui (C++20)
std::format() Formattazione type-safe (C++20)
co_await Supporto coroutine (C++20)