Référence rapide C++

Classes, templates, STL, pointeurs intelligents, C++ moderne

Bases

Hello World

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

Compiler et exécuter

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

Variables et constantes

```
int x = 42;
auto y = 3.14;             // déduction de type
const int MAX = 100;
constexpr int SIZE = 256;  // constante à la compilation
```

Espaces de noms

```
namespace math {
    double pi = 3.14159;
}
using namespace std;       // utiliser avec parcimonie
using std::cout;           // préférer la sélection
```

Classes

Définition d'une classe

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

Héritage

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

Spécificateurs d'accès

public	Accessible depuis n'importe où
protected	Accessible dans la classe et les classes dérivées
private	Accessible uniquement dans la classe
friend	Accorder l'accès à une fonction ou classe spécifique

Membres spéciaux

Constructeur	**MaClasse(args)** — initialiser l'objet
Destructeur	**~MaClasse()** — libérer les ressources
Constructeur de copie	**MaClasse(const MaClasse&)**
Constructeur de déplacement	**MaClasse(MaClasse&&)** — transférer la propriété
Opérateur d'affectation par copie	**operator=(const MaClasse&)**
Opérateur d'affectation par déplacement	**operator=(MaClasse&&)**

Templates

Template de fonction

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

Conteneurs STL

Conteneurs séquentiels

vector<T>	Tableau dynamique, accès aléatoire rapide
deque<T>	File à double entrée
list<T>	Liste doublement chaînée
array<T,N>	Tableau de taille fixe (taille à la compilation)
forward_list<T>	Liste simplement chaînée

Conteneurs associatifs

map<K,V>	Paires clé-valeur ordonnées (arbre rouge-noir)
set<T>	Éléments uniques ordonnés
unordered_map<K,V>	Table de hachage, O(1) en moyenne
unordered_set<T>	Ensemble de hachage, O(1) en moyenne
multimap<K,V>	Ordonné, autorise les clés dupliquées

Opérations sur vector

```
std::vector<int> v = {1, 2, 3};
v.push_back(4);
v.emplace_back(5);         // construire en place
v.size();  v.empty();
v[0];  v.at(0);            // at() vérifie les bornes
```

Itérateurs et algorithmes

Utilisation des itérateurs

```
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 basé sur plage
```

Algorithmes courants

sort(begin, end)	Trier en ordre croissant
find(begin, end, val)	Trouver la première occurrence
count(begin, end, val)	Compter les occurrences
transform(b, e, out, fn)	Appliquer une fonction à chaque élément
accumulate(b, e, init)	Réduire les éléments (somme par défaut)
reverse(begin, end)	Inverser l'ordre des éléments
unique(begin, end)	Supprimer les doublons consécutifs

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; });
```

Pointeurs intelligents

unique_ptr

```
auto p = std::make_unique<int>(42);
std::cout << *p << std::endl;
// supprimé automatiquement hors de portée
// ne peut pas être copié, seulement déplacé
```

shared_ptr

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

Comparaison

unique_ptr<T>	Propriété exclusive, sans surcharge
shared_ptr<T>	Propriété partagée par comptage de références
weak_ptr<T>	Observateur non-propriétaire d'un **shared_ptr**
make_unique<T>()	Méthode préférée pour créer un **unique_ptr**
make_shared<T>()	Méthode préférée pour créer un **shared_ptr**

Lambdas

Syntaxe lambda

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

Modes de capture

[x]	Capturer **x** par valeur (copie)
[&x]	Capturer **x** par référence
[=]	Capturer toutes les variables utilisées par valeur
[&]	Capturer toutes les variables utilisées par référence
[=, &x]	Tout par valeur, **x** par référence
[this]	Capturer le pointeur de l'objet englobant

Lambda avec STL

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

Chaînes et E/S

std::string

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

Conversions de chaînes

std::to_string(42)	Nombre vers chaîne
std::stoi(s)	Chaîne vers **int**
std::stod(s)	Chaîne vers **double**
std::stol(s)	Chaîne vers **long**

Flux E/S

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

Référence rapide C++

E/S fichier

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

Gestion des erreurs

Exceptions

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

Exceptions standard

`std::exception`	Classe de base pour toutes les exceptions standard
`std::runtime_error`	Erreur d'exécution avec message
`std::logic_error`	Erreur logique (violation de précondition)
`std::out_of_range`	Index ou itérateur hors limites
`std::invalid_argument`	Argument de fonction invalide
`std::bad_alloc`	Échec d'allocation mémoire

noexcept

```
void safe_func() noexcept {
    // garantie de ne pas lancer d'exception
}
bool can_throw = noexcept(safe_func());  // true
```

C++ moderne (17/20)

Liaisons structurées (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 et 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);
```

Fonctionnalités modernes clés

`auto`	Déduction de type pour variables et types de retour
`constexpr`	Évaluation à la compilation
`if constexpr`	Condition à la compilation (C++17)
`std::span<T>`	Vue non-propriétaire sur des données contiguës (C++20)
`std::format()`	Formatage type-safe (C++20)
`co_await`	Support des coroutines (C++20)