

Cours C++

Bibliothèque standard : tableaux et quelques autres classes

Partie 1

Bibliothèque standard

- ▶ **Conteneurs** : `std::vector`, `std::list`, `std::array`, `std::map` (=dict de python), `std::stack`, `std::queue`, itérateurs...
- ▶ **Chaines de caractères** : `std::string`, UTF-8, locales, `std::stringstream`, `std::format`, expressions régulières...
- ▶ **Algorithmes** : `std::transform()`, `std::find()`, `std::count()`, `std::replace()`, `std::reverse()`, `std::sort()`, `min`, `max`, intersections, unions... et la toute nouvelle `<ranges>`
- ▶ Mesure du temps et des dates : `<chrono>`
- ▶ Génération pseudo-aléatoire (loi uniforme, normale, poissonienne...) : `<random>`
- ▶ **Parallélisme** : threads, verrous...
- ▶ **Pointeurs automatiques** : `std::shared_ptr`, `std::weak_ptr`, `std::atomic`
- ▶ Opérations vectorisées ("à la numpy") : `<valarray>`
- ▶ **Fonctions mathématiques spéciales** (fonctions de Bessel...)
- ▶ **Objets utiles** : `std::tuple`, `std::optional`, `std::complex`...

Le conteneur vector

- ▶ La classe `vector` est proche par son utilisation du tableau défini en C
- ▶ Avantages majeurs de la classe `vector` :
 1. possibilité de stocker n'importe quel type (uniforme) de données
 2. capacité à réallouer automatiquement l'espace mémoire nécessaire lors de l'ajout d'éléments ou lors de redimensionnement → *taille dynamique*
 3. désallocation de mémoire automatique
 4. propose un *itérateur* : manipulable par les algorithmes de la bibliothèque standard (tri, somme...) et parcouru avec les *range-based loop*

La classe vector

```
#include <vector>
#include <iostream>

int main()
{
    // Déclaration d'un tableau d'entiers vide
    std::vector<int> tableau;

    // Ajout de trois éléments (et allocation mémoire si nécessaire)
    tableau.push_back(4);
    tableau.push_back(2);
    tableau.push_back(5);

    // La méthode size donne le nombre actuel d'éléments
    for (size_t i = 0; i < tableau.size(); ++i)
        std::cout << i << ": " << tableau[i] << std::endl;

    // La mémoire est automatiquement libérée ici
}
```

La classe vector

```
// Création d'un tableau d'entiers contenant 70,70,70,70,70
std::vector<int> tableau2(5, 70);

// Réassignation des valeurs de ce tableau
tableau2[0] = 5;
tableau2[1] = 3;
tableau2[2] = 7;
tableau2[3] = 4;
tableau2[4] = 8;

// Insertion après la position 2
tableau2.insert(tableau2.begin()+2, 42);

// Récupération du dernier élément du tableau (8)
cout << tableau2.back() << endl;

// Création par initializer list
std::vector<int> tableau3 = {4, 8, 1, -4, 5};
```

Range-based loops

Tout conteneur présentant des *itérateurs* peuvent être parcourus avec une boucle similaire au `for x in ...` de Python :

```
for (int x : tableau2) {  
    cout << x << endl;  
}
```

Si les éléments doivent être modifiés, il faut les prendre par référence :

```
for (int& x : tableau2) {  
    x += 2;  
}
```

[Bonus] Quelques algorithmes

Les **algorithmes de la bibliothèque standard** sont dans le header `<algorithm>`.

Recherche :

```
std::vector::iterator it = std::find(tableau3.begin(), tableau3.end(), 4);  
  
if (it != tableau3.end())  
    cout << "4 est en position" << (it - tableau3.begin()) << endl;
```

Ici, `it` est un *itérateur*. Se comportent comme un pointeur.

`vector::begin()` pointe le premier élément, `vector::end()` pointe après le dernier élément. Peuvent être avancés (`it++`, `it+5`), reculés (`it--`), soustraits...

Tri ascendant :

```
std::sort(tableau3.begin(), tableau3.end());  
// tableau3 est maintenant : {-4, 1, 4, 5, 8}
```

Tableau vs vector : remplissage à partir d'un fichier

Exemple de remplissage d'un tableau de points à partir d'un fichier de points (fichier deux colonnes x et y)

```
struct point {  
    double x, y;  
};
```


Tableau vs vector : remplissage à partir d'un fichier

```
ifstream data("donnees.txt");
double x, y;
unsigned int i = 0, nl = 0;
while (data >> x >> y) i++;
data.close();
nl = i;
```

```
point * tabpoint = new point[nl];
i = 0;
data.open("donnees.txt");
while (data >> x >> y) {
    tabpoint[i] = {x, y};
    i++;
}
```

```
for (i = 0; i < nl; i++)
    cout << tabpoint[i].x << ", " << tabpoint[i].y << endl;
```

```
delete[] tabpoint;
```

```
ifstream data("donnees.txt");
double x, y;
vector<point> tabpoint;
while (data >> x >> y) {
    tabpoint.push_back({x, y});
}

for (point p : tabpoint)
    cout << tabpoint[i].x << ", "
        << tabpoint[i].y << endl;
```

Tableau vs vector : remplissage à partir d'un fichier

```
ifstream data("donnees.txt");
double x, y;
unsigned int i = 0, nl = 0;
while (data >> x >> y) i++;
data.close();
nl = i;
```

```
point * tabpoint = new point[nl];
i = 0;
data.open("donnees.txt");
while (data >> x >> y) {
    tabpoint[i] = {x, y};
    i++;
}
```

```
for (i = 0; i < nl; i++)
    cout << tabpoint[i].x << ", " << tabpoint[i].y << endl;
```

```
delete[] tabpoint;
```

```
ifstream data("donnees.txt");
double x, y;
vector<point> tabpoint;
while (data >> x >> y) {
    tabpoint.push_back({x, y});
}

for (point p : tabpoint)
    cout << tabpoint[i].x << ", "
        << tabpoint[i].y << endl;
```

Chaînes de caractères

- ▶ En C, le texte est représenté par des chaînes de caractères **char*** terminées par un zéro. Comme les tableaux C, pénible à manipuler.
- ▶ Dans le même esprit que `std::vector` : la classe `std::string`
- ▶ Avantages majeurs :
 1. capacité à réallouer automatiquement l'espace mémoire nécessaire lors de l'ajout de texte ou lors de redimensionnement → *taille dynamique*
 2. désallocation de mémoire automatique
 3. concaténations de chaînes, insertion, recherche, sous-chaîne...

Chaînes de caractères : exemple

```
// Initialisation à partir d'un char*
std::string s1 = "C'est";

// Ajout à la chaîne
s1 += " bien ";

std::string s2 = "partique ?";
// Remplacement en position 1
s2.replace(1, 2, "ra");
// Accès à un caractère par indice
s2[s2.size()-1] = '!';

// Concaténation et assignation à une nouvelle std::string
auto s = s1 + s2;

std::cout << s << std::endl;
// Affiche : C'est bien pratique !
```

Chaînes de caractères : autre exemple

```
using namespace std;

// Initialisation d'une chaîne vide
string s;

// Lecture depuis le terminal
cin >> s;

// Recherche du premier caractère '@'
string::size_type pos = s.find('@');
if (pos != string::npos) {
    cout << "Le caractère @ a été trouvé en position " << pos << endl;

    // Sous-chaîne : affiche tout ce qui est avant le @
    cout << s.substr(0, pos) << endl;
}
```

Conversion chaînes → nombres

- ▶ `std::istream` : fonctionne comme `std::cin` ou `std::ifstream`
- ▶ `stoi`, `stol`, `stoul`, `stof`, `stod`, `stold`...

[Bonus] Formattage des chaînes de caractères

1. Avec `std::ostringstream`

Avantage : s'utilise exactement comme `std::cout`. N'affiche rien en sortie, mais stocke le texte dans un buffer. Le formattage s'effectue avec des *manipulateurs*.

```
#include <sstream>
#include <iomanip>

float pi = 3.1415;
std::ostringstream s; // déclaration du buffer-flux
s << std::fixed << std::setprecision(2);
s << "pi = " << pi;
std::string texte = s.str(); // récupération de la chaîne de caractère
```

→ Chaîne de caractère texte contenant "pi = 3.14".

[Bonus] Formattage des chaînes de caractères

2. Avec `std::to_string`

Convertit un nombre en chaîne de caractère. Exemple :

```
#include <string>
using namespace std::string_literals;

int x = 4;
std::string texte = "chose"s + std::to_string(x) + "truc";
```

→ Chaîne de caractère `texte` contenant `"chose4truc"`.

Peu flexible pour les flottants : notation décimale seulement et précision fixée.

Note : l'opérateur `+` n'est défini que sur `std::string`, pas sur `const char*`, d'où la nécessité d'écrire `std::string("chose")+...` ou le littéral `"chose"s` associé et pas `"chose"+...`

[Bonus] Formattage des chaînes de caractères

3. Avec `std::format`

Ce qui est le plus familier et pratique. Fonctionne exactement comme en Python !

```
#include <format>

double t = 1234.5;
double x = 0.776, y = 0.921;
std::string texte =
    std::format("Position at t={:.1e} : (x={:.2f},y={:.2f})", t, x, y);
```

→ "Position at t=1.2e+3 : (x=0.78,y=0.92)".

Attention, la version de la bibliothèque standard sur les machines du magistère n'est pas assez récente pour comporter `std::format`... Vous pouvez installer son ancêtre, la [libfmt](#).

Tuples

Pour retourner plusieurs valeurs d'une fonction, on a vu le passage par référence/pointeur :

```
void ma_fonction (int a, float b, bool& ret1, float& ret2) {  
    ...  
    ret1 = true; ret2 = 3.14;  
}
```

```
bool r1; float r2;  
ma_fonction(42, 1.41, r1, r2);
```

On peut aussi faire comme en Python en retournant un **tuple** :

```
#include <tuple>  
  
std::tuple<bool, float> ma_fonction (int a, float b) {  
    ...  
    return {true, 3.14};  
}  
  
auto [r1, r2] = ma_fonction(42, 1.41);
```

Tuples

Pour retourner plusieurs valeurs d'une fonction, on a vu le passage par référence/pointeur :

```
void ma_fonction (int a, float b, bool& ret1, float& ret2) {  
    ...  
    ret1 = true; ret2 = 3.14;  
}
```

```
bool r1; float r2;  
ma_fonction(42, 1.41, r1, r2);
```

On peut aussi faire comme en Python en retournant un **tuple** :

```
#include <tuple>  
  
std::tuple<bool, float> ma_fonction (int a, float b) {  
    ...  
    return {true, 3.14};  
}  
  
auto [r1, r2] = ma_fonction(42, 1.41);
```

Tuples

Le type `std::tuple<>` prend en argument template la liste des types composant le tuple. La taille et le type d'un tuple n'est pas dynamique.

Outre le dépaquetage avec `auto [...] = ...`, on peut accéder à un élément du tuple par son numéro :

```
std::tuple<bool, float> tup = {true, 3.14};  
cout << std::get<1>(tup) << endl; // affiche 3.14
```

Tuples

Néanmoins, il est souvent plus clair et explicite de retourner des structures :

```
struct segment_t { bool ok; point_t orig; vec_t vecteur; }
```

```
segment_t ma_fonction (...) {  
    ...  
    return { true, {1.,2.}, {3.,4.} };  
}
```

```
segment_t res = ma_fonction(...);  
if (res.ok) {  
    ...
```

Dans ce cas, la déclaration-assignation en une seule ligne fonctionne encore :

```
auto [ok, orig, vec] = ma_fonction(...);
```