

# Notions C++ de bases pour les algorithmes STL

Algorithmes parallèles STL avec C++ 20

---

Michaël El Kharroubi

27.02.2024

HPC 2024 - HEPIA

# Introduction

---

Nous allons voir succinctement quels sont les outils de C++ indispensables pour pouvoir utiliser la parallélisation sur GPU avec les algorithmes STL

# Itérateurs

---

# Qu'est-ce qu'un itérateur en C++

Le concept n'est pas strictement similaire au design pattern itérateur

Les itérateurs en C++ sont des objets qui permettent de parcourir une collection

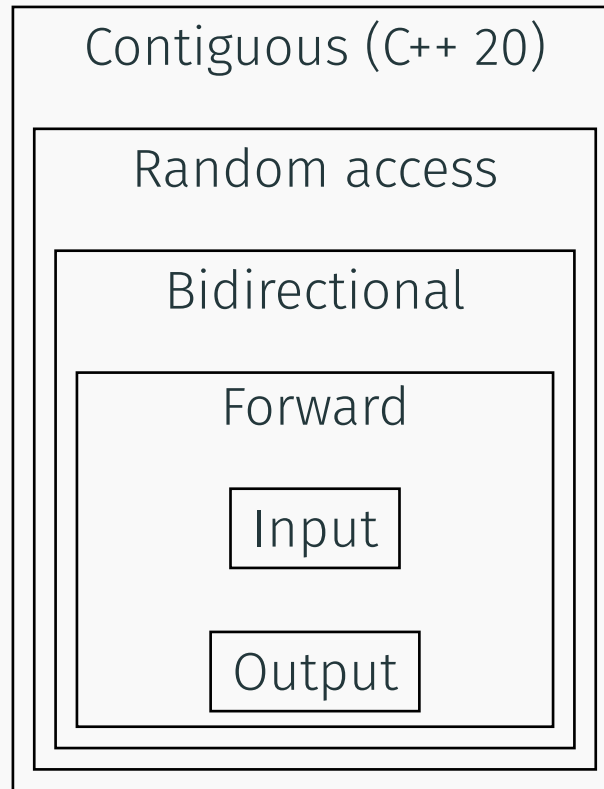
La collection peut-être virtuelle ou il peut s'agir d'une zone mémoire contigüe

On peut également concevoir l'itérateur comme une généralisation du pointeur en C++

# Les types d'itérateurs

Il existe plusieurs types d'itérateurs en C++

# Les types d'itérateurs



# Comment utiliser un itérateur

Toutes les collections de la STL proposent une méthode `begin` et `end`. Elles permettent d'obtenir un itérateur sur le début et la fin de la collection

L'itérateur le plus basique peut être incrémenté `it++` ou `++it`

L'itérateur bidirectionnel peut aussi être décrémenté `it--` ou `--it`

Selon le type d'itérateur, il est également possible de le déréférencer `*it`

S'il est possible d'y accéder aléatoirement, on peut utiliser l'opérateur `it[i]` pour accéder au  $i^{\text{ème}}$  élément

Je vous recommande de lire cette page pour plus d'exemples : <https://cplusplus.com/reference/iterator>



# Vecteurs



# Qu'est-ce qu'un vecteur

Un **std::vector** est une collection de la STL. Il s'agit d'un tableau de taille dynamique

La mémoire est gérée par l'objet. Il n'y a donc pas besoin de l'allouer avec `malloc` ou de la libérer avec `free`

Il est possible, par exemple, de l'allouer comme un tableau ou de lui donner une taille fixe et une valeur de remplissage

```
std::vector<int> v = {1, 2, 3, 4};  
size_t n = 100;  
std::vector<float> v1(n, 0.0);
```

# Autres méthodes intéressantes d'un vecteur

Il est possible de récupérer un pointeur sur les données du vecteur en utilisant la méthode `data` :

```
std::vector<int> v = {1, 2, 3, 4};  
int* ptr = v.data();
```

Grâce aux méthodes `begin` et `end`, on peut également itérer sur le vecteur avec une range-for loop :

```
std::vector<int> v = {1, 2, 3, 4};  
for (int &i : v){  
    i = 2*i;  
}
```

Il est possible de redimensionner le vecteur à l'aide de la méthode `resize`

# Tableaux et Spans

---

# Le type tableau (C-style array)

En C++, il existe un type tableau. On peut par exemple déclarer un tableau statique de cette manière :

```
int a[3] = {1, 2, 3}; // Déclaration statique
```

On peut aussi déclarer un tableau dynamique sur le tas de cette manière :

```
int *a = new int[100]; // Allocation dynamique
```

```
delete[] a; // Libération de la mémoire
```

On accède aux éléments d'un tableau avec l'opérateur [], par exemple :

```
a[5] = 12.3;
```

# Les inconvénients du type tableau

Le type tableau est un héritage du C. Comme c'est le cas en C, si le tableau est casté en pointeur, on perd l'information sur la taille du tableau

Si le tableau est alloué dynamiquement, il faut gérer la mémoire

Le type tableau ne propose aucune propriété, ni méthode. Il s'agit uniquement d'une zone mémoire

# Le type `std::array`

La STL propose une collection pour les tableaux statiques, `std::array`. Cette collection combine les performances d'un simple tableau, et les avantages d'une collection STL

On peut l'initialiser de cette manière :

```
std::array<int, 3> a2 = {1, 2, 3};
```

Comme c'est un tableau statique, il n'est pas nécessaire de libérer la mémoire

Le type `std::array` offre, entre autres, les méthodes `data`, `begin` ou `end`

La méthode `size` permet de connaître la taille du tableau

# Le type `std::array`

La méthode `at` permet d'accéder à un élément du tableau en vérifiant que l'index passé est valide (bounds-check)



Il peut arriver que nous ayons uniquement un tableau simple ou un pointeur vers des données. En résumé, une simple zone mémoire

Si nous voulons pouvoir utiliser notre zone mémoire comme une collection STL, pour profiter des différentes méthodes que nous avons vu, il existe les `std::span`

Cette structure encapsule simplement un pointeur et la taille de la zone pointée

# Exemple de span

```
int *ptr = new int[5];  
std::span<int> sp{ptr, 5};
```

```
int i = 0;  
for(int &val: sp){  
    val = i++;  
}
```

```
auto mid = sp.subspan(1, 3);  
for(int &val: mid){  
    val *= -1;  
}
```

```
delete[] ptr;
```

# Lambda & captures

---

# Les fonctions anonymes

De plus en plus de langages proposent un mécanisme pour écrire des fonctions anonymes, aussi appelées expressions lambda

Depuis C++ 11, il existe un moyen d'écrire des fonctions anonymes. La syntaxe simplifiée d'une lambda en C++ est la suivante :

```
[captures](paramètres){corps de la fonction}
```

Si je veux par exemple définir une fonction qui multiplie un entier par 2, je peux écrire :

```
auto times_two = [](int i){return 2*i;};  
int x = times_two(3); // x vaut 6
```

# Les captures

Les expressions lambda en C++ peuvent capturer leur environnement

Si je souhaite par exemple ajouter une constante à un entier, je peux écrire :

```
int cst = 5;  
auto add_cst = [cst](int i){ return i+cst;};  
int x = add_cst(10); // x vaut 15
```

# Les types de captures

Il existe deux moyens de capturer une variable :

## Par copie

```
int var = 5;
auto add_var = [v=var](int i)
                { return i+v; };

var = 2;
int x = add_var(10); // x vaut 15
```

## Par référence

```
int var = 5;
auto add_var = [v=&var](int i)
                { return i+*v; };

var = 2;
int x = add_var(10); // x vaut 12
```

# Les captures totales

On peut également copier l'environnement au complet :

## Par copie

```
int var = 5;
auto add_var = [=](int i)
                { return i+var; };

var = 2;
int x = add_var(10); // x vaut 15
```

## Par référence

```
int var = 5;
auto add_var = [&](int i)
                { return i+var; };

var = 2;
int x = add_var(10); // x vaut 12
```

# Algorithmes STL

---



# Introduction

La librairie standard propose un ensemble d'algorithmes qui permettent de travailler avec des itérateurs

Ces algorithmes permettent entre autre de :

- initialiser/remplir
- trier
- transformer
- réduire
- supprimer/remplacer des éléments

On retrouve des fonctions similaires avec les streams en java, ou les itérateurs en Rust

Je vous recommande de lire la page suivante pour plus d'information : <https://en.cppreference.com/w/cpp/algorithm>

# Exemple tri

```
std::vector<int> v = {6, 3, 4, 12, 1, 15};

std::sort(v.begin(), v.end());

std::vector<int> v_sorted = {1, 3, 4, 6, 12, 15};

bool is_sorted = std::equal(v.begin(), v.end(), v_sorted.begin());

std::cout << is_sorted << std::endl;
```

# Exemple de transformation

```
std::vector<int> v = {6, 3, 4, 12, 1, 15};
std::vector<std::string> parity{6};

std::transform(v.begin(), v.end(),
               parity.begin(),
               [](int i){
                   std::string s = (i % 2) == 0 ? "even" : "odd";
                   return s;
               });

for (std::string &p : parity){
    std::cout << p << " ";
}
std::cout << std::endl;
```

# Exemple de reduction

```
constexpr int kN = 100;
std::vector<int> v(kN, 0);

std::iota(v.begin(), v.end(), 1);

int sum = std::accumulate(v.begin(), v.end(),
                          0,
                          [](int acc, int i){
                              return acc + i;
                          });

bool sum_correct = sum == (kN*(kN+1))/2;
std::cout << sum_correct << std::endl;
```

# Exemple de suppression conditionnelle (filter)

```
std::vector<int> v = {6, 3, 4, 12, 1, 15};

auto last = std::remove_if(v.begin(), v.end(),
                           [](int i){
                               return (i % 2) == 1;
                           });

size_t filtered_size = std::distance(v.begin(), last);
v.resize(filtered_size);

for (int i : v){
    std::cout << i << " ";
}
std::cout << std::endl;
```

Notions à retenir

---

# Notions essentielles à retenir

- Les collections proposent un itérateur sur le début et la fin avec les méthodes `begin` et `end`
- Un `std::vector` est un tableau dynamique et un `std::array` est un tableau statique
- Si l'on est pas responsable des données, il faut utiliser un `std::span`
- Les lambdas sont des fonctions anonymes qui peuvent capturer totalement ou partiellement leur environnement
- On peut capturer par référence ou par copie
- Les algorithmes STL à connaître sont :
  - `iota`
  - `transform`
  - `for_each`
  - `accumulate`
  - `remove_if`



Questions ?

---