

Programmation I :

Cours de programmation (C++)

Tableaux

Jamila Sam

Laboratoire d'Intelligence Artificielle
Faculté I&C

Vidéos, transparents et quiz

<https://www.coursera.org/learn/init-prog-cpp/home/week/5>

 Semaine 5

		taille initiale connue <i>a priori</i> ?	
		non	oui
taille pouvant varier lors de l'utilisation du tableau ?	oui	vector	(vector)
	non	(vector)	array (C++11) tableaux « à la C »

Tableau dynamique : *collection* de données de *même type*, c'est-à-dire dont le nombre peut changer au cours du déroulement du programme,

En C++ : type `vector`

Nécessite : `#include <vector>`

(En toute rigueur, `vector` n'est pas un type, mais un « *template* ».

Nous verrons les « templates » tard au second semestre).

C++11 Initialisation d'un tableau dynamique

En C++11, il y a cinq façons d'initialiser un tableau dynamique :

- ▶ vide

```
vector<int> tab;
```

- ▶ avec un ensemble de valeurs initiales

```
vector<int> tab({ 20, 35, 26, 38, 22 });
```

- ▶ avec une taille initiale donnée et tous les éléments « nuls »

```
vector<int> tab(5);
```

- ▶ avec une taille initiale donnée et tous les éléments à une même valeur donnée

```
vector<int> tab(5, 1);
```

- ▶ avec une copie d'un autre tableau

```
vector<int> tab(tab2);
```

Accès direct aux éléments d'un tableau

Le $i+1^{\text{ème}}$ élément d'un tableau `tab` est désigné par
`tab[i]`



Attention ! Les indices correspondant aux éléments d'un tableau de taille `taille` **varient entre 0 et `taille-1`**

Le 1^{er} élément d'un tableau `tab` précédemment déclaré est donc `tab[0]` et son 10^e élément est `tab[9]`



Attention !! Il n'y a **pas de contrôle de débordement !!**

`tab[10000]`

Il est impératif que l'élément que vous désigniez **existe** effectivement !

Sinon risque de **Segmentation Fault** !

Exemple (à ne **pas** suivre !) d'erreur classique :


```
vector<double> v;  
v[0] = 5.4; // NON !!
```

→ `v[0]` n'existe pas encore !

Accès aux éléments d'un tableau (2)

Très souvent, on voudra accéder aux éléments d'un tableau en effectuant une **itération** sur ce tableau.


Il existe en fait au moins *trois* façons d'itérer sur un tableau :

- ▶  avec les itérations sur ensemble de valeurs

```
for (auto element : tableau)
for (auto& element : tableau)
```

- ▶ [C, C++98] avec une itération `for` « classique » :

```
for (size_t i(0); i < tableau.size(); ++i)
```

- ▶  [C++98] avec des itérateurs (2^e semestre)

Lequel choisir ?

- ▶ à chaque fois que c'est possible : le premier
- ▶ sinon : le deuxième.

Fonctions spécifiques

Quelques fonctions disponibles pour un tableau dynamique
`tableau` de type `vector<type>` :

`tableau.size()` : renvoie la taille de `tableau` (type de retour : `size_t`)

`tableau.front()` : renvoie une référence au 1^{er} élément
`tableau.front()` est donc équivalent à `tableau[0]`

`tableau.back()` : renvoie une référence au dernier élément.
`tableau.back()` est donc équivalent à `tableau[tableau.size() - 1]`

`tableau.empty()` : détermine si `tableau` est vide ou non
(`bool`).

`tableau.clear()` : supprime tous les éléments de `tableau` (et le transforme donc en un tableau vide). Pas de (type de) retour.

`tableau.pop_back()` : supprime le dernier élément de
`tableau`. Pas de (type de) retour.

`tableau.push_back(valeur)` : ajoute un nouvel élément de
valeur `valeur` à la fin de `tableau`. Pas de (type de) retour.

Les tableaux multidimensionnels

tableau à plusieurs dimensions : **tableau de tableaux...**

Le type de base d'un tableau peut être n'importe quel type, y compris composé. En particulier, le type de base d'un tableau peut être lui même un tableau.

Exemple :

```
vector<vector<int>> tab(5, vector<int>(6));
```

correspond à la déclaration d'un tableau de 5 tableaux de 6 entiers

`tab[i]` est donc un « `vector<int>` », c'est-à-dire un tableau dynamique d'entiers (qui, au départ, en contient 6)

`tab[i][j]` sera alors le $(j+1)$ -ième élément de ce tableau.



Le template vector



```
#include <vector>
```

Déclaration : `vector< type > identificateur;`

Déclaration/Initialisation :

```
vector< type > identificateur(taille);
```

Accès au (i+1)-ème élément : `tab[i];`

Fonctions spécifiques :

`int tab.size()` : renvoie la taille

`bool tab.empty()` : détermine s'il est vide ou non

`void tab.clear()` : supprime tous les éléments

`void tab.pop_back()` : supprime le dernier élément

`void tab.push_back(valeur)` : ajoute un nouvel élément à la fin

		taille initiale connue <i>a priori</i> ?	
		non	oui
taille pouvant varier lors de l'utilisation du tableau ?	oui	vector	(vector)
	non	(vector)	array (C++11) tableaux « à la C »

Nécessite : C++ 11 et

```
#include <array>
```

Inconvénients des tableaux de taille fixe à la C

Les tableaux de taille fixe à la C :

- ▶ sont toujours passés par référence
- ▶ n'ont pas connaissance de leur propre taille
- ▶ ne peuvent pas être manipulés globalement (pas de « = »)
- ▶ ne peuvent pas être retournés par une fonction
- ▶ ont une syntaxe d'initialisation particulière

☞ **AUCUN** avantage !

Mais je pense qu'ils resteront malgré tout assez répandus (inertie)... :- (

Pour ceux que cela intéresse : voir l'annexe (site du MOOC)

C++11 Déclaration d'un tableau de taille fixe

Une variable correspondant à un tableau de taille fixe se déclare de la façon suivante :

```
array<type, taille> identificateur;
```

où *identificateur* est le nom du tableau, *type* correspond au type des éléments du tableau et *taille* est le nombre d'éléments que contient le tableau.

Ce nombre doit être **connu à l'avance** (→ sinon *vector*)

C++11 Initialisation d'un tableau de taille fixe

Comme pour les variables de type élémentaire, un tableau de taille fixe peut être initialisé directement lors de sa déclaration :

```
array<type, taille> identificateur({val1, ... , valn});
```

ou

```
array<type, taille> identificateur = {val1, ... , valn};
```

Exemple :

```
constexpr int taille(5);

/* pas encore supporté par tous les *
 * compilateurs :-(*
 */
array<int, taille> ages (
    { 20, 35, 26, 38, 22 } );

// alternative :
array<int, taille> ages = {
    20, 35, 26, 38, 22
};
```

Âge
20
35
26
38
22

Un `array` non initialisé contient « n'importe quoi ».

Tableaux dynamiques

```
#include <vector>
vector<double> tab;
vector<double> tab2(5);
```

```
tab[i][j]
```

```
tab.size()
```

```
for(auto element : tab)
```

```
for(auto& element : tab)
```

```
tab.push_back(x);
```

```
tab.pop_back();
```

```
vector<vector<int>> tableau(
    { { 0, 1, 2, 3, 42 },
      { 4, 5, 6 },
      { 7, 8 },
      { 9, 0, 1 } }
);
```

Tableaux statiques

```
#include <array>
array<double, 5> tab;
```

```
—
```

```
—
```

```
array<array<int, 3>, 4> matrice =
{
    0, 1, 2 ,
    3, 4, 5 ,
    6, 7, 8 ,
    9, 0, 1 ,
};
```



Les tableaux de taille fixe



```
#include <array>
```

Déclaration : `array<type, taille> identificateur;`

Déclaration/Initialisation :

```
array<type, taille> identificateur =
    {val1, ... , valtaille};
```

Accès aux éléments : `tab[i]` i entre **0** et **taille-1**

Fonctions spécifiques :

`size_t tab.size()` : renvoie la taille

Tableau multidimensionnel :

```
array<array<type, nb_colonnes>, nb_lignes>
identificateur;
```

```
tab[i][j] = ...;
```

- ▶ Complément sur les types de base

Types élémentaires « avancés »

En plus des types composés, signalons qu'il existe aussi d'autres types élémentaires, dérivés des types élémentaires présentés.

Trois **modificateurs** peuvent être utilisés :

- ▶ pour les `int` et les `double`, on peut demander d'avoir une *plus grande précision* de représentation à l'aide du modificateur `long`, et même `long long` (C++11).

Exemple : `long int nb_etoiles;`

- ▶ pour les `int`, on peut aussi demander d'avoir une *moins grande précision* de représentation à l'aide du modificateur `short`.

Exemple : `short int nb_cantons;`

- ▶ pour les `int` (et les `char`), on peut demander de travailler avec des données *non signées*, à l'aide du modificateur `unsigned`.

Exemple : `unsigned int nb_cacahouetes;`

On peut bien sûr combiner :

```
unsigned long int nb_etoiles;  
unsigned short int nb_cantons;
```

Types élémentaires « avancés »

En C++, **la taille des types n'est pas spécifiée** dans la norme.

Seules indications :

- ▶ le plus petit type est `char`
- ▶ les inégalités suivantes sont toujours vérifiées sur les tailles mémoires :

`char ≤ short int ≤ int ≤ long int`

`double ≤ long double`

Cependant, les tailles généralement utilisées sont

8 bits pour les `char`

16 bits pour les `short int`

32 bits pour les `long int`

En **C++11**, existe(ro)nt également les types entiers de tailles définies : `int8_t`, `uint8_t`, ..., `int64_t`, `uint64_t`

Types élémentaires « avancés »

Ces choix typiques conduisent aux bornes suivantes (cf `numeric_limits` dans la bibliothèque `<limits>`, par exemple `numeric_limits<int>::min()`) :

type	min.	max.
<code>char</code>	-128	127
<code>unsigned char</code>	0	255
<code>short int</code>	-32768	32767
<code>unsigned short int</code>	0	65535
<code>long int</code>	-2147483648	2147483647
<code>unsigned long int</code>	0	4294967295

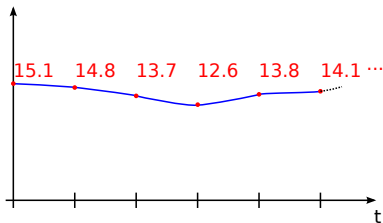
type	min. (valeur absolue)	max.	précision
<code>double</code>	2.22507e-308	1.79769e+308	2.22045e-16
<code>long double</code>	3.3621e-4932	1.18973e+4932	1.0842e-19

Note : « *précision* » correspond au plus petit nombre x tel que $1 + x \neq 1$.

- ▶ Analyse de données (températures)

Etude de cas analyse de données

On souhaite écrire un programme qui fasse des statistiques sur un ensemble de relevés de températures:



(moyenne, températures extrêmes etc.)

Pour préparer le prochain cours

- ▶ Vidéos et quiz du MOOC semaine 6 :
 - ▶ string : introduction [10:09]
 - ▶ string : traitements [12:37]
 - ▶ Typedef : alias de types [07:30]
 - ▶ Structures [24:18]

- ▶ Le prochain cours :
 - ▶ de 15h15 à 16h (résumé et quelques approfondissements)