

Information, Calcul et Communication

Composante Pratique: Programmation C++

MOOC sem5 : vector / array / tableau “à la C”

Tableau à la C, **vector** ou **array** ?

Rappels sur vector: définition et initialisation

La boîte à outil de **vector** : parcours, méthodes

Tableaux «à la C» et array à plusieurs indices en mémoire

Présentation du thème du projet

tableau à la C, vector ou array ?

Un **tableau** rassemble **plusieurs valeurs** de même **type** sous un même **identificateur**

Les éléments sont consécutifs en mémoire (au moins pour un tableau à 1 indice)

Un **tableau** est **dynamique** si le nombre de valeurs peut changer en cours d'exécution



`#include <vector>`

		Taille connue à la compilation ?	
		non	oui
Taille pouvant varier à l' exécution ?	oui	vector	(vector)
	non	(vector)	<ul style="list-style-type: none">• Array• tableau «à la C»

Rappels sur vector: définition et initialisation

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

- vide

```
vector<double> tab;
```

- avec un ensemble de valeurs initiales

```
vector<double> tab({ 2., 3.5, 2.6, 3.8, 22. });
```

2.	3.5	2.6	3.8	22.
----	-----	-----	-----	-----

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

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

0.	0.	0.	0.	0.
----	----	----	----	----

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

```
vector<double> tab(5, 1.0);
```

1.	1.	1.	1.	1.
----	----	----	----	----

- avec une copie d'un autre tableau de même type

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

- Définition d'un vector de **n** vector en indiquant un **tableau anonyme**

```
vector< vector<bool> > tab(n, vector<bool>(m));
```

avec n=3
et m=6

0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Accès direct aux éléments du tableau

Avec un indice entier commençant à **zéro** = «**décalage**» par rapport au début du tableau

ATTENTION_1: pour un tableau **tab** de **TAILLE** élément, les éléments sont rangés de **tab[0]** à **tab[TAILLE -1]**

ATTENTION_2: aucune vérification de la validité de L'indice n'est faite à la compilation ou à l'exécution

Un **débordement de tableau** peut avoir des conséquences immédiates (segmentation fault) ou plus sournoises (modification d'une autre variable).
=> **Comportement indéterminé**

Tableau **tab** de 5 double

adresse	valeur	
0x04	2 .	tab[0]
0x0C	3 . 5	tab[1]
0x14	2 . 6	tab[2]
0x1C	3 . 8	tab[3]
0x24	22 .	tab[4]

Accès direct aux éléments du tableau (2) le range-for

Une nouvelle structure de contrôle de boucle en plus de **while** et **for**: le **range for**

C++11:

```
for( auto element : tableau) //tableau n'est PAS modifiable
```

```
for( auto& element : tableau) //tableau est modifiable
```

// A privilégier si on parcourt un seul tableau à la fois

// et si on accède à un seul élément.

// on ne peut pas sauter des éléments.

// pour les AUTRES contextes, utiliser **for** quand la taille est connue et **while** sinon.

Exemple: lire_tab_dyn.cc



Fonctions spécifiques à vector: les méthodes

`#include <vector>` offre un ensemble de fonctions encore appelées «**méthodes**»

Syntaxe de l'appel d'une méthode: `nom_tableau.nom_methode(parametres)`

`tableau.size()` : renvoie la TAILLE de tableau (type de retour : `size_t`)
↳ unsigned int.

`tableau.front()` : renvoie une référence au 1er 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]`

Permet de
mettre en
œuvre le
concept
de Pile

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

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

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

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

`tableau1.swap(tableau2)` : échange le contenu des deux tableaux

Et les tableaux «à la C» dans tout ça ?

Propriété: garantit la réservation d'un unique bloc de mémoire pour ranger tous les éléments du tableau de manière séquentielle, *même pour un tableau à plusieurs indices*.

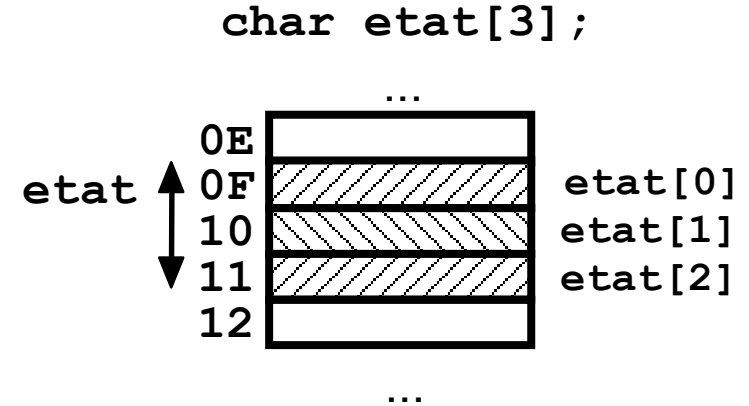
La syntaxe de l'accès à un élément `tab[i]` d'un tableau `tab` est la même que pour **vector** et **array** avec `i` compris entre 0 et `TAILLE-1`.

En mémoire `i` correspond au **décalage** (offset) par rapport à l'adresse du premier élément du tableau.

Mêmes risques de débordement d'indice →

Nombreuses autres faiblesses:

- ***taille fixe*** qui doit être connue à la déclaration.
- pas de méthode `size()`
- Manipulation *délicate* avec les fonctions

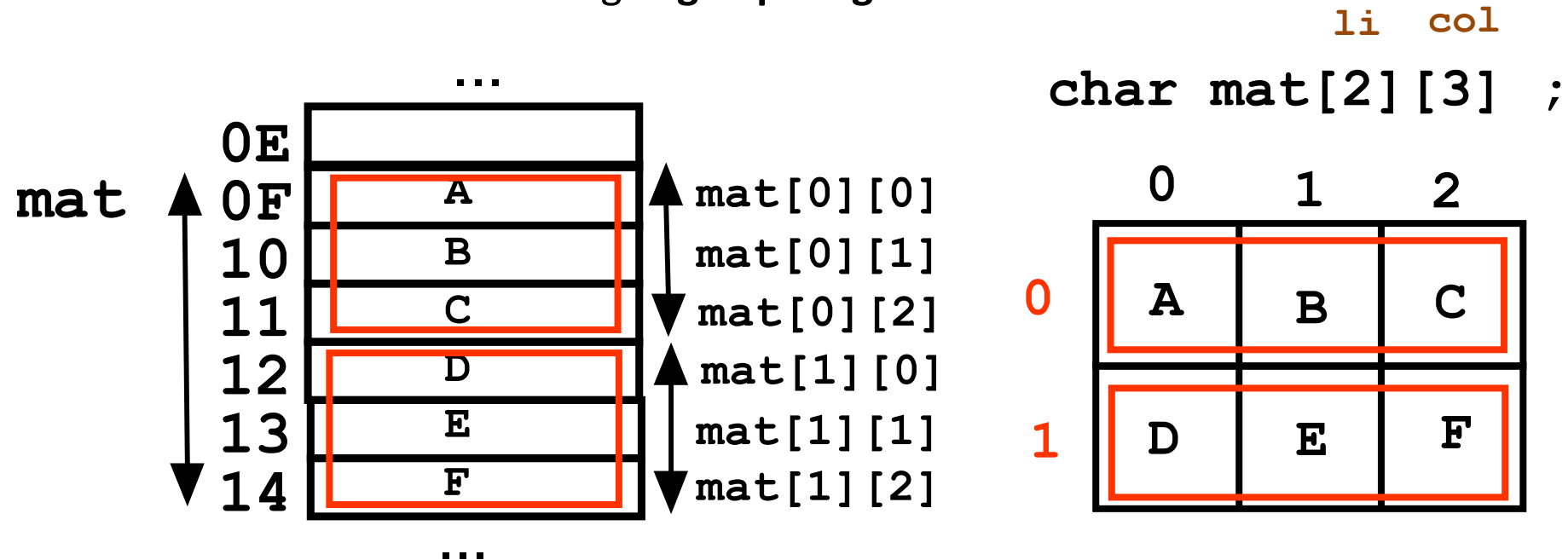


`etat[3]` ou `etat[-1]`
sont **en dehors** de l'espace
réservé pour le tableau `etat`

=> Segmentation fault ou
corruption d'une autre variable

Tableaux «à la C» et array à plusieurs indices en mémoire

Un tableau à 2 indices est rangé **ligne par ligne** en mémoire



Définition: l'élément $[i][j]$ du tableau est situé en mémoire à un emplacement qui est décalé d'un **offset** par rapport à l'adresse du premier élément :

L'offset de $[0][0]$ est zéro car c'est le premier élément du tableau

Règle générale: **offset de** $[i][j] = i * \text{TAILLE_LIGNE} + j$

Point fort : le calcul de l'offset de dépend pas de la taille du tableau = coût constant