# Cours d'introduction à la programmation (en C++)
# Types avancés I (en C++)

Jamila Sam
Jean-Cédric Chappelier
Vincent Lepetit

Faculté I&C

## Rencontre du 5e type

À ce stade du cours, la représentation des **données** se réduit aux types élémentaires `int`, `double`et `bool`.

Ils permettent de représenter, dans des *variables*, des concepts simples du monde modélisé dans le programme :
dimensions, sommes, tailles, expressions logiques, ...

Cependant, de nombreuses données **plus sophistiquées** ne se réduisent pas à un objet informatique élémentaire.

☞ un langage de programmation évolué doit donc fournir le moyen de **composer les types élémentaires** pour construire des types plus complexes, les *types composés*.

## Exemples de données structurées

| Âge |
|---|
| 20 |
| 35 |
| 26 |
| 38 |
| 22 |

| Nom | Taille | Âge | Sexe |
|---|---|---|---|
| Dupond | 1.75 | 41 | M |
| Dupont | 1.75 | 42 | M |
| Durand | 1.85 | 26 | F |
| Dugenou | 1.70 | 38 | M |
| Pahut | 1.63 | 22 | F |

- tableaux
- structures de données hétérogènes
  (par exemple, « un enregistrement » dans le tableau de droite ci-dessus)
- chaînes de caractères
  (par exemple, le « nom »)
- ...

## Petit exemple introductif

Supposons que l'on souhaite écrire un programme de jeu à plusieurs joueurs.

| Score | Écart à la moyenne |
|---|---|
| 1000 | -1860 |
| 1500 | -1360 |
| 2490 | -370 |
| 6450 | 3590 |
| ... | ... |

Commençons modestement par... deux joueurs.

## Solution avec les moyens actuels

```
int score1(calcule_score(....));
int score2(calcule_score(....));

// Calcul de la moyenne
int moyenne_joueurs(moyenne(score1, score2));

// Affichages
cout << "Score         Ecart Moyenne" << endl;
cout << score1 << "    " << score1 - moyenne_joueurs << endl;
cout << score2 << "    " << score2 - moyenne_joueurs << endl;
```

Comment passer à plus de joueurs ?

☞ utiliser plus de variables

## Solution avec les moyens actuels (2)

```
int score1(calcule_score(....));
int score2(calcule_score(....));
int score3(calcule_score(....));
int score4(calcule_score(....));
int score5(calcule_score(....));

// Calcul de la moyenne
int moyenne_joueurs(moyenne(score1, score2, score3, score4, score5));

// Affichages
cout << "Score         Ecart Moyenne" << endl;
cout << score1 << "    " << score1 - moyenne_joueurs << endl;
cout << score2 << "    " << score2 - moyenne_joueurs << endl;
cout << score3 << "    " << score3 - moyenne_joueurs << endl;
cout << score4 << "    " << score4 - moyenne_joueurs << endl;
cout << score5 << "    " << score5 - moyenne_joueurs << endl;
```

## Solution avec les moyens actuels (3)

```
int score1(calcule_score(....));
int score2(calcule_score(....));
int score3(calcule_score(....));
int score4(calcule_score(....));
int score5(calcule_score(....));

// Calcul de la moyenne
int moyenne_joueurs(moyenne(score1, score2, score3, score4, score5));

// Affichages
cout << "Score         Ecart Moyenne" << endl;
for(int i(1); i <= 5; ++i) {
  cout << scorei << "    " << scorei -  moyenne_joueurs << endl;
}
```

## Solution avec les moyens actuels : limites

Mais

1. comment l'écrire (`scorei` n'est pas correct) ?
2. comment faire si on veut considérer 100, 1000... joueurs ?
3. comment faire si le nombre de joueurs n'est pas connu au départ ?

☞ Solution : les **tableaux**

## Solution avec tableau

```
unsigned int nb_joueurs(5);
vector<int, nb_joueurs> scores;

for(auto score : scores) {
  score = calcule_score(....);
}

// Calcul de la moyenne
int moyenne_joueurs(moyenne(scores));

// Affichages
cout << "Score         Ecart Moyenne" << endl;
for(auto score : scores) {
  cout << score << "    " << score - moyenne_joueurs << endl;
}
```

## Les tableaux

Un **tableau** est une collection de valeurs *homogènes*,
c'est-à-dire constitué d'éléments qui sont tous du **même type**.

Exemple : Tableau `scores` contenant 4 `int`

| 1000 | 1500 | 2490 | 6450 |
|---|---|---|---|
| `scores[0]` | `scores[1]` | `scores[2]` | `scores[3]` |

On utilise donc les tableaux lorsque *plusieurs* variables de *même* type doivent être *stockées*/mémorisées.

On pourra définir des tableaux d'`int`, de `double`, de `bool`, ...
... mais aussi de n'importe quel autre type à disposition
par exemple des tableaux de tableaux.

## Les différentes sortes de tableaux

Il existe en général quatre sortes de tableaux :

| | | taille initiale connue *a priori* ? non | taille initiale connue *a priori* ? oui |
|---|---|---|---|
| taille pouvant varier lors de l'utilisation du tableau ? | oui | 1. | 2. |
| | non | 3. | 4. |

Remarques :

- avec le premier type de tableau (1.), on peut faire tous les autres
  ☞ les autres permettent des *optimisations*
- pratiquement aucun langage de programmation n'offre les 4 variantes

## Les tableaux en C++

En C++, on utilise :

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

Dans un premier temps, nous allons nous intéresser aux
**tableaux dynamiques** (`vector`)

Viendront ensuite les tableaux de taille fixe

## Les `vector`

Un **tableau dynamique**, est une *collection* de données homogènes, dont *le nombre peut changer* au cours du déroulement du programme, par exemple lorsqu'on ajoute ou retire des éléments au/du tableau.

Les tableaux dynamiques sont définis en C++ par le biais du type

`vector`

Pour les utiliser, il faut tout d'abord importer les définitions associées à l'aide d'un `include` :

```
#include <vector>
```

## Déclaration d'un tableau dynamique

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

```
vector<type> identificateur;
```

où *identificateur* est le nom du tableau et *type* correspond au type des éléments du tableau.

Exemple :

```
#include <vector>
...
vector<int> tableau;
```

Le type des éléments peut être n'importe quel type C++ valide.

## C++11 Initialisation d'un tableau dynamique

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

- vide
- avec un ensemble de valeurs initiales
- avec une taille initiale donnée et tous les éléments « nuls »
- avec une taille initiale donnée et tous les éléments à une même valeur donnée
- avec une copie d'un autre tableau

## C++11 Initialisation d'un tableau dynamique

Un tableau dynamique déclaré sans initialisation correspond à un **tableau vide** (**sans aucun** élément !). Il est initialisé comme tel par le compilateur.

```
vector<int> tableau;
```

---

Depuis la norme C++11 , l'initialisation des tableaux dynamiques s'est améliorée. On peut maintenant les initialiser avec des valeurs initiales différentes :

```
vector<type> identificateur({ val_1, ..., val_n });
```

Exemple :

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

Note : on peut aussi écrire

```
vector<int> ages = { 20, 35, 26, 38, 22 };
```

## Initialisation d'un tableau dynamique

Une taille initiale peut être indiquée si nécessaire.
La syntaxe de la déclaration/initialisation est alors :

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

Exemple :

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

correspond à la déclaration d'un tableau **initialement** constitué de 5 entiers, tous nuls.

**Attention !** Ce n'est **pas** la même chose que (à venir)

```
array<int, 5> tab;
```

car dans ce dernier cas on *ne* pourra *plus* changer la taille du tableau, alors que dans le cas de `vector` on le peut.
(et en plus le `array` n'est pas initialisé)

| | | taille initiale connue *a priori* ? non | taille initiale connue *a priori* ? oui |
|---|---|---|---|
| taille pouvant varier lors de l'utilisation du tableau ? | oui | `vector` | (`vector`) |
| | non | (`vector`) | `array` (C++11) tableaux « à la C » |

## Initialisation d'un tableau dynamique

La déclaration d'un tableau avec taille initiale peut en plus être associée à une initialisation explicite des éléments contenus dans le tableau, mais tous à la *même* valeur.
Cela s'écrit :

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

où *valeur* est la *même* valeur initiale affectée à tous les éléments du tableau

On peut aussi initialiser un tableau dynamique à l'aide d'une copie d'un autre tableau dynamique :

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

où *reference* est une référence à un tableau de même type de base *type*.

Exemples :

```
vector<int> tab1(5, 1);
vector<int> tab2(tab1);
```

correspondent toutes deux à la déclaration d'un tableau d'entiers dont les 5 éléments de départ sont initialisés à la valeur 1.

## 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);
```

## Affectation globale d'un tableau

Tout tableau (qui n'a pas été déclaré comme constant) peut être modifié par une **affectation globale** du tableau en tant que tel.

On parle ici de l'affectation d'un tableau *complet*, dans sa totalité.

La syntaxe est :

*tableau1* = *tableau2* ;

où *tableau2* est un tableau de même type que *tableau1*.

Exemple :

```
vector<int> tab1({ 1, 2, 3 });
vector<int> tab2;
...
tab2 = tab1 ; // copie de tout tab1 dans tab2
```

## Accès direct aux éléments d'un tableau

Le i+1ème élément d'un tableau `tab` est référencé par

`tab[i]`

**Attention !** Les indices correspondant aux éléments d'un tableau de taille `T` **varient entre** `0` **et** `T-1`

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

**Attention !** Il n'y a **pas de contrôle de débordement ! !**
Il est impératif que l'élément que vous référencez **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 :

- C++11 avec les itérations sur ensemble de valeurs

  `for(auto element : tableau)`

- avec une itération `for` « classique » :

  `for(size_t i(0); i < TAILLE; ++i)`

  ☞ *TAILLE* ? ? voir plus loin

- [avancé] avec des itérateurs (non présenté dans ce cours)

Laquelle choisir ?

- à chaque fois que c'est possible : la première
- sinon : la deuxième

## C++11 Itérations sur un tableau

- Si l'on ne veut pas modifier les éléments du tableau :

  `for(auto` *nom_de_variable* `:` *tableau*`)`

- Si l'on veut modifier les éléments du tableau :

  `for(auto&` *nom_de_variable* `:` *tableau*`)`

où *type* est le type des éléments contenus dans le tableau.

Exemples :

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

for(auto& age : ages) {
  cout << "Age de l'employé suivant ? ";
  cin >> age;
}

cout << "Age des employés : " << endl;
for(auto age : ages) {
  cout << "    " << age << endl;
}
```

## C++11 Itérations sur un tableau (2)

Notez que les itérations C++11
`for(auto nom_de_variable : tableau)`
ne permettent pas :

- d'itérer sur plusieurs tableaux à la fois :
  par exemple il n'est pas possible de traverser en une passe deux tableaux pour les comparer, les mélanger, ...
- d'accéder à plusieurs éléments
  par exemple on ne peut pas comparer un élément du tableau et son suivant
- de sauter des éléments, d'avancer de deux en deux, ...

## Itérations sur un tableau (3)

On peut aussi itérer sur un tableau avec un `for` classique... ...mais il faut pour cela connaitre la taille.

Il existe pour cela une « *fonction spécifique* », c'est-à-dire propre à chaque variable `vector` : la fonction `size()` :

`tab.size()` retourne la taille du tableau `tab`.
Cette taille est de type `size_t`, un « `int` » particulier, toujours positif.

On pourra donc écrire :

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

Exemple :

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

for(size_t i(0); i < ages.size(); ++i) {
  cout << "Age de l'employé " << i+1 << " ? ";
  cin >> ages[i];
}
```

## Quelques exemples de manipulation de tableaux

Soit un tableau déclaré par :

```
vector<double> tab(10);
```

Affichage du tableau :

- si l'on n'a pas besoin d'expliciter les indices :

```
cout << "Le tableau contient : ";
for(auto element : tab) {
  cout << element << " ";
}
cout << endl;
```

- si l'on veut expliciter les indices :

```
for(size_t i(0); i < tab.size(); ++i) {
  cout << "L'élément " << i << " vaut " << tab[i] << endl;
}
```

## Quelques exemples de manipulation de tableaux

(ré)Affectation de tous les éléments à la valeur 1.2 :

```
for(auto& el : tab) {
  el = 1.2;
}
```

ou alors

```
tab = vector<double>(tab.size(), 1.2);
```

```
vector<double> tab2(tab.size(), 1.2);
tab = tab2;
```

## Quelques exemples de manipulation de tableaux

Saisie au clavier des éléments du tableau :

- si l'on n'a pas besoin d'expliciter les indices :

```
for(auto& element : tab) {
  cout << "Entrez l'élément suivant :" << endl;
  cin >> element;
}
```

- si l'on veut expliciter les indices :

```
for(size_t i(0); i < tab.size(); ++i) {
  cout << "Entrez l'élément " << i <<  ":" << endl;
  cin >> tab[i];
}
```

## Fonctions spécifiques

Un certain nombre d'opérations sont **directement attachées** au type `vector`.

L'utilisation de ces opérations spécifiques se fait avec la syntaxe suivante :

*nom_de_tableau.nom_de_fonction(arg1, arg2, ...);*

Exemple :

```
vector<double> mesures;
size_t nombre_de_mesures(0);
...
nombre_de_mesures = mesures.size();
...
```

## Fonctions spécifiques

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

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

`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]`

`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 retour.

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

## `push_back` et `pop_back`

```
vector<double> v(3, 4.5);

v.pop_back();
v.push_back(5.6);
v.push_back(6.7);
v.pop_back();
```

## Exemple

Ecrivons une fonction qui (ré)initialise un tableau dynamique d'entiers en les demandant à l'utilisateur, qui peut

- ajouter des nombres strictement positifs au tableau
- recommencer au début en entrant *0*
- effacer le dernier élément en entrant un nombre négatif

```
Saisie de 3 valeurs :
Entrez la valeur 0 : 5
Entrez la valeur 1 : 2
Entrez la valeur 2 : 0
Entrez la valeur 0 : 7
Entrez la valeur 1 : 2
Entrez la valeur 2 : -4
Entrez la valeur 1 : 4
Entrez la valeur 2 : 12

-> 7 4 12
```

## Exemple

Ecrivons une fonction qui (ré)initialise un tableau dynamique d'entiers en les demandant à l'utilisateur, qui peut

- ajouter des nombres strictement positifs au tableau
- recommencer au début en entrant *0*
- effacer le dernier élément en entrant un nombre négatif

```
vector<int> tab;

saisie(tab, 5); // saisie de 5 éléments

saisie(tab); // saisie de 4 éléments

vector<int> tab2(12);

saisie(tab2, tab2.size());
```

## Exemple

Ecrivons une fonction qui (ré)initialise un tableau dynamique d'entiers en les demandant à l'utilisateur, qui peut

- ajouter des nombres strictement positifs au tableau
- recommencer au début en entrant *0*
- effacer le dernier élément en entrant un nombre négatif

```
void saisie(vector<int>& vect, size_t taille = 4)
{
  vect.clear();
  cout << "Saisie de " << taille << " valeurs :" << endl;
  while (vect.size() < taille) {
    cout << "Entrez la valeur " << vect.size() << " : ";
    int val;
    cin >> val;
    if ((val < 0) and (not vect.empty())) { vect.pop_back();     }
    else if (val == 0)                    { vect.clear();        }
    else if (val > 0)                     { vect.push_back(val); }
  }
}
```

## Les tableaux multidimensionnels

Comment déclarer un tableau à plusieurs dimensions ?

☞ On ajoute simplement un niveau de plus :

C'est en fait un **tableau de tableaux**...

Exemple :

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

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

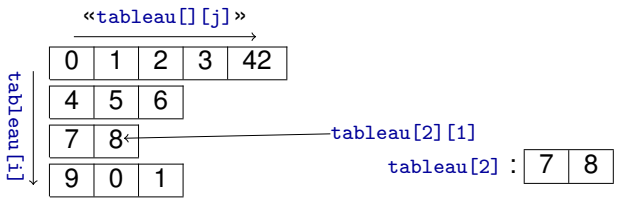
`tab[i]` est donc un « `vector<int>` », c'est-à-dire un tableau dynamique d'entiers (qui, au départ, en contient 6)

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

## Les tableaux multidimensionnels (2)

On a l'habitude de se représenter `tab[i][j]` comme l'élément de la `(i+1)`ème ligne et de la `(j+1)`ème colonne.

Mais attention ! Un `vector<vector<int>>` n'est pas une matrice, mais un tableau dynamique de tableaux dynamiques d'entiers (pas nécessairement tous de la même taille !).




## Code de l'exemple (version C++11)

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

for(auto ligne : tableau) {
  for(auto element : ligne) {
    cout << element << " ";
  }
  cout << endl;
}

for(size_t i(0); i < tableau.size(); ++i) {
  cout << "tableau[" << i <<"].size()="
       << tableau[i].size() << endl;
}
```

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 42 |
| 4 | 5 | 6 | | |
| 7 | 8 | | | |
| 9 | 0 | 1 | | |

## C++11 Tableaux de taille fixe

Nous avons jusqu'ici vu les tableaux dynamiques (c'est-à-dire dont la taille varie au cours du déroulement du programme).

Que faire dans les cas particuliers où la taille

- est *connue à l'avance*, *avant* que le programme ne commence, et
- ne va pas changer ; ne dépend pas du déroulement du programme.

☞ **tableaux de taille fixe**

Depuis la norme C++11, il existe *deux* sortes de tableaux de taille fixe :

- les « anciens tableaux », « à la C » (*build-in array*) ;
- les `array` de la (nouvelle) bibliothèque standard, conçus pour être plus faciles d'utilisation.

## 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 ils resteront malgré tout certainement assez répandus (inertie)... :-(

Pour ceux que cela intéresse : voir annexe à ce cours.

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

En C++11, le type « tableau de taille fixe » est défini dans la bibliothèque `array`.

Pour les utiliser, il faut ajouter en début de programme :

```
#include <array>
```

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** au moment où on écrit le programme (→ sinon `vector`)

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

Exemples :

```
#include <array>
...
array<double, 3> vecteur_3d;
```

correspond à la déclaration d'un *tableau de 3* `double`.

```
const size_t nb_cantons(26);
array<unsigned int, nb_cantons> habitants;
```

correspond à la déclaration d'un tableau de 26 entiers.

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

Comme pour les tableaux dynamiques, un tableau de taille fixe peut être initialisé directement lors de sa déclaration :

```
array<type, taille> identificateur({val_1, ... , val_taille});
```
ou
```
array<type, taille> identificateur = {val_1, ... , val_taille};
```

Exemple :

```
const size_t taille(5);

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

| Âge |
|---|
| 20 |
| 35 |
| 26 |
| 38 |
| 22 |

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

## Utilisations d'un tableau de taille fixe

L'accès aux éléments d'un tableau de taille fixe se fait de la même façon que pour un tableau dynamique :

- directement : `tab[i]`
- par itération C++11 : `for(auto element : tableau)`
- itération `for` « classique »

Les tableaux de taille fixe `array` ont aussi une « fonction spécifique » `size()` qui renvoie leur taille.

On peut également faire des affectations globales de tableaux de taille fixe `array` :

```
array<int, 3> tab1 = { 1, 2, 3 };
array<int, 3> tab2 ;
...
tab2 = tab1 ; // copie de tab1 dans tab2
```

## Tableaux à plusieurs dimensions

Comme pour les tableaux dynamiques, on peut déclarer des tableaux de taille fixe multidimentionnels.

(on peut même faire des tableaux dynamiques de tableaux de taille fixe, des tableaux de taille fixe de tableaux dynamiques, etc.)

Exemples :

```
array<array<double, 2>, 2> rotation;
array<array<int, nb_statistiques>, nb_cantons> statistiques;
array<array<array<double, 4>, 2>, 3> tenseur;

rotation[1][0] = 0.231;
```

`statistiques[i]` est un « `array<int, nb_statistiques>` », c'est-à-dire un tableau de `nb_statistiques` entiers

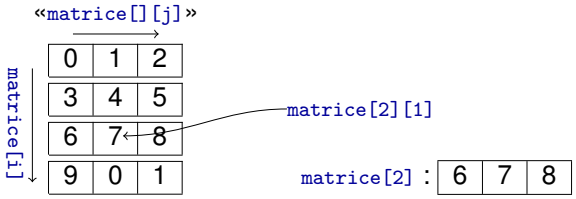
☞ `statistiques` est bien un tableau de tableaux.

## Tableaux à plusieurs dimensions (2)

Les tableaux multidimensionnels peuvent également être initialisés lors de leur déclaration.
Il faut bien sûr spécifier autant de valeurs que le produit des dimensions.

Exemple :

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




## Pour résumer

| Tableaux dynamiques | Tableaux statiques |
|---|---|
| `#include <vector>` | `#include <array>` |
| `vector<double> tab;` | `array<double, 5> 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 } } );` | `array<array<int, 3>, 4> matrice = { 0, 1, 2 , 3, 4, 5 , 6, 7, 8 , 9, 0, 1 };` |