Cours 5

Les tableaux

Vincent Lepetit

vincent.lepetit@epfl.ch

Supposons qu'on souhaite écrire un programme qui lit 5 valeurs avant d'en afficher les carrés:

```
Entrez 5 nombres entiers:
11
9
14
25
63
NOMBRE   CARRE
11       121
9        81
14       196
25       625
63       3969
```

Le programme doit *conserver* les 5 valeurs fournies avant de commencer l'affichage.

→ *une seule variable n'est pas suffisante.*

On pourrait utiliser 5 variables:

```
int nombre1, nombre2, nombre3, nombre4, nombre5;

cout << "Entrez 5 nombres entiers:" << endl;
cin >> nombre1;
cin >> nombre2;
cin >> nombre3;
cin >> nombre4;
cin >> nombre5;

cout << "NOMBRE CARRE" << endl;
cout << nombre1 << " " << nombre1 * nombre1 << endl;
cout << nombre2 << " " << nombre2 * nombre2 << endl;
cout << nombre3 << " " << nombre3 * nombre3 << endl;
cout << nombre4 << " " << nombre4 * nombre4 << endl;
cout << nombre5 << " " << nombre5 * nombre5 << endl;
```

Mais comment faire si:

- ***on veut considérer 100, 1000… valeurs ?***
- ***on veut laisser l'utilisateur choisir le nombre de valeurs ?***

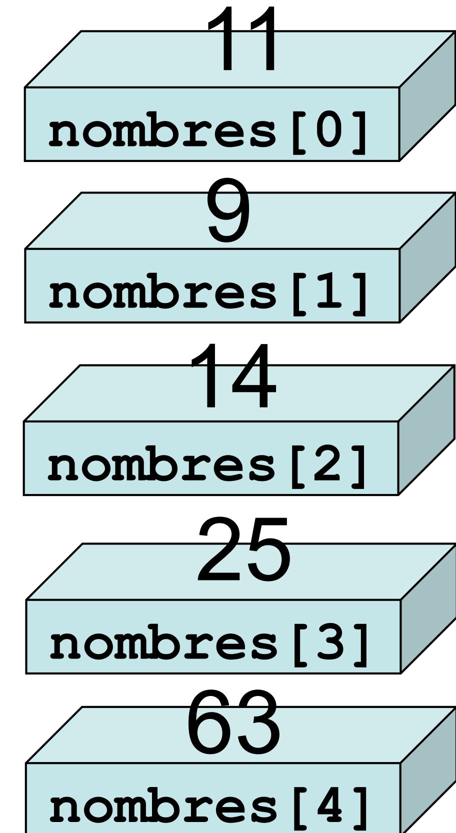
Les tableaux

Pour résoudre un tel problème, on utilise un *tableau*:

- Un tableau regroupe plusieurs valeurs sous un seul nom (`nombres` dans l'exemple qui suit).

- Chaque valeur, appelée élément, est repérée par ce nom suivi d'un **indice** entre crochets `[ ]`

Exemple: `nombres[0]`, `nombres[1]`, ... `nombres[4]`



Le même programme avec un tableau...

```
int nombres[5];
```

Déclaration du tableau

```
cout "Entrez 5 nombres entiers:" << endl;
cin >> nombres[0];
cin >> nombres[1];
cin >> nombres[2];
cin >> nombres[3];
cin >> nombres[4];

cout << "NOMBRE CARRE" << endl;
cout << nombres[0] << "  " << nombres[0] * nombres[0] << endl;
cout << nombres[1] << "  " << nombres[1] * nombres[1] << endl;
cout << nombres[2] << "  " << nombres[2] * nombres[2] << endl;
cout << nombres[3] << "  " << nombres[3] * nombres[3] << endl;
cout << nombres[4] << "  " << nombres[4] * nombres[4] << endl;
```

Eléments du tableau: s'utilisent comme des variables

...plus une boucle

Il faut bien sûr utiliser des boucles puisqu'on répète 5 fois des instructions similaires:

```
int nombres[5];

cout << "Entrez 5 nombres entiers:" << endl;
for(int i = 0; i < 5; i++)
  cin >> nombres[i];

cout << "NOMBRE CARRE" << endl;
for(int i = 0; i < 5; i++)
  cout << nombres[i] << "   " << nombres[i] * nombres[i]
<< endl;
```

On peut maintenant facilement considérer 100, 1000… valeurs: il suffit de remplacer le 5 par 100, 1000…

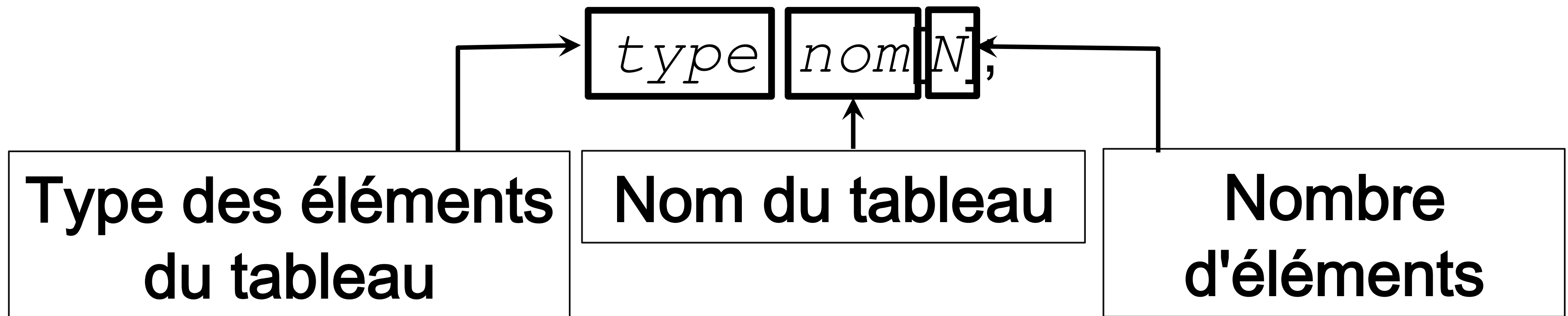
Déclaration de tableaux

Comme une variable, un tableau doit être déclaré en début de programme, pour que le compilateur connaisse

- le type des valeurs contenues par le tableau, et
- le nombre de valeurs:

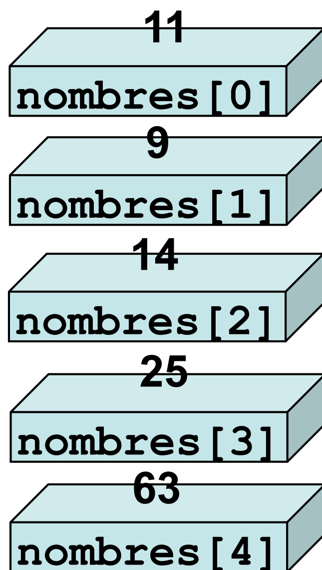
type nom[*N*];

Déclaration de tableaux

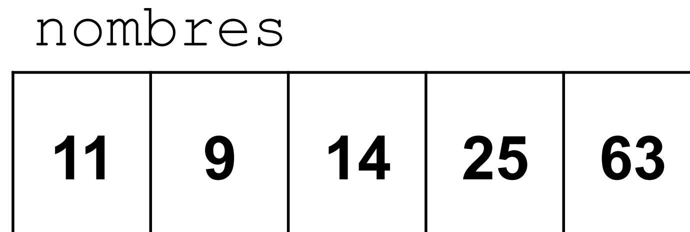


Représentation

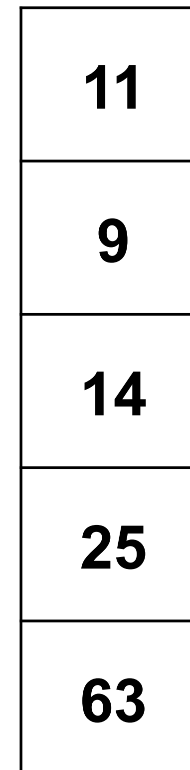
`int nombres[5];`



OU



OU



Exemples de déclaration

Par exemple:

```
int nombres[5];
float T[10];
bool tests[100];
```

On peut déclarer plusieurs tableaux et des variables en une seule instruction:

```
int tab[100], tab2[50], n;
```

On peut initialiser un tableau lors de sa déclaration. Par exemple:

```
int tab[5] = {10, 20 , 5, 0, 3};
```

place la valeur `10` dans `tab[0]`, `20` dans `tab[1]`, etc...

Attention

- La taille du tableau (le nombre de valeurs) doit être connue à la compilation.

  ***On ne peut pas faire**:*

```
int n = 5;
int nombres[n]; // !!!
```

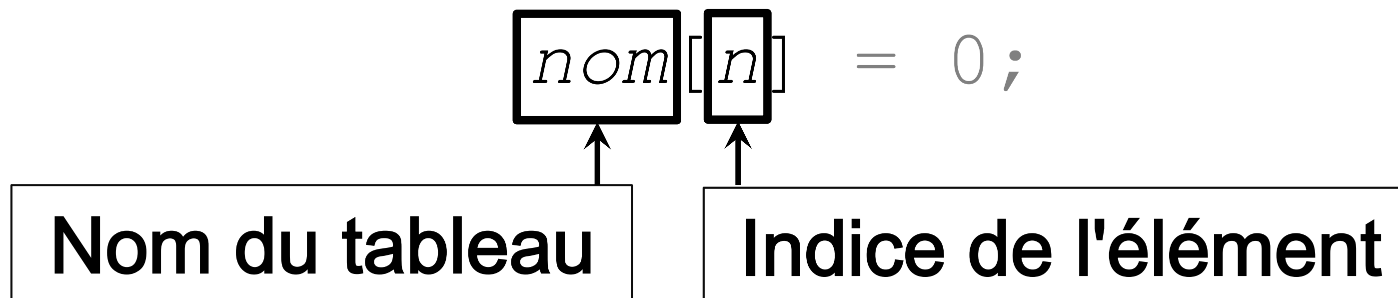
- La taille d'un tableau ne peut pas changer au cours du programme.

Manipuler les éléments d'un tableau

```
nom[n] = 0;
```

Un **élément** d'un tableau s'emploie exactement comme une variable.

Manipuler les éléments d'un tableau



un élément

```
nom[n] = 0;
```

Ne pas confondre:
- la déclaration d'un tableau et
- l'accès à un élément

Les deux s'écrivent avec le nom du tableau et une valeur entre crochets, mais ils font des choses très différentes.

Déclaration d'un tableau:

```
int nombres[5];
```

5 est la taille du tableau: on déclare un tableau de 5 éléments.

Accès à un élément:

```
nombres[2] = 0;
```

2 est l'indice de l'élément: on affecte 0 à l'élément d'indice 2.

Manipuler les éléments d'un tableau

Un **élément** d'un tableau s'emploie exactement comme une variable.

Notamment, on peut:

- lui affecter une valeur:

```
nombres[2] = 0;
```

- l'utiliser dans une expression arithmétique:

```
n = nombres[2] * nombres[3] + 1;
```

- l'utiliser dans un `cout` ou un `cin`:

```
cout << nombres[2];
cin >> T[3];
```

Les indices

Un **indice** est un numéro d'élément de tableau qu'on met entre crochets. Dans l'exemple:

```
a = nombres[2];
```

2 est utilisé comme indice.

En C et en C++, les tableaux commencent à l'indice 0:

- Le premier élément d'un tableau est repéré par l'indice 0;
- Le dernier élément d'un tableau de taille `n` est donc repéré par l'indice `n-1`.
- Par exemple, l'élément d'indice 2 est le troisième élément du tableau.

L'**indice** peut être:

- une constante (`nombres[2]`) ou une variable (`nombres[i]`), mais aussi
- une expression arithmétique: `nombres[2 * i + 1]`.

mais doit être de type `int`:

par exemple, on ne peut pas utiliser `nombres[y]` si `y` a été déclarée de type `float`.

Par contre, on peut utiliser `nombres[int(y)]`.

Un tableau en C commence à l'indice `0`.

L'indice du dernier élément d'un tableau de taille `N` est `N-1` (et non pas `N`).

Avec la déclaration:

```
int nombres[5];
```

`nombres[0]` est le premier élément du tableau;
`nombres[4]` est le dernier élément.

Quelques exemples de manipulation de tableaux

Affichage des éléments d'un tableau

Soit un tableau déclaré par:

```
float tab[5] = {1.2, 5.1, 7.6, -3.9, 1.12};
```

On ne peut pas afficher le tableau en une seule instruction.

On peut utiliser un boucle `for`:

```
for(int i = 0; i < 5; i++)
  cout << "L'element " << i << " contient: " << tab[i] << endl;
```

ce qui affichera:

```
L'element 0 contient 1.2
L'element 1 contient 5.1
L'element 2 contient 7.6
...
```

tab
1.2
5.1
7.6
-3.9
1.12

Initialisation de tous les éléments à 0

Initialisation de tous les éléments à 0:

```
for(int i = 0; i < 5; i++)
  tab[i] = 0;
```

tab

0
0
0
0
0

Lire les éléments d'un tableau au clavier

Comme pour l'affichage, on ne peut pas lire le tableau en une seule instruction.

On peut utiliser un boucle `for`:

```
for(int i = 0; i < 5; i++)
 {
   cout << "Entrez la valeur de l'element " << i << endl;
    cin >> tab[i];
 }
```

tab
9.8
8.3
3.4
2.1
5.2

Calculer la somme des éléments d'un tableau

On peut réutiliser la technique vue lors de la leçon 4:

U_0 = *valeur initiale*

et

$U_n = f(U_{n-1})$ ou $U_n = f(U_{n-1}, n)$

Par exemple, si on veut simplement calculer les N premières valeurs de (U_n), le code sera alors de la forme:

```
type U = valeur initiale;
for(int n = 1; n <= N; n++)
{
   ...
   U = f(U, n);
   ...
}
```

Calculer la somme des éléments d'un tableau

Dans ce cas, la suite (U_n) est:

$U_0 = 0$
et
$U_n = U_{n-1} + \texttt{tab}[n]$

Le code pour calculer la somme des éléments de `tab` est donc:

```
float U = 0;
for(int n = 0; n < 5; n++)
  U = U + tab[n];
```

tab
9.8
8.3
3.4
2.1
5.2

Attention aux bornes de la boucle `for`:

`n` commence à 0 pour compter le premier élément de `tab` dans la somme;

`n` va jusque 5 non compris pour ne pas sortir du tableau.

Quand utiliser un tableau ?

En gros, quand plusieurs valeurs du même type doivent être stockées.

Dans le cas du premier exemple, il fallait utiliser un tableau car on voulait que le programme affiche les valeurs après que l'utilisateur les aie toutes tapées: il fallait stocker les valeurs.

Quand ne pas utiliser de tableau ?

Si le programme devait afficher le carré à chaque fois que l'utilisateur entre une valeur:

```
Entrez un nombre entier:
4
Son carre est 16.
Entrez un nombre entier:
3
Son carre est 9.
...
```

les valeurs n'avaient pas à être stockées, et un tableau n'aurait pas été nécessaire. Une seule variable aurait suffi.

Les tableaux n'avaient pas été nécessaires pour les exercices de calcul de suites de la séance précédente:

```
U = 1;
for(int i = 0; i < 10; i++)
{
  U = U / 2;
  cout << U << endl;
}
```

Faciliter la modification de la taille d'un tableau

Reprenons le premier exemple:

```
#include <iostream>
using namespace std;

int main(int argc, char ** argv)
{
  int nombres[5];

  cout << "Entrez 5 nombres entiers:" << endl;
  for(int i = 0; i < 5; i++)
    cin >> nombres[i];

  cout << "NOMBRE CARRE" << endl;
  for(int i = 0; i < 5; i++)
    cout << nombres[i] << "  "
         << nombres[i] * nombres[i] << endl;

  return 0;
}
```

La valeur 5 apparaît 4 fois:

Si on veut modifier le programme pour qu'il lise 10 valeurs, il faut modifier le programme à 4 endroits, sans en oublier.

Le mot-clé `const`

Le mot-clé `const` permet de définir une constante:

```
const int nb_entiers = 5;
```

qui peut être utilisée pour définir la taille d'un tableau. Contrairement à une variable, on ne peut pas changer la valeur d'une constante, seulement utiliser sa valeur.

```
#include <iostream>
using namespace std;

int main(int argc, char ** argv)
{
  const int nb_entiers = 5;
  int nombres[nb_entiers];

  cout << "Entrez " << nb_entiers << " nombres entiers:"
       << endl;
  for(int i = 0; i < nb_entiers; i++)
    cin >> nombres[i];

  cout << "NOMBRE CARRE" << endl;
  for(int i = 0; i < nb_entiers ; i++)
    cout << nombres[i] << "  "
         << nombres[i] * nombres[i] << endl;

  return 0;
}
```

Quand la taille du tableau n'est pas connue

Supposons qu'on souhaite laisser l'utilisateur choisir le nombres de valeurs à entrer:

```
Combien de valeurs ?
3
Entrez 3 nombres entiers:
11
9
14
NOMBRE        CARRE
11            121
9             81
14            196
```

Ici, la taille du tableau n'est pas connue à l'avance. En langages C et C++, il y a deux solutions:

- une solution simple: déclarer un (grand) tableau de taille suffisante. Cette solution gaspille évidemment de la mémoire.

- une solution qui consiste à allouer dynamiquement le tableau, c'est-à-dire la place en mémoire du tableau sera définie lors de l'exécution du programme.

En C++, cela se fait en utilisant `new` et `delete`. Nous verrons cela dans les prochaines leçons.

Quand la taille du tableau n'est pas connue

Nous allons pour l'instant utiliser la première solution:

```
#include <iostream>
using namespace std;

int main(int argc, char ** argv)
{
  // Un tableau suffisamment grand pour y mettre toutes les valeurs:
  const int nb_max = 100;
  int nombres[nb_max];
  int nb;

  // Il faudrait egalement verifier que l'utilisateur entre
  // un nombre positif t inferieur a nb_max:
  cout << "Combien de valeurs ?" << endl;
  cin >> nb;

  cout << "Entrez " << nb << " nombres entiers:" << endl;
  for(int i = 0; i < nb; i++)
    cin >> nombres[i];

  cout << "NOMBRE CARRE" << endl;
  for(int i = 0; i < nb; i++)
    cout << nombres[i] << "  "
         << nombres[i] * nombres[i] << endl;

  return 0;
}
```

Attention à ne pas sortir du tableau

Il n'y a pas de vérification que l'indice soit correct.

Par exemple:

```
int T[5];


T[5] = 0; // !!
```

provoquera une erreur un moment ou un autre.

Attention à ne pas sortir du tableau

```
int T[5];
T[5] = 0; // !!
```

Aucune erreur ou *warning* n'est signalée à la compilation.

A l'exécution, cette erreur *peut* provoquer l'arrêt du programme avec un message ressemblant à:

`Segmentation fault`

ce qui signifie que le programme a essayé d'accéder à une partie de la mémoire qui ne lui appartient pas.

PIRE: **Il se peut aussi qu'il écrase une valeur qui lui appartienne, ce qui ne provoque pas de *`Segmentation fault`*, mais qui empêchera son déroulement correct.**

Attention donc aux boucles qui peuvent faire sortir du tableau, par exemple:

```
for(int i = 0; i <= 5; i++)
  T[i] = 0;
```

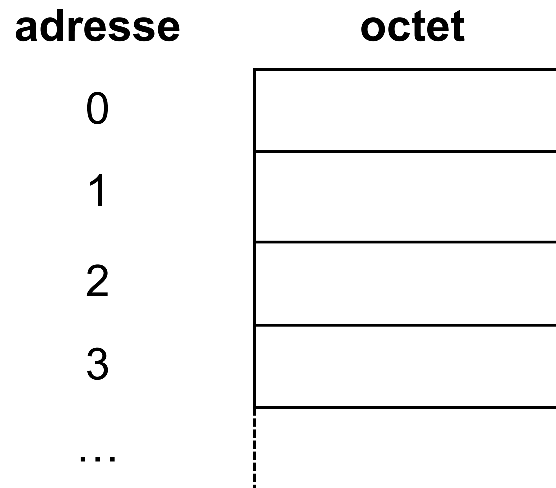
Que se passe-t'il exactement quand on exécute:

```
int T[5];
T[5] = 0; // !!
```

?

Mémoire

La mémoire d'un ordinateur est composée d'un grand nombre **d'octets** (une valeur entre 0 et 255) :



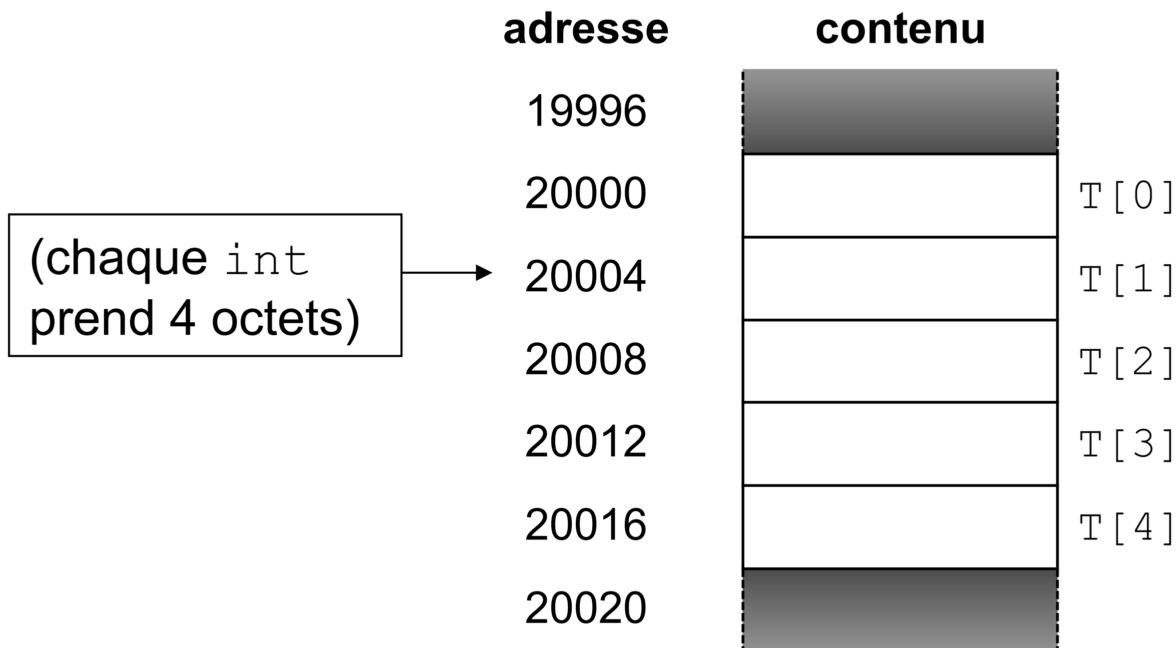
Chaque octet est repéré par une ***adresse***, qui est ***un nombre entier***.

La déclaration d'un tableau en mémoire

Les éléments d'un tableau sont placés consécutivement en mémoire.
Par exemple, la déclaration:

```
int T[5];
```

réserve de la place pour 5 `int` en mémoire (l'adresse 20000 est arbitraire):



Les éléments d'un tableau en mémoire

Si on fait:

```
T[0] = 3;
T[1] = 5;
T[2] = 17;
T[3] = 6;
T[4] = 1;
```

on obtient en mémoire:

adresse	contenu	
19996		
20000	3	T[0]
20004	5	T[1]
20008	17	T[2]
20012	6	T[3]
20016	1	T[4]
20020		

Attention à ne pas sortir du tableau

Quand on exécute

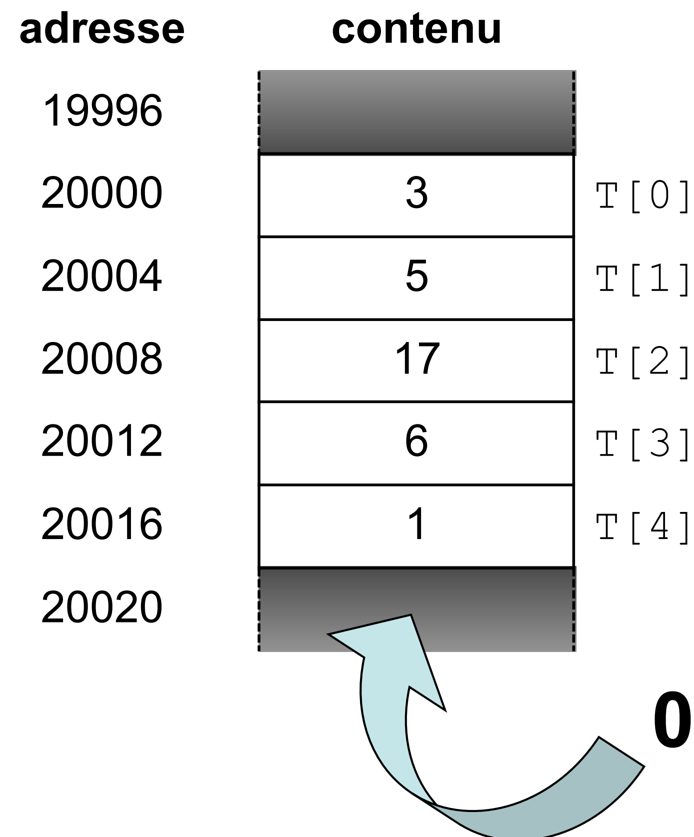
```
T[5] = 0; // !!
```

on essaie d'écrire la valeur 0 à un endroit de la mémoire en dehors du tableau.

Même problème pour

```
int a = T[5];
```

on essaie de lire une valeur en mémoire qui ne correspond pas à un élément du tableau.



Que veut dire le message `Segmentation fault` ?

Le message `Segmentation fault` n'est pas affiché par votre programme mais par le système d'exploitation (Linux donc) qui a détecté que votre programme faisait une opération interdite, et qui arrête le programme et signale l'erreur.

Chaque programme dispose d'une partie seulement de la mémoire de l'ordinateur, il ne peut pas écrire ou lire en dehors de cette mémoire qui lui est réservée.

Si on fait:

```
int T[5];
T[5] = 0; // !!
```

ou

```
int a = T[5];
```

il est fort probable qu'on essaie d'écrire ou de lire une partie de la mémoire qui n'appartient pas au programme, et ces opérations provoqueront l'arrêt du programme, et l'affichage du message `Segmentation fault`.

Pas de chance

Il se peut qu'en faisant:

```
int T[5];
T[5] = 0; // !!
```

on écrive néanmoins dans une partie de la mémoire qui appartient au programme, en dehors du tableau `T`, mais dans une variable ou un autre tableau du programme.

Dans ce cas:

→ il n'y a pas de `Segmentation fault`;

→ on écrase des données du programme;

→ le programme risque fort de ne pas se comporter correctement.

Que faire si votre programme ne fonctionne pas correctement, ou s'il provoque le message `Segmentation fault`?

On peut afficher les valeurs des différentes variables manipulées par le programme pour s'aider à comprendre le problème. Par exemple, le code suivant provoque un `Segmentation fault`:

```
int a = 7, b = 8;
int T[5];
for(int i = 0; i < 1000; i ++)
  T[i] = 0;
```

Pour comprendre ce qui se passe, on peut afficher les valeurs de l'indice `i`, et des variables `a` et `b`. Remplaçons la boucle par:

```
for(int i = 0; i < 1000; i++)
{
  T[i] = 0;
  cout << "i = " << i << ", a = " << a << ", b = " << b << endl;
}
```

```
int a = 7, b = 8;
int T[5];
for(int i = 0; i < 1000; i++)
{
  T[i] = 0;
  cout << "i = " << i << ", a = " << a << ", b = " << b << endl;
}
```

adresse	contenu	
19996		
20000	3	T[0]
20004	5	T[1]
20008	17	T[2]
20012	6	T[3]
20016	1	T[4]
20020	7	a
20024	8	b
20028		

```
int a = 7, b = 8;
int T[5];
for(int i = 0; i < 1000; i++)
{
  T[i] = 0;
  cout << "i = " << i << ", a = " << a << ", b = " << b << endl;
}
```

Le programme affiche *sur mon ordinateur*:

```
i = 0, a = 7, b = 8
```

adresse	contenu	
19996		
20000	~~3~~ 0	T[0]
20004	5	T[1]
20008	17	T[2]
20012	6	T[3]
20016	1	T[4]
20020	7	a
20024	8	b
20028		

```
int a = 7, b = 8;
int T[5];
for(int i = 0; i < 1000; i++)
{
  T[i] = 0;
  cout << "i = " << i << ", a = " << a << ", b = " << b << endl;
}
```

Le programme affiche *sur mon ordinateur*:

```
i = 0, a = 7, b = 8
i = 1, a = 7, b = 8
i = 2, a = 7, b = 8
i = 3, a = 7, b = 8
i = 4, a = 7, b = 8
```

adresse	contenu	
19996		
20000	**0**	T[0]
20004	**0**	T[1]
20008	**0**	T[2]
20012	**0**	T[3]
20016	**0**	T[4]
20020	5	a
20024	6	b
20028		

```
int a = 7, b = 8;
int T[5];
for(int i = 0; i < 1000; i++)
{
  T[i] = 0;
  cout << "i = " << i << ", a = " << a << ", b = " << b << endl;
}
```

Le programme affiche *sur mon ordinateur*:

```
i = 0, a = 7, b = 8
i = 1, a = 7, b = 8
i = 2, a = 7, b = 8
i = 3, a = 7, b = 8
i = 4, a = 7, b = 8
i = 5, a = 0, b = 8
```

adresse	contenu	
19996		
20000	**0**	T[0]
20004	**0**	T[1]
20008	**0**	T[2]
20012	**0**	T[3]
20016	**0**	T[4]
20020	~~5~~ 0	a
20024	6	b
20028		

```
int a = 7, b = 8;
int T[5];
for(int i = 0; i < 1000; i++)
{
  T[i] = 0;
  cout << "i = " << i << ", a = " << a << ", b = " << b << endl;
}
```

Le programme affiche *sur mon ordinateur*:

```
i = 0, a = 7, b = 8
i = 1, a = 7, b = 8
i = 2, a = 7, b = 8
i = 3, a = 7, b = 8
i = 4, a = 7, b = 8
i = 5, a = 0, b = 8
i = 6, a = 0, b = 0
```

adresse	contenu	
19996		
20000	**0**	T[0]
20004	**0**	T[1]
20008	**0**	T[2]
20012	**0**	T[3]
20016	**0**	T[4]
20020	**0**	a
20024	**0**	b
20028		

```
int a = 7, b = 8;
int T[5];
for(int i = 0; i < 1000; i++)
{
  T[i] = 0;
  cout << "i = " << i << ", a = " << a << ", b = " << b << endl;
}
```

Le programme affiche *sur mon ordinateur*:

```
i = 0, a = 7, b = 8
i = 1, a = 7, b = 8
i = 2, a = 7, b = 8
i = 3, a = 7, b = 8
i = 4, a = 7, b = 8
i = 5, a = 0, b = 8
i = 6, a = 0, b = 0
i = 7, a = 0, b = 0
```

adresse	contenu	
19996		
20000	**0**	T[0]
20004	**0**	T[1]
20008	**0**	T[2]
20012	**0**	T[3]
20016	**0**	T[4]
20020	**0**	a
20024	**0**	b
20028	**0**	

Le programme affiche *sur mon ordinateur*:

```
i = 0, a = 7, b = 8
i = 1, a = 7, b = 8
i = 2, a = 7, b = 8
i = 3, a = 7, b = 8
i = 4, a = 7, b = 8
i = 5, a = 0, b = 8
i = 6, a = 0, b = 0
i = 7, a = 0, b = 0
 ...
i = 517, a = 0, b = 0
Segmentation fault
```

- De `i` égal à 12 jusque 516, `T[i] = 0;` écrase d'autres parties de la mémoire du programme qu'on ne peut pas visualiser ici.

- Quand `i` vaut 517, `T[i] = 0;` essaie d'accéder à une partie de la mémoire qui n'appartient pas au programme. Le système d'exploitation arrête alors le programme, et affiche le `Segmentation fault`.

adresse	contenu	
19996		
20000	**0**	T[0]
20004	**0**	T[1]
20008	**0**	T[2]
20012	**0**	T[3]
20016	**0**	T[4]
20020	**0**	a
20024	**0**	b
20028	**0**	
	...	

L'ajout de l'affichage permet de voir quand le programme plante, et ce qui se passe avant.

En analysant l'affichage, on doit pouvoir retrouver l'erreur de programmation.

Qu'affichent ces programmes ?

A:

```
int T[5];

for(int i = 0; i < 5; i++)
  T[i] = 5 * i;
for(int i = 4; i >= 0; i--)
  cout << T[i] << endl;
```

B:

```
bool D[4];

D[0] = true;
D[1] = false;
D[2] = false;
D[3] = true;

for(int i = 0; i < 5; i++)
  if (D[i])
    cout << "*";
  else
    cout << "-";
cout << endl;
```

C:

```
int T[4];

T[0] = 1;
T[1] = 3;
T[2] = 2;
T[3] = 2;

cout << T[ T[0] ] << endl;

int a = 0;
for(int i = 0; i < 100; i++)
  a = T[a];
cout << a << endl;
```

Conseils pour écrire du code

Problème à résoudre

Soit un tableau `T`:

```
int T[NB];
```

Écrire le code qui décale les éléments de `T`, tel que:

- l'élément à l'indice `i` se retrouve à l'indice `i-1`,
- le premier élément du tableau se retrouve à la dernière place.

Par exemple:

Si le tableau contenait les valeurs: 0 1 2 3 4 (avec `NB` = 5),

il contiendra 1 2 3 4 0 après le décalage.

1. Faire un schéma;

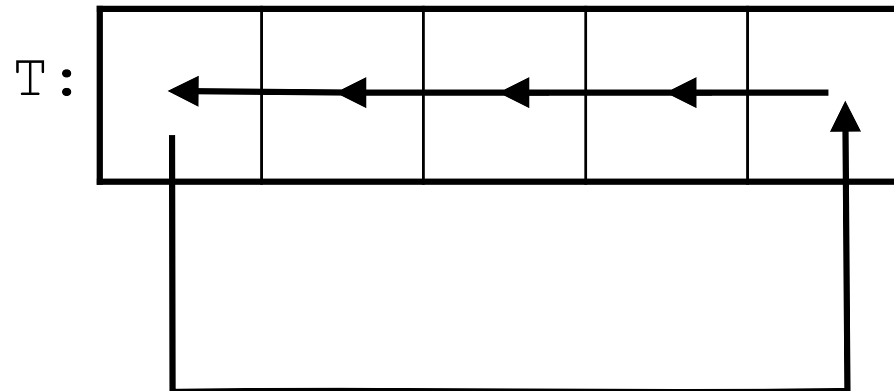
2. Décomposer le problème en problèmes plus simples;

3. Réfléchir sur un exemple, sans penser aux boucles ou aux conditions;

4. Généraliser, en écrivant cette fois les boucles et les conditions;

5. Vérifier le programme sur un exemple.

Schéma



Soit un tableau `T`:

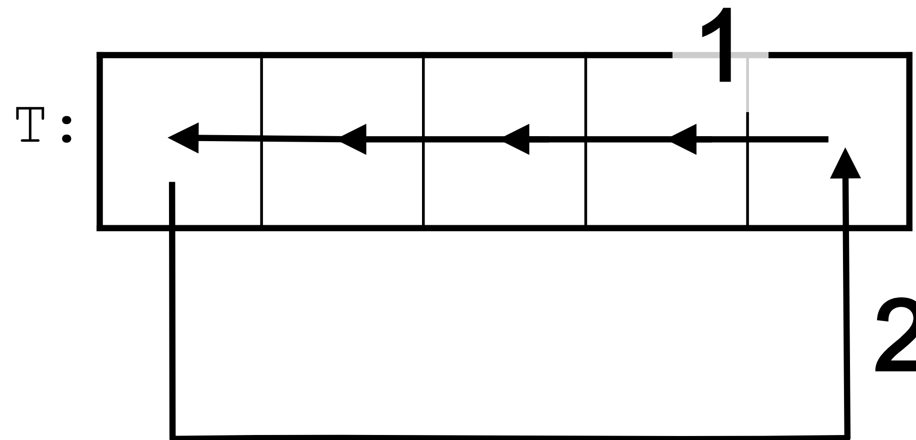
```
int T[NB];
```

Écrire le code qui décale les éléments de `T`, tel que:

- l'élément à l'indice `i` se retrouve à l'indice `i-1`,
- le premier élément du tableau se retrouve à la dernière place.

Par exemple, si le tableau contenait les valeurs: 1 2 3 4 5 (avec NB = 5), il contiendra 2 3 4 5 1 après ce décalage.

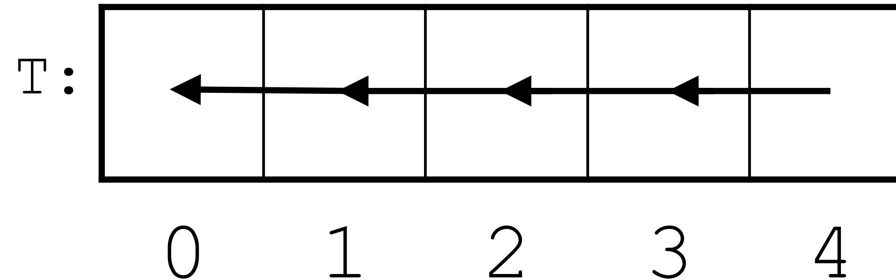
Décomposer le problème



Il faut:

1. décaler les éléments 0, 1, ... vers la gauche;
2. gérer correctement l'élément d'indice 0.

Décaler les éléments 0, 1, ... vers la gauche
Raisonner sur un exemple



Dans le cas d'un tableau à 5 éléments (`NB` = 5), il faut:

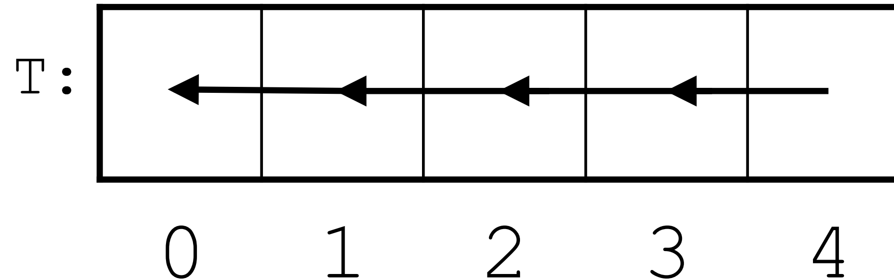
copier `T[1]` dans `T[0]`,

`T[2]` dans `T[1]`...

jusque `T[4]` dans `T[3]`:

```
T[0] = T[1];
T[1] = T[2];
...
T[3] = T[4];
```

Décaler les éléments 0, 1, ... vers la gauche
Généraliser (1)



Généralisons à un tableau à NB éléments (au lieu de 5):

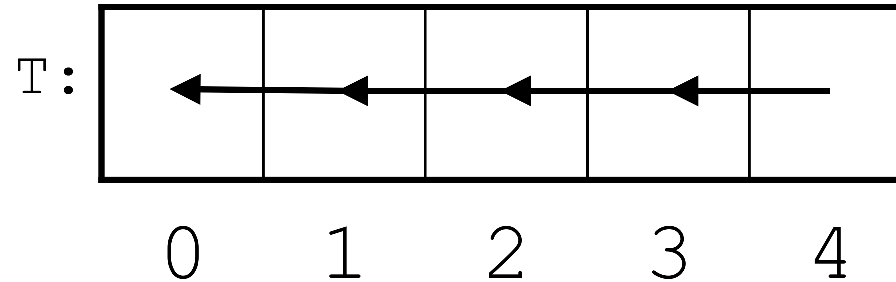
"*copier jusque* `T[4]` *dans* `T[3]`" devient

copier jusque `T[NB-1]` dans `T[NB-2]`,

ce qui s'écrit:

```
T[0] = T[1];
T[1] = T[2];
...
T[NB-2] = T[NB-1];
```

Décaler les éléments 0, 1, ... vers la gauche Généraliser (2)



Remplaçons:

```
T[0] = T[1];
T[1] = T[2];
...
T[NB-2] = T[NB-1];
```

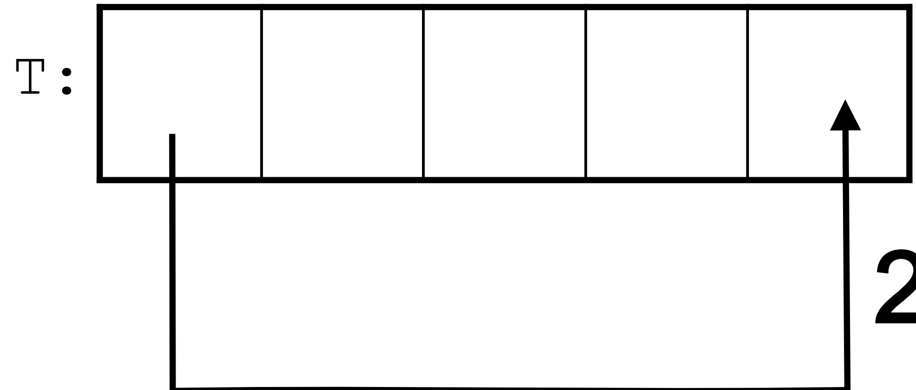
par une boucle `for`:

```
for(int i = 0; i < NB-1; i++)
  T[i] = T[i+1];
```

Une autre possibilité est:

```
for(int i = 1; i < NB; i++)
  T[i-1] = T[i];
```

Gérer correctement l'élément d'indice 0



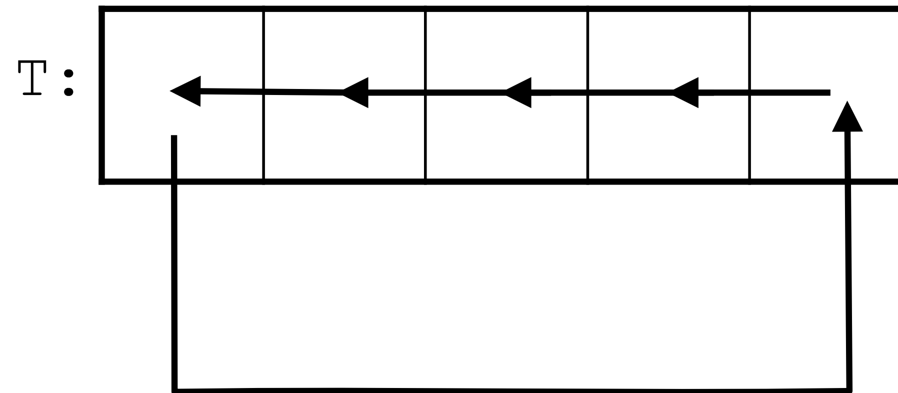
Une première solution est de faire (pour un tableau à 5 éléments):

```
T[4] = T[0];
```

ce qui se généralise pour un tableau à `NB` éléments en:

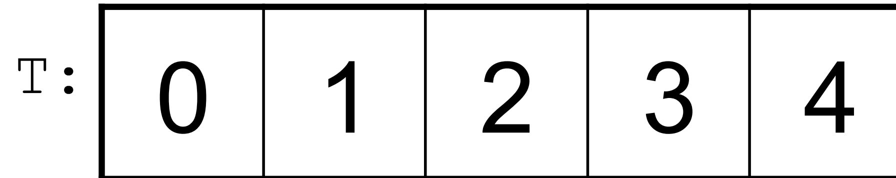
```
T[NB-1] = T[0];
```

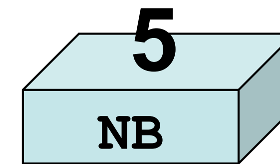
Première solution




```
T[NB-1] = T[0];

for(int i = 0; i < NB-1; i++)
   T[i] = T[i+1];
```

Vérifier le code sur un exemple

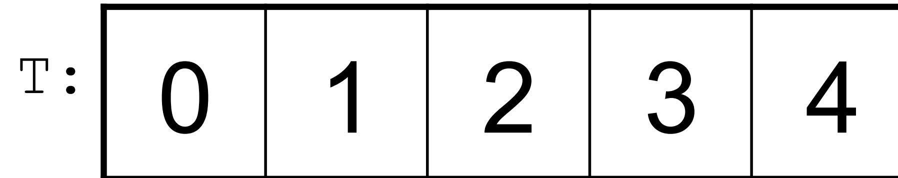


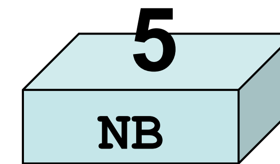



```
T[NB-1] = T[0];

for(int i = 0; i < NB-1; i++)
   T[i] = T[i+1];
```

Vérifier le code sur un exemple

T:






```
T[NB-1] = T[0];

for(int i = 0; i < NB-1; i++)
   T[i] = T[i+1];
```

Vérifier le code sur un exemple

T:

0	1	2	3	0

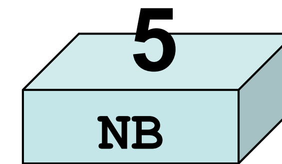
5
NB

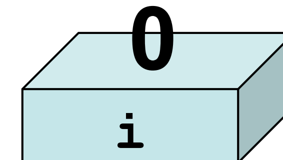
```
T[NB-1] = T[0];

for(int i = 0; i < NB-1; i++)
   T[i] = T[i+1];
```

Vérifier le code sur un exemple

T:

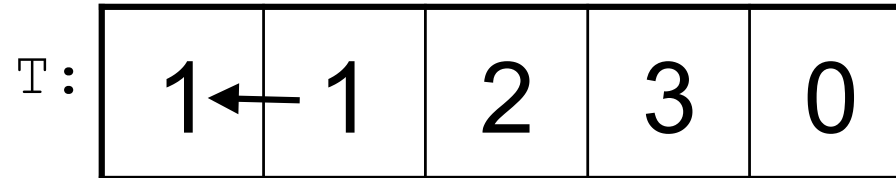
0	1	2	3	0

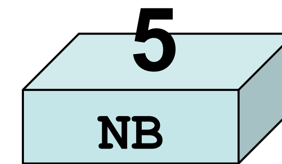




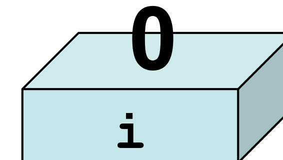

```
T[NB-1] = T[0];

for(int i = 0; i < NB-1; i++)
    T[i] = T[i+1];
```

Vérifier le code sur un exemple

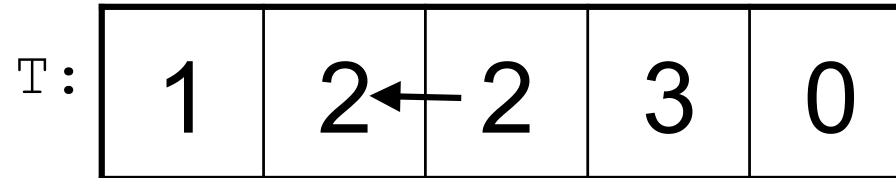


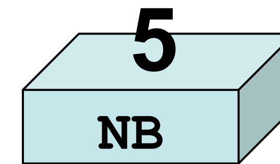


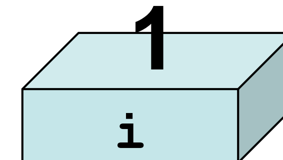

```
T[NB-1] = T[0];

for(int i = 0; i < NB-1; i++)
    T[i] = T[i+1];
```



Vérifier le code sur un exemple

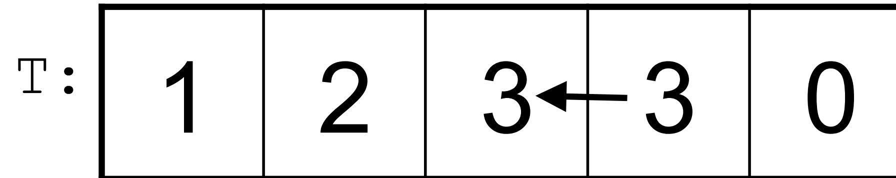


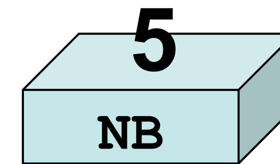


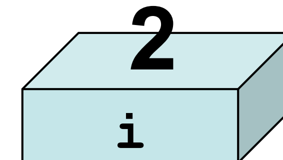



```
T[NB-1] = T[0];

for(int i = 0; i < NB-1; i++)
    T[i] = T[i+1];
```

Vérifier le code sur un exemple

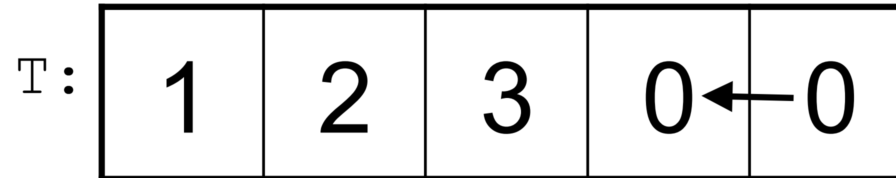


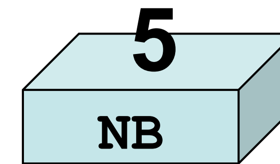




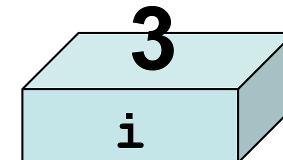

```
T[NB-1] = T[0];

for(int i = 0; i < NB-1; i++)
    T[i] = T[i+1];
```

Vérifier le code sur un exemple






```
T[NB-1] = T[0];

for(int i = 0; i < NB-1; i++)
    T[i] = T[i+1];
```



!

`T:`

1	2	3	0	0

alors qu'on voulait:

1	2	3	4	0

Que s'est-il passé ?

T:

1	2	3	0	0

L'instruction:

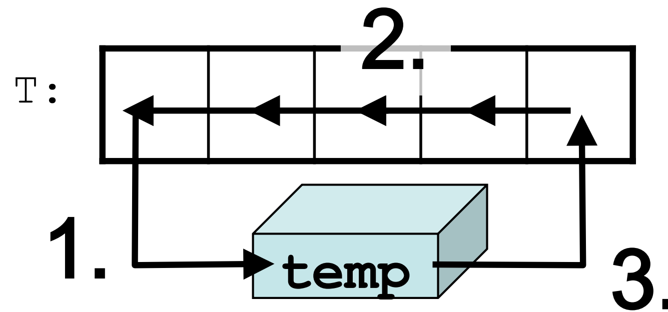
```
T[NB-1] = T[0];
```

a effacé le 4 avant que la boucle puisse le copier à l'indice 3.

Comment corriger le code ?

```
T[NB-1] = T[0];

for(int i = 0; i < NB-1; i++)
   T[i] = T[i+1];
```

Solution



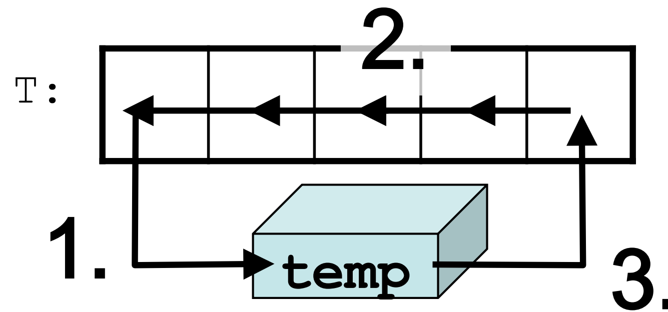
Il faut:

1. sauvegarder la valeur de `T[0]` dans une variable intermédiaire;

2. on peut maintenant décaler les éléments avec la boucle `for`;

3. recopier la valeur sauvegardée de `T[0]` dans `T[NB-1]`.

Solution



Il faut:

1. sauvegarder la valeur de `T[0]` dans une variable intermédiaire:

```
temp = T[0];
```

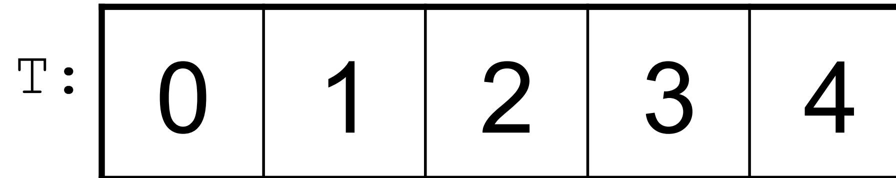
2. on peut maintenant décaler les éléments avec la boucle `for`:

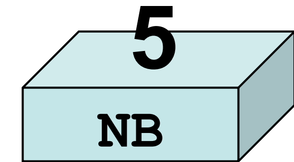
```
for(int i = 0; i < NB-1; i++)
  T[i] = T[i+1];
```

3. recopier la valeur sauvegardée (dans `temp`) de `T[0]` dans `T[NB-1]`:

```
T[NB-1] = temp;
```

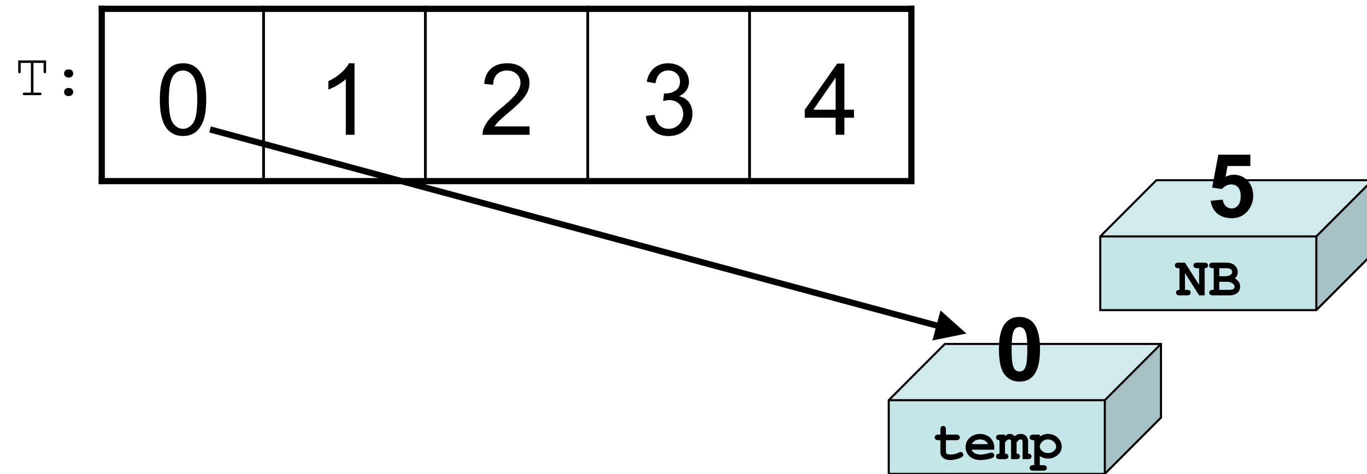
Vérifier le code sur un exemple





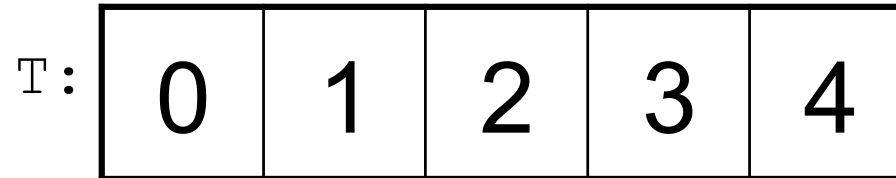

```
int temp = T[0];
for(int i = 0; i < NB-1; i++)
    T[i] = T[i+1];
T[NB-1] = temp;
```

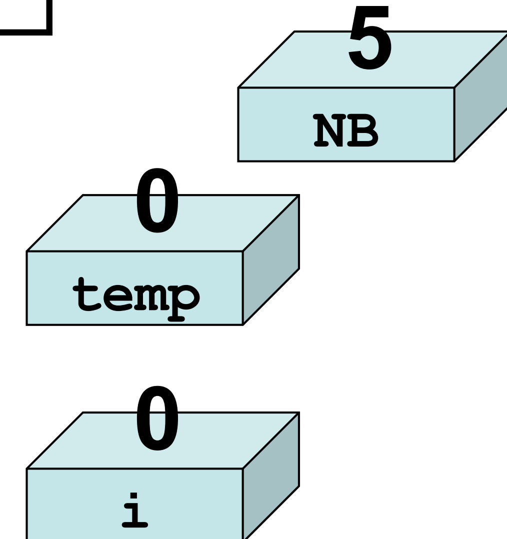
Vérifier le code sur un exemple




```
int temp = T[0];
for(int i = 0; i < NB-1; i++)
    T[i] = T[i+1];
T[NB-1] = temp;
```

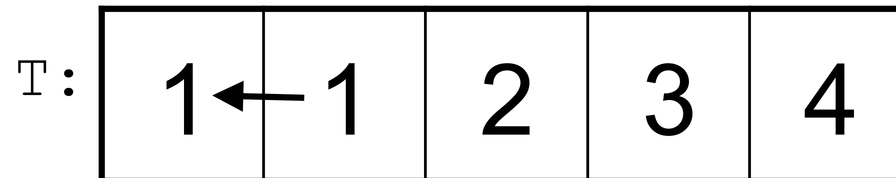
Vérifier le code sur un exemple

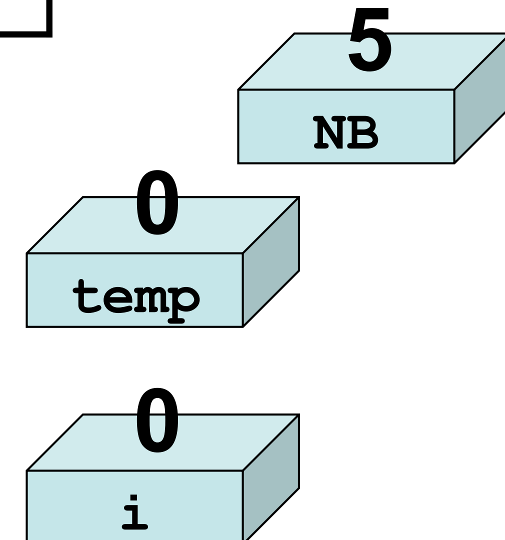





```
int temp = T[0];
for(int i = 0; i < NB-1; i++)
→ T[i] = T[i+1];
T[NB-1] = temp;
```

Vérifier le code sur un exemple



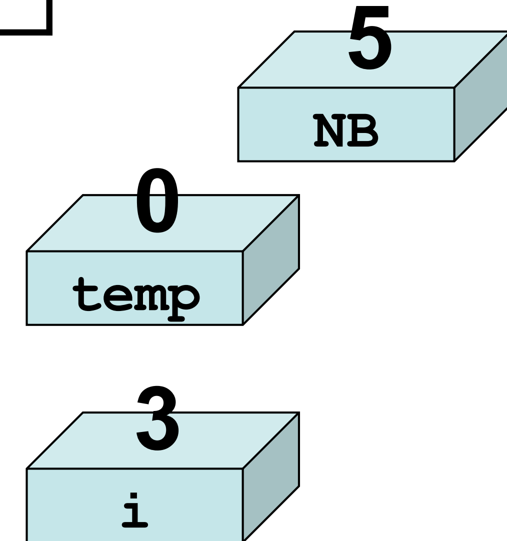



```
int temp = T[0];
for(int i = 0; i < NB-1; i++)
    T[i] = T[i+1];
T[NB-1] = temp;
```

Vérifier le code sur un exemple

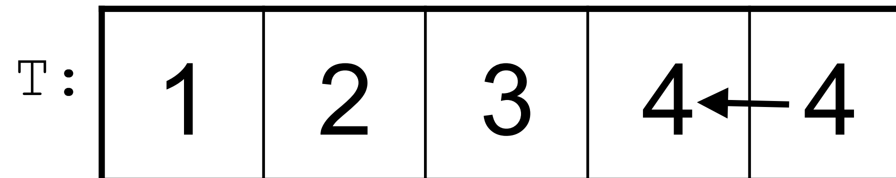
T:

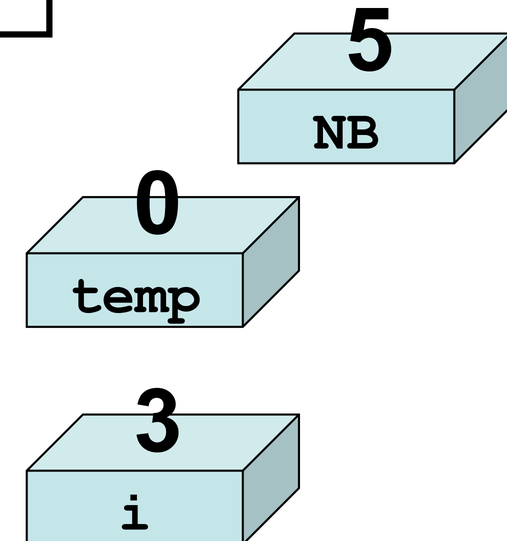
1	2	3	3	4




```
int temp = T[0];
for(int i = 0; i < NB-1; i++)
    T[i] = T[i+1];
T[NB-1] = temp;
```

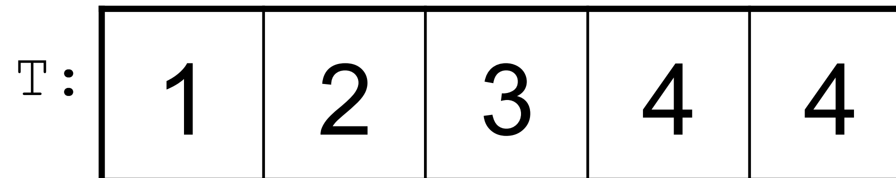
Vérifier le code sur un exemple

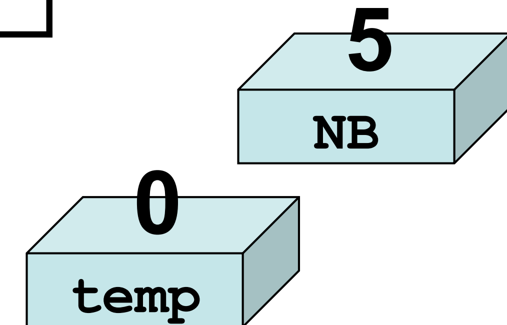





```
int temp = T[0];
for(int i = 0; i < NB-1; i++)
    T[i] = T[i+1];
T[NB-1] = temp;
```

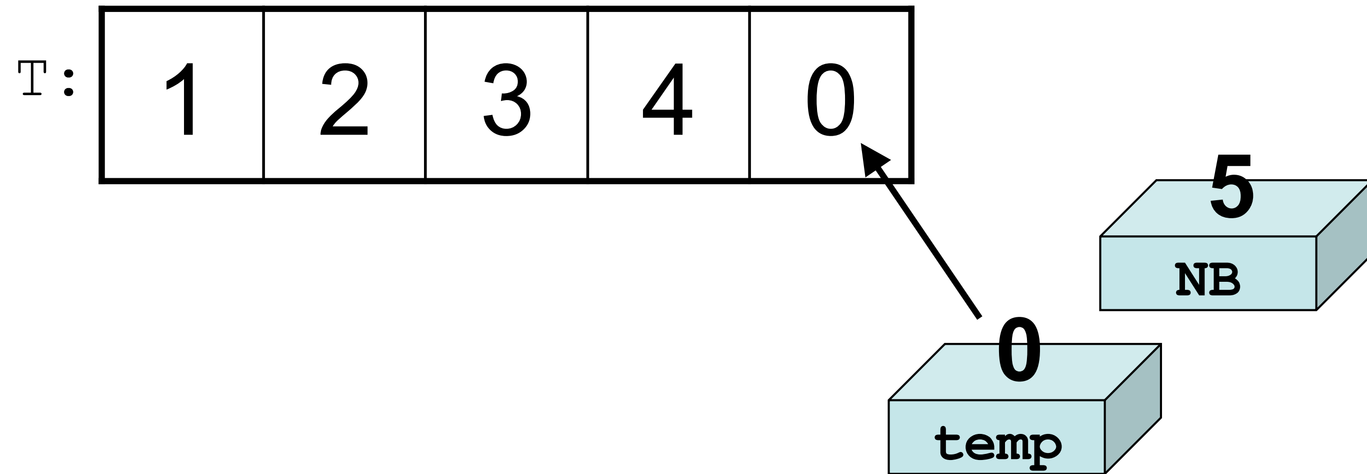
Vérifier le code sur un exemple






```
int temp = T[0];
for(int i = 0; i < NB-1; i++)
    T[i] = T[i+1];
T[NB-1] = temp;
```

Vérifier le code sur un exemple




```
int temp = T[0];
for(int i = 0; i < NB-1; i++)
    T[i] = T[i+1];
T[NB-1] = temp;
```

Exercices

1. Décalage dans l'autre sens:
   0 1 2 3 4 devient 4 0 1 2 3.

2. Écrire le code qui inverse les éléments du tableau:
   0 1 2 3 4 devient 4 3 2 1 0