

Pointeur

Mot Clé / Fonction	
NULL	<p>Valeur de pointeur constante prédéfinie \Leftrightarrow Adresse 0</p> <p>Adresse 0 est non accessible \Leftrightarrow erreur à l'exécution</p> <p>Valeur renvoyée si l'allocation dynamique de mémoire n'a pu se faire</p> <p>Adresse 0 est la seule valeur entière que l'on peut comparer à des pointeurs</p>
free(pointeur)	<p>Permet de libérer les espaces mémoires alloués par un malloc, un calloc ou un realloc lorsqu'ils ne sont plus utilisés</p>



Toujours faire un test après une allocation dynamique de mémoire pour vérifier qu'elle s'est déroulée correctement

Pointeur

- Allocation dynamique de mémoire
 - Séquence type

```
#include <stdlib.h>
#include <stdio.h>

int main(){
    int *ptr = NULL;

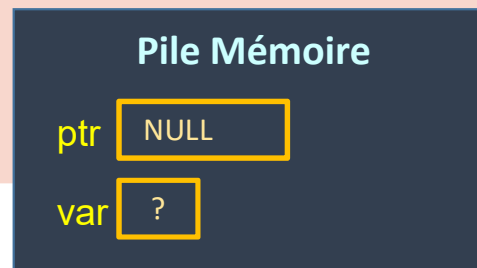
    ptr = (int*) malloc(sizeof(int)); /* Allocation */
    if (ptr == NULL){/* On vérifie si l'allocation a été un échec. */
        printf("Pb d'allocation mémoire pour ptr dans main\n");
        return EXIT_FAILURE ; //on quitte le programme
    }
    /* on continue le programme */
    *ptr=150;
    printf("La valeur entière à l'adresse %p est %d", ptr,*ptr);
    free(ptr);
    return EXIT_SUCCESS;
}
```

Pointeur

- Allocation dynamique de mémoire

Problèmes et erreurs classiques

Défaut d'initialisation de pointeur



```
#include <stdlib.h>
#include <stdio.h>

int main(){
    int *ptr = NULL;
    int var;

    var=*ptr;
    return EXIT_SUCCESS;
}
```

pas d'erreur de compilation

mais erreur à l'exécution



Pas d'affectation d'une adresse existante ni d'allocation dynamique sur le tas pour ptr

Pointeur

- Allocation dynamique de mémoire

Problèmes et erreurs classiques

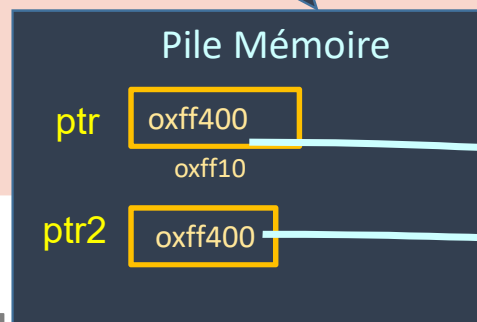
Référence multiple à une même zone mémoire

```
#include <stdlib.h>
#include <stdio.h>

int main() {
    int* ptr = (int*) malloc(sizeof(int));
    int* ptr2 = ptr;

    *ptr = 5;
    printf("%d\n", *ptr2 );
    free( ptr );
    *ptr2 = 10;
    printf("%d\n", *ptr2 );

    return EXIT_SUCCESS;
}
```



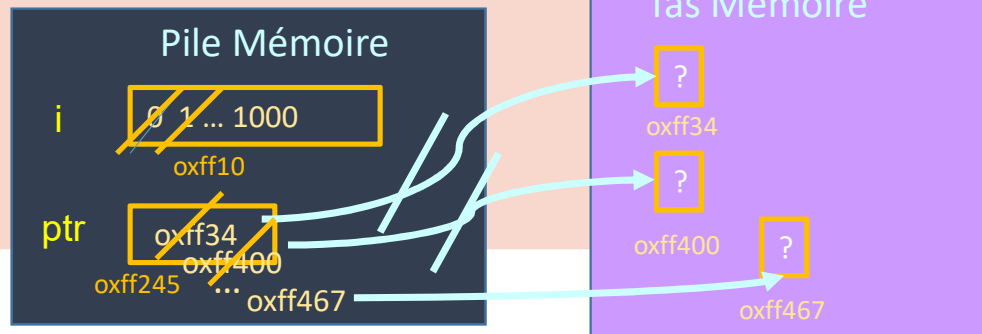
Pointeur

- Allocation dynamique de mémoire

Problèmes et erreurs classiques

Fuite Mémoire

Perte du pointeur associé à un secteur mémoire rend impossible la libération du secteur à l'aide de free



```
#include <stdlib.h>

int main(){
    int i = 0;           // un compteur
    int* ptr = NULL;     // un pointeur
    while (i < 1001) {
        ptr = (int*) malloc(sizeof(int));
        if (ptr != NULL)
        {
            /* traitement ....*/
        }
    }
    return EXIT_SUCCESS;
}
```

À la sortie de cette boucle un millier d'entiers alloués dans le tas, et non libérés car le pointeur ptr est écrasé à chaque itération (sauf la dernière)

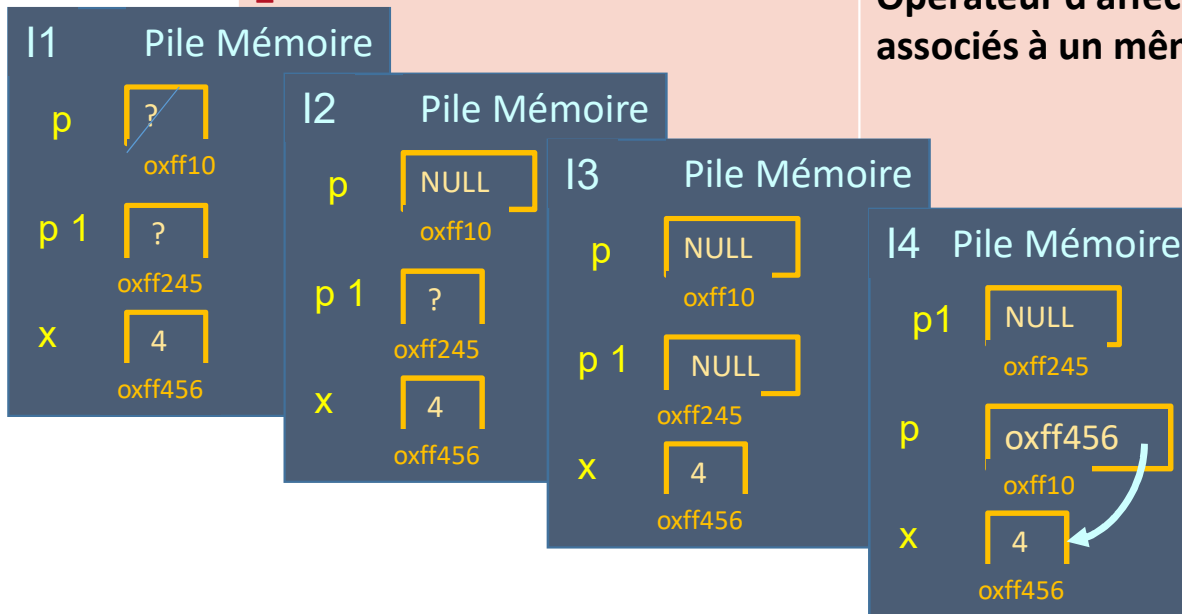
Pointeur

- Opérateurs

Opérateurs

=

Opérateur d'affectation, autorisée que pour les pointeurs associés à un même type



```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *p, *p1, x=4; /*I1*/

    p = NULL;          /*I2*/
    p1 = p;             /*I3*/
    p = &x;             /*I4*/

    return EXIT_SUCCESS;
}
```

Pointeur

- Opérateurs de comparaison

Opérateurs	
==	Opérateur de comparaison de pointeurs si les opérandes sont de type pointeur
	Opérateur supérieur et inférieur
> <	Opérateur supérieur ou égal et inférieur ou égal
>= <=	Ces Opérateurs de relation d'ordre n'ont de sens que si on a une idée de comment sont implémentés les objets en mémoire

Pointeur

- Opérateurs arithmétiques

Opérateurs les + courants	
++	Incrémentation du nombre d'octets de la valeur pointée
--	Décrémentation du nombre d'octets de la valeur pointée
+ n	Ajout d'une constante $n * \text{nb d'octets de la valeur pointée}$
-n	Soustraction d'une constante $n * \text{nb d'octets de la valeur pointée}$
+= n	
-= n	

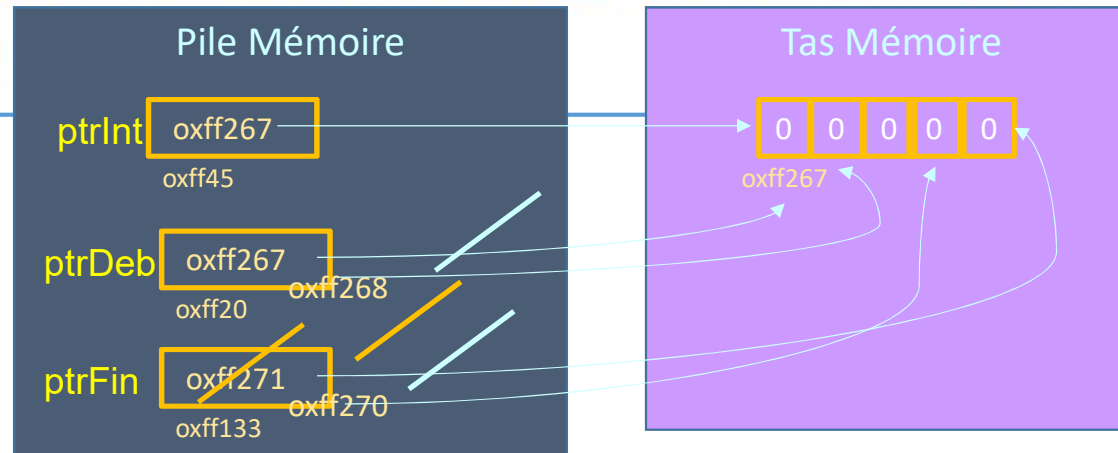
Pointeur

- Opérateurs arithmétiques et de comparaison : illustration

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *ptrInt, *ptrDeb, *ptrFin;
    const int n=5;

    ptrInt = (int*)calloc(n, sizeof(int));
    for(ptrDeb=ptrInt, ptrFin=ptrDeb+n-1; ptrDeb<=ptrFin; ptrDeb++, ptrFin--)
        printf("ptrDeb:%p est inférieur ou égal à ptrFin: %p\n", ptrDeb, ptrFin);
    printf("Sortie de Boucle car ptrDeb:%p est supérieur à ptrFin:%p", ptrDeb, ptrFin);
    free(ptrInt);
    return EXIT_SUCCESS;
}
```



Pointeur

- Opérateurs arithmétiques et de comparaison : illustration

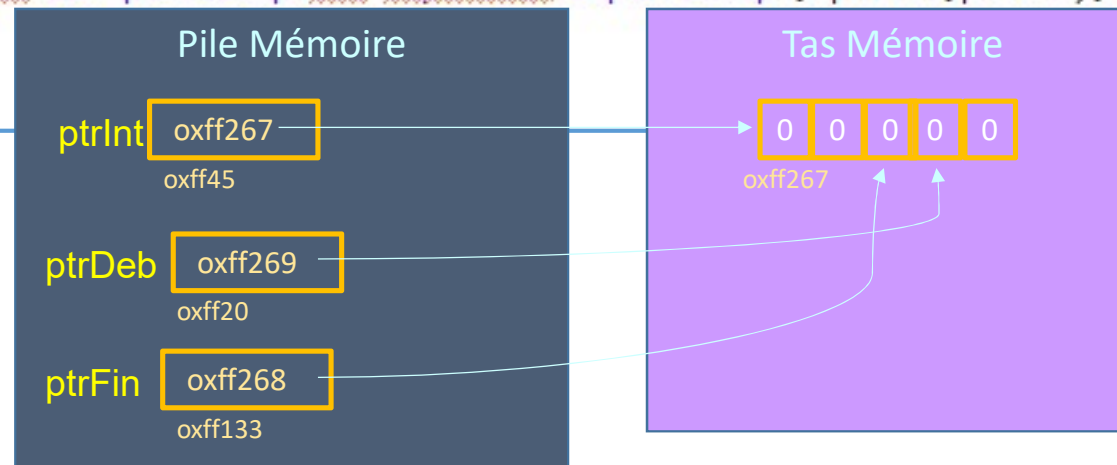
```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *ptrInt, *ptrDeb, *ptrFin;
    const int n=5;

    ptrInt = (int*)calloc(n, sizeof(int));
    for(ptrDeb=ptrInt, ptrFin=ptrDeb+n-1; ptrDeb<=ptrFin; ptrDeb++, ptrFin--)
        printf("ptrDeb:%p est inférieur ou égal à ptrFin: %p\n", ptrDeb, ptrFin);
    printf("Sortie de Boucle car ptrDeb:%p est supérieur à ptrFin:%p", ptrDeb, ptrFin);
    free(ptrInt);
    return EXIT_SUCCESS;
}
```



A la sortie de boucle



Pointeur

- Opérateurs arithmétiques : illustration

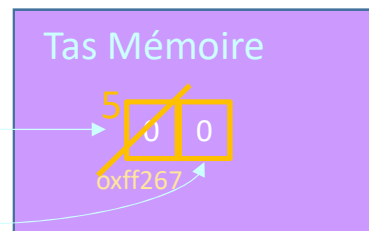
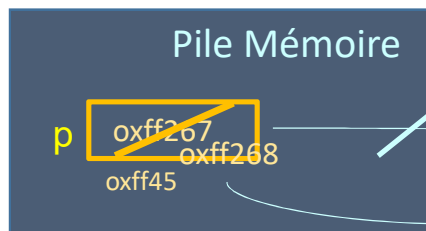


++ est prioritaire sur *

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *p;

    p = (int*)calloc(2,sizeof(int));
    *p=5;
    printf("*p: %d \n", *p);
    printf("++p: %d""", ++p);
    return EXIT_SUCCESS;
}
```



Properties

<terminated> T

*p: 5

++p: 0

Pointeur

- Opérateurs arithmétiques : illustration

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *p;

    p = (int*)calloc(2,sizeof(int));
    *p=5;
    printf("*p: %d \n", *p);
    *p=*p++;
    printf("*p++: %d \n", *p);
    return EXIT_SUCCESS;
}
```

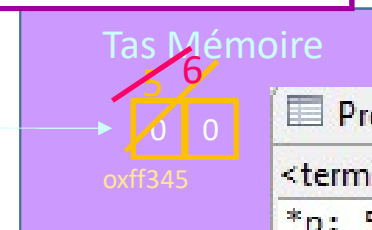


++ est prioritaire sur *

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    int *p;

    p = (int*)calloc(2,sizeof(int));
    *p=5;
    printf("*p: %d \n", *p);
    *p=(*p)++;
    printf("( *p)++: %d \n", *p);
    return EXIT_SUCCESS;
}
```



Properties
<terminated>
*p: 5
*p++: 0

Properties
<terminated>
*p: 5
(*p)++: 6

Tableaux

- Permettent de stocker des variables **de même type** de manière **contiguë en mémoire**
- Caractérisés par trois éléments
 - Type de chaque élément
 - Nombre d'éléments
 - Le nombre de dimensions

Tableaux

- Illustration de tableaux 1D, 2D, 3D

Tableau 1D \Leftrightarrow Vecteur

0	1	2
12	9	13

Tableau 2D \Leftrightarrow Matrice

	0	1	2	3	4	5
0	10	0	0	0	0	0
1	20	90	0	40	0	0
2	30	20	10	20	50	0

Tableau 3D

	Colonne					
Ligne	0	10	10	10		
1		10	10	10		
2		10	10	10		
					Profondeur	0
						1

Tableaux

- Syntaxe de déclaration/définition

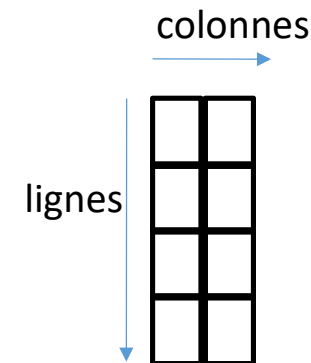
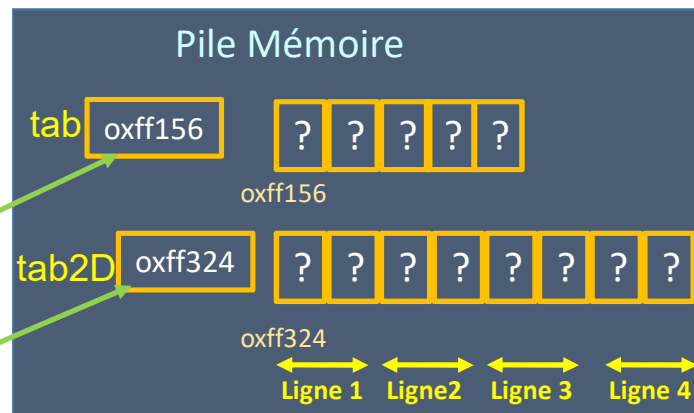
type nomTab [nbelts_dim1][nbelts_dim2] ..[nb_elts_dimN];

Exemples	
char tab[5];	Déclaration d'un tableau 1D de 5 variables de type char
int tab2D[4][2];	Déclaration d'un tableau 2D (4 lignes, 2 colonnes) de variables de type int



Allocation
automatique dans
la pile mémoire

Adresse mémoire
non réassignable



Tableaux

- Syntaxe de déclaration/définition

type nomTab [**nbelts_dim1**][**nbelts_dim2**] ..[**nb_elts_dimN**];



Le **nombre d'éléments** de chaque dimension est une constante dont la valeur est fixée avant la compilation

⇔ Si on ne connaît pas exactement cette valeur on surdimensionne le tableau

Le nombre de dimensions est également fixé avant la compilation

Valide

```
char tab[5];
```

```
#define NB 5  
char tab[NB];
```

Valide

```
int tab2D[4][2];
```

```
#define NB1 4  
#define NB2 2
```

```
int tab[NB1][NB2];
```

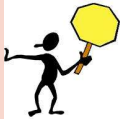
Non Valide

```
int n;  
printf("Saisie de n:");  
scanf("%d" &n);
```

```
.....  
int t[n];
```



Tableaux

#define NB 5 int tab[NB];	
0	Indice de la 1ere case du tableau
NB-1	Indice de la dernière case du tableau
	Aucune vérification des valeurs d'indice par le compilateur
tab[i] avec $0 \leq i < \text{NB}$	Accès à la valeur de la (i+1) ^{ère/ème} case du tableau
&tab[i]	Accès à l'adresse de la (i+1) ^{ère/ème} case du tableau
tab	<p>⇔ &tab[0] Accès à l'adresse de la 1ère case du tableau</p> <p>Adresse non réassignable</p> <p>⇔ non modifiable lors de l'exécution du programme</p> <p>⇔ on ne peut pas lui réaffecter l'adresse d'une autre variable</p> <p>tab = &... ⇔ écriture non autorisée</p>



Tableaux

<pre>#define NB_LIG 4 #define NB_COL 2 int tab2D[NB_LIG][NB_COL];</pre>	
0	Indice de la 1ere case du tableau
$(NB_LIG * NB_COL) - 1$	Indice de la dernière case du tableau
<code>tab2D[i][j]</code> avec $0 \leq i < NB_LIG$ $0 \leq j < NB_COL$	Accès à la valeur de la case de ligne i et de colonne j du tableau
<code>&tab2D[i][j]</code>	Accès à l'adresse de la case de ligne i et de colonne j du tableau
<code>tab2D</code>	\Leftrightarrow <code>&tab[0][0]</code> Accès à l'adresse de la 1ère case du tableau 2D \Leftrightarrow Adresse non réassignable <code>tab2D = &...</code> \Leftrightarrow écriture non autorisée



Tableaux

Définition + initialisation simultanée tableau 1D

#define NB 5

int tab[NB]={10,20,30,40,50};

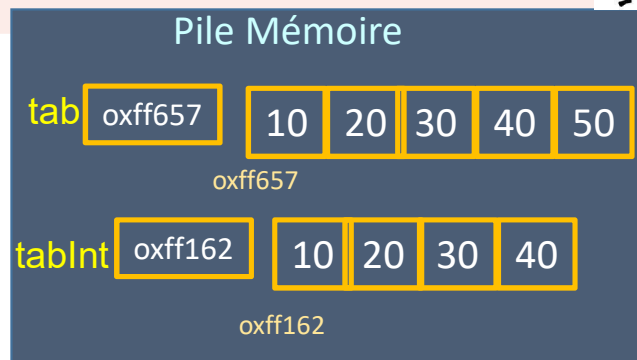
On connaît au moment de l'écriture du programme les valeurs initiales
Elles sont affectées dans l'ordre à chaque case à partir de la première

int tabInt[] = {10, 20, 30, 40} ;

Le compilateur compte le nombre de valeurs dans la liste et réserve
autant de cases en mémoire



seul cas où la taille peut être omise



Tableaux

Définition + initialisation simultanée tableau2D

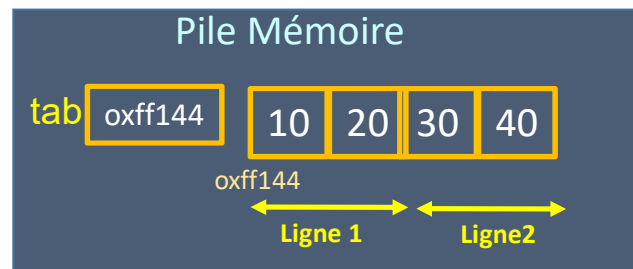
```
#define NB_LIG 2  
#define NB_COL 2
```

```
int tab[NB_LIG][NB_COL]={10,20,30,40};
```

```
int tab[NB_LIG][NB_COL]={ {10,20},{30,40}};
```

On connaît au moment de l'écriture du programme les valeurs initiales

Elles sont affectées dans l'ordre à chaque case à partir de la première case et ligne par ligne



10	20
30	40

Tableaux

Définition + initialisation simultanée tableau2D

```
#define NB_LIG 2
#define NB_COL 2
```

```
int tab[NB_LIG][NB_COL]={10,20,30,40};
int tab[NB_LIG][NB_COL]={10,20,30,40};
```

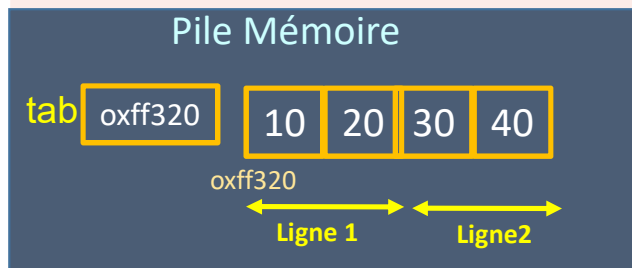
On connaît au moment de l'écriture du programme les valeurs initiales
Elles sont affectées dans l'ordre à chaque case à partir de la première ligne par ligne

```
int tab[NB_LIG][NB_COL]={10, 20, 30, 40} ;
```

Le compilateur compte le nombre de valeurs dans la liste et réserve autant de cases



Seul cas où le nombre d'éléments de la première dimension peut être omise, sa valeur est évaluée en fonction du nombre total d'éléments et du nombre de colonnes



10	20
30	40

Tableaux

Initialisation au moment de l'exécution du programme

```
#define NB 5
```

```
....
```

```
int tab[NB];
```

```
....
```

```
for(i = 0 ; i < NB ; i++){
```

```
    printf("Entrer la valeur n° %d : ", i+1) ;
```

```
    scanf("%d", &tab[i]) ;
```

```
}
```

Une boucle permet de balayer
chaque case mémoire du tableau

Indice de boucle i varie de **0** à **NB-1**

La valeur lue au clavier est mise dans
la case d'adresse **&tab[i]**

Tableaux

Initialisation au moment de l'exécution du programme

```
#define NB_LIG 2
#define NB_COL 2

....

int tab2D[NB_LIG] [NB_COL], i, j;

....

for(i = 0 ; i < NB_LIG ; i++){
    for(j=0 ; j < NB_COL ; j++){
        printf("Entrer la valeur n° %d, %d : ", i, j) ;
        scanf("%d", &tab2D[i] [j]) ;
    }
}
```

Deux boucles imbriquées:

- Une qui gère le balayage de lignes, avec l'indice **i**
- L'autre qui, avec l'indice **j**, pour chaque ligne permet de balayer chaque case des colonnes

La valeur lue au clavier est mise dans la case d'adresse **&tab2D[i][j]**

Tableaux

Initialisation au moment de l'exécution du programme

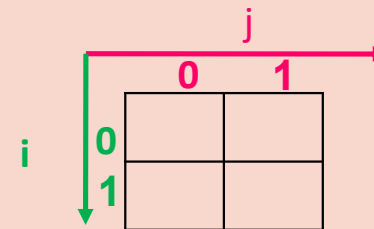
```
#define NB_LIG 2
#define NB_COL 2

....
int tab2D[NB_LIG][NB_COL], i, j;
....
for(i = 0 ; i < NB_LIG ; i++){
    for(j=0 ; j < NB_COL ; j++){
        printf("Entrer la valeur n° %d, %d : ", i, j) ;
        scanf("%d", &tab2D[i][j]) ;
    }
}
```

Deux boucles imbriquées:

- Une qui gère le balayage de lignes, avec l'indice **i**
- L'autre qui, avec l'indice **j**, pour chaque ligne permet de balayer chaque case des colonnes

La valeur lue au clavier est mise dans la case d'adresse `&tab2D[i][j]`



Tableaux

```
#define NB 4
```

```
....
```

```
int tab1[NB]={10,20,30,40};
```

```
int tab2[NB]={10,20,30,40};
```

```
....
```

```
if (tab1==tab2)
```



Pile Mémoire



Comparaison de tableaux

N'a aucun sens

Revient à comparer `&tab1[0]` et `&tab2[0]`

Or ces adresses n'ont aucune raison d'être identiques

Tableaux

```
#define NB 4
```

```
...
```

```
int tab1[NB]={10,20,30,40};
```

```
int tab2[NB]={10,20,30,40};
```

```
int i;
```

```
....
```

```
for(i = 0 ; i < NB && tab1[i]==tab2[i] ; i++);
```

```
if(i==NB)
```

```
    printf("tab1 et tab2 sont identiques\n");
```

```
else
```

```
    printf("tab1 et tab2 sont différents\n");
```

Comparaison de tableaux

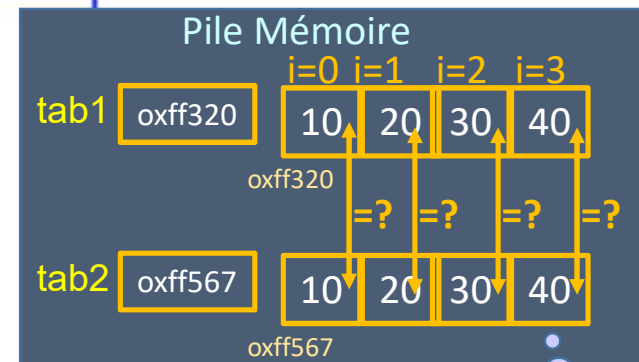
Comparaison de chaque valeur pour le même indice de case dans les deux tableaux et on s'arrête dès que deux valeurs sont différentes

Pile Mémoire



Tableaux

```
int main () {  
    int tab1[NB]={10,20,30,40};  
    int tab2[NB]={10,20,30,40};  
    int i;  
    for(i=0;i<NB && tab1[i]==tab2[i];i++);  
    if(i==NB)  
        printf("tab1 et tab2 sont identiques\n");  
    else  
        printf("tab1 et tab2 ne sont pas identiques\n");  
    return EXIT_SUCCESS;  
}
```

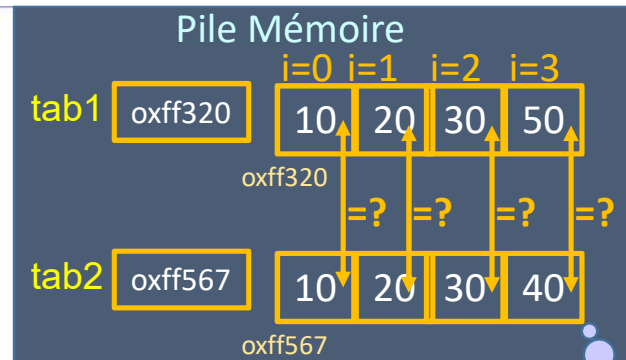


Properties Problems Tasks
<terminated> TestSwitch.exe [C/C++ A
tab1 et tab2 sont identiques

Tous les tests sont vrais
⇔ Sortie de boucle avec
i est égal à NB

Tableaux

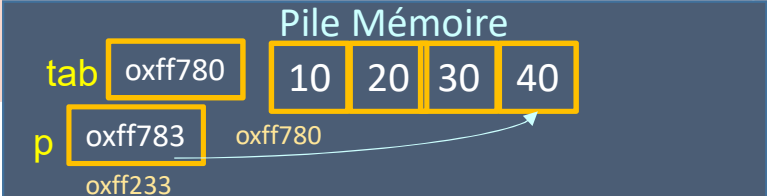

```
int main () {  
    int tab1[NB]={10,20,30,50};  
    int tab2[NB]={10,20,30,40};  
    int i;  
  
    for(i=0;i<NB && tab1[i]==tab2[i];i++);  
    if(i==NB)  
        printf("tab1 et tab2 sont identiques\n");  
    else  
        printf("tab1 et tab2 ne sont pas identiques\n");  
  
    return EXIT_SUCCESS;  
}
```



Dernier test est faux
↔ Sortie de boucle avec
i est égal à NB-1

Properties Problems Task
<terminated> TestSwitch.exe [C/C++]
tab1 et tab2 sont différents

Tableaux

Arithmétique d'adresse et tableaux	
<pre>#define NB 4 int tab[NB]={10,20,30,40}; int i=2; int *p;</pre>	
<pre>tab ⇔ &tab[0] tab + i ⇔ &tab[0] + 2 ⇔ &tab[2] &tab[2] - 1 ⇔ &tab[1]</pre>	<p>Adresse de la 1^{ère} case du tableau</p> <p>Adresse de la 3^{ème} case du tableau</p> <p>Adresse de la 2^{ème} case du tableau</p>
<pre>p = &tab[3]; p = &tab[3] - 3;</pre>	<p>p pointe sur la 4^{ème} case du tableau tab</p> <p>p pointe sur la 1^{ère} case du tableau tab</p>
 <p>Pile Mémoire</p> <p>tab 0xff780 10 20 30 40</p> <p>p 0xff783 0xff780</p> <p>0xff233</p>	 <p>Pile Mémoire</p> <p>tab 0xff780 10 20 30 40</p> <p>p 0xff780 0xff780</p> <p>0xff233</p>