Introduction au C/C++

Cours 3

Rémi Watrigant
(fortement inspiré du cours de V. Boudet, P. Giorgi et M. Joab de l'Université Montpellier 2)

Université de Nîmes

2013-2014

Plan

Plan

Variables et fonctions : *une entente peu cordiale*

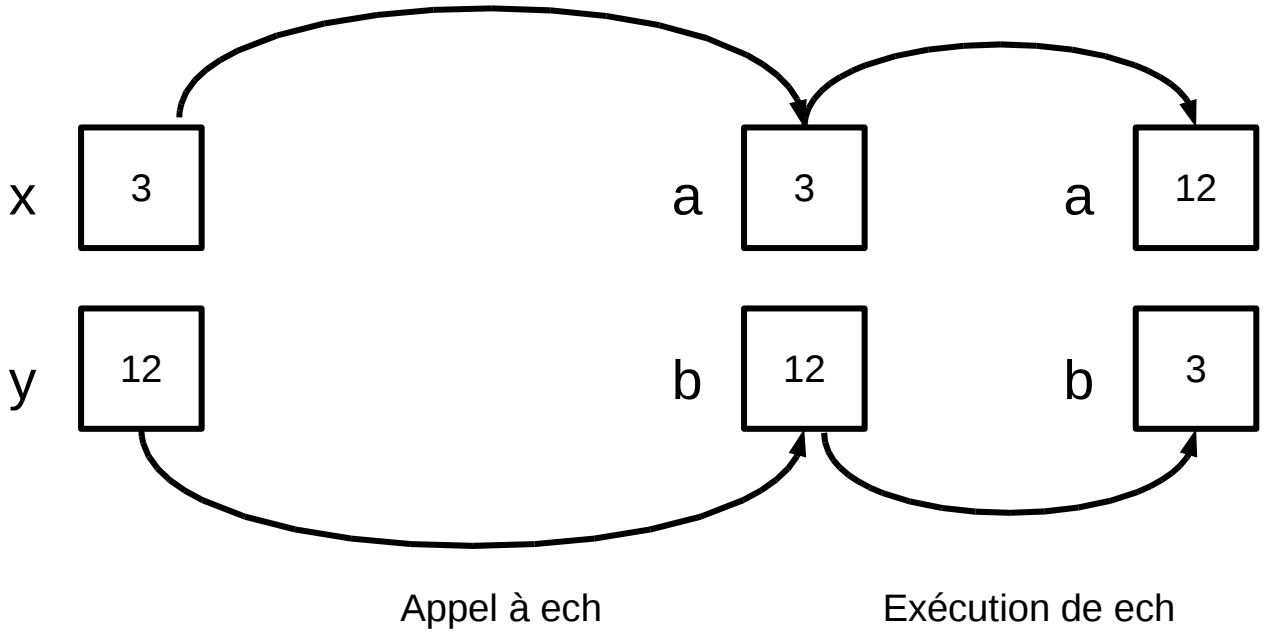
Exemple

```
void echange(int a, int b){
   int c;
   c=a;
   a=b;
   b=c;
}
```

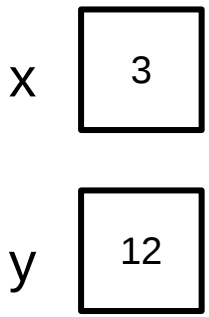
l'appel à la fonction `echange` sur des variables `x` et `y` n'effectue pas l'effet attendue (échanger les valeurs).

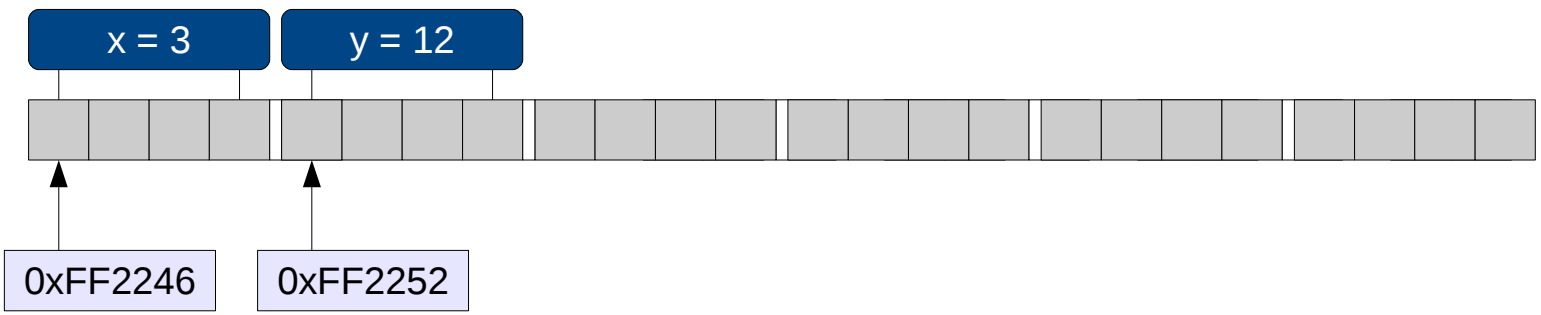
→ paramètres toujours passés par copie dans les fonctions ! ! !

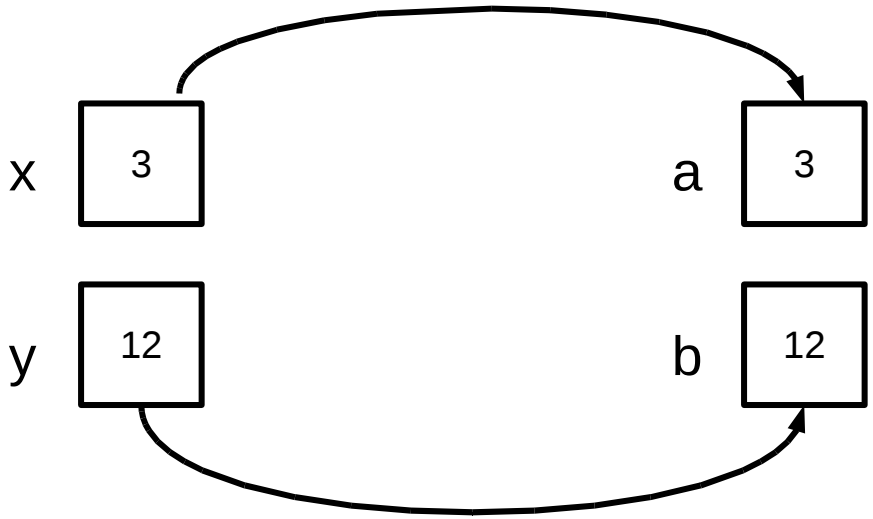
Passage des paramètres par copie



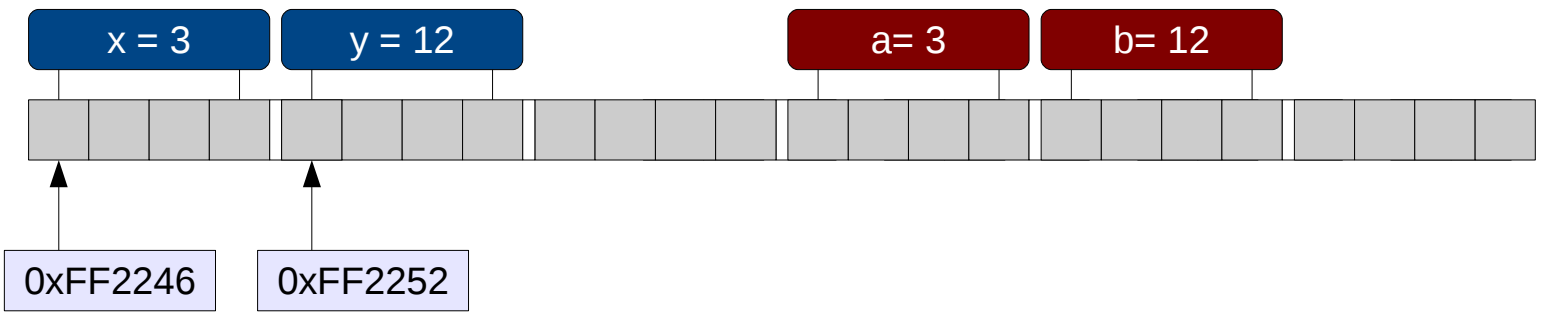

```
int x,y;
x=3;y=12;
echange(x,y);
```

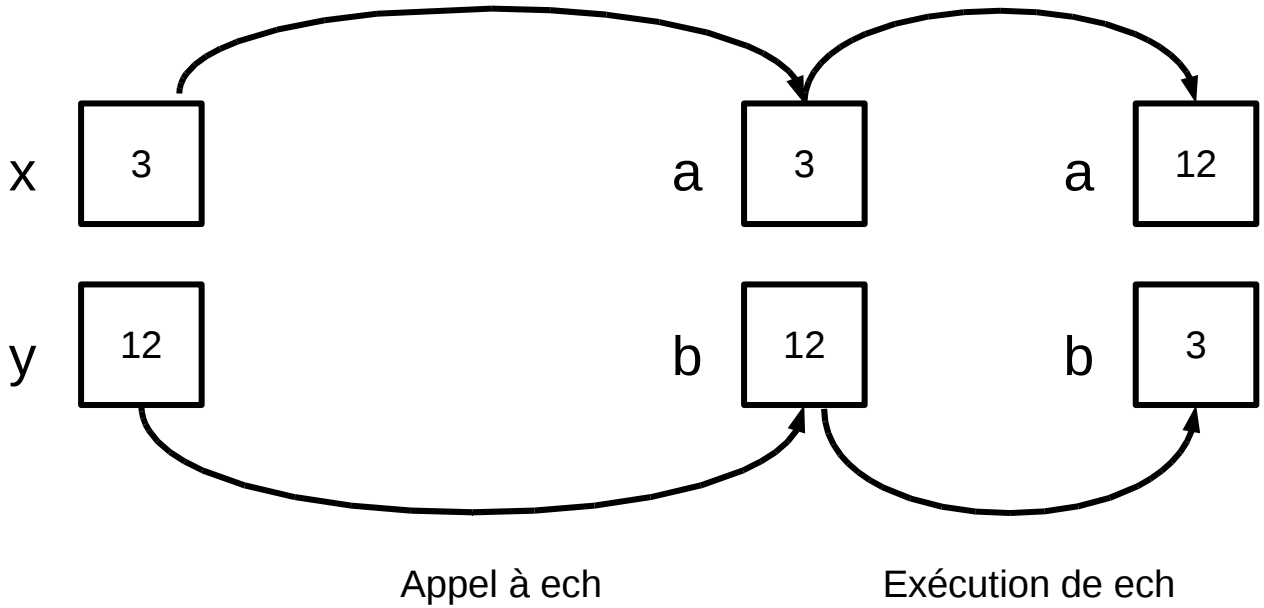

x
3
y
12

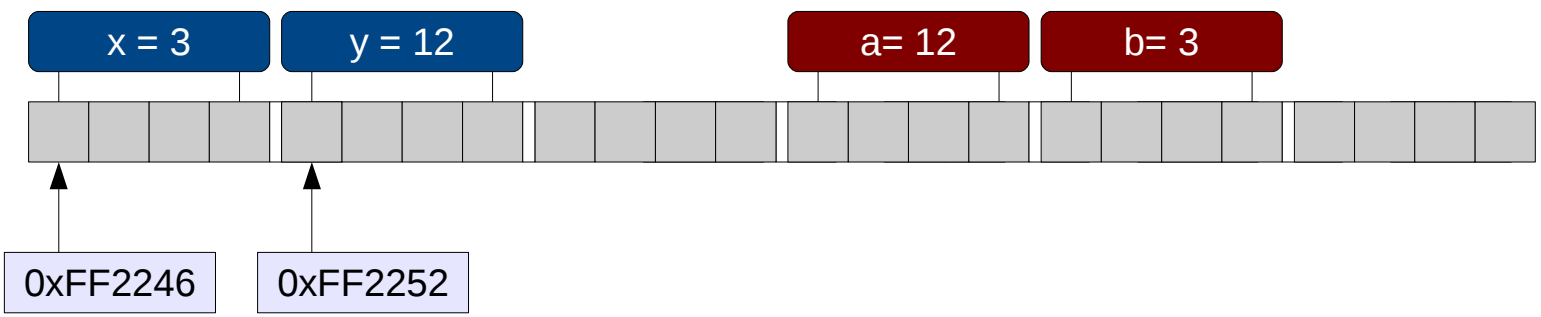

x = 3
y = 12
0xFF2246
0xFF2252

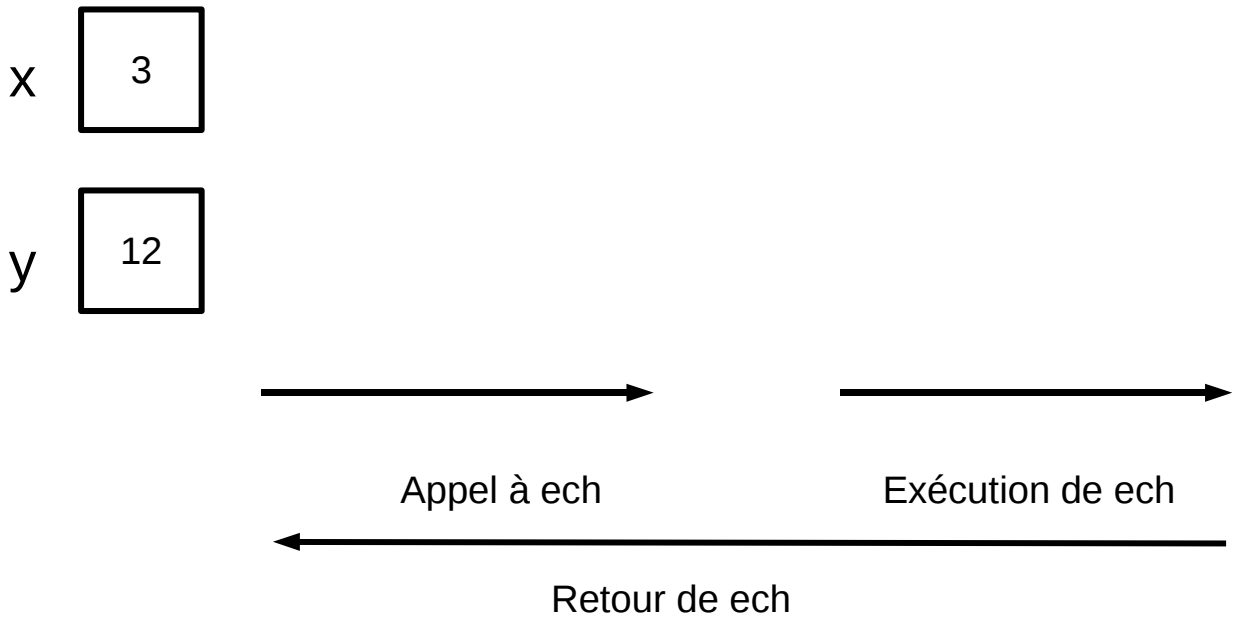


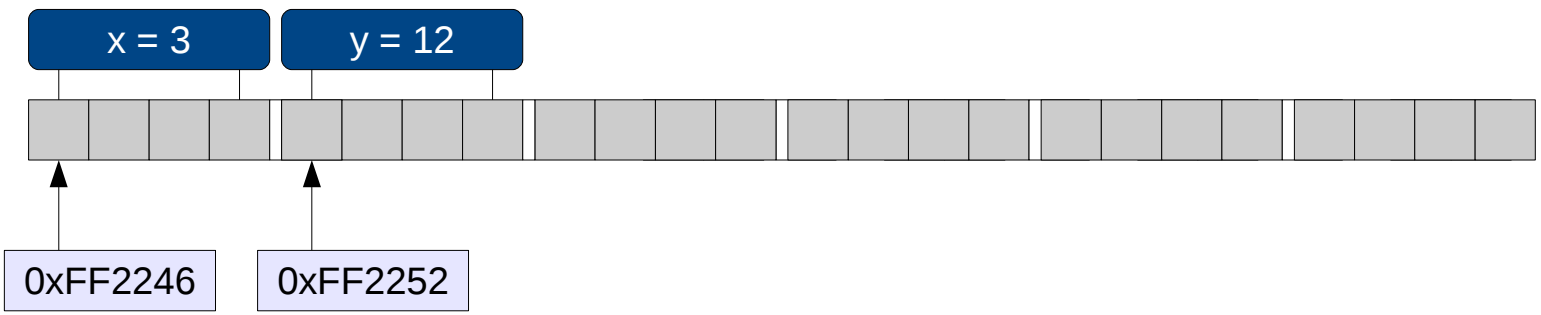
Appel à ech




x
3
y
12
a
3
b
12
a
12
b
3
Appel à ech
Exécution de ech


x = 3
y = 12
a= 12
b= 3
0xFF2246
0xFF2252


x
3
y
12
Appel à ech
Exécution de ech
Retour de ech


x = 3
y = 12
0xFF2246
0xFF2252

Limitations des variables : telles qu'on les connait

Les variables :

- ne peuvent pas être modifiées par les fonctions
- sont limitées au bloc dans lequel elles ont été définies

Remarque

Ces limitations proviennent de la manipulation des variables par leur identificateur.

Solution : manipuler les variables par leur adresse (*pointeur*)

Plan

Les pointeurs

Definition

Un pointeur est une variable qui contient l'adresse mémoire d'une donnée (une autre variable).

Une variable dite *pointeur* est définie par

- un identificateur : le nom du pointeur
- un type de donnée : pour la donnée pointée

Remarque

Les pointeurs permettent de manipuler des données par leur adresse plutôt que par leur identificateur.

Les variables de type pointeur

Definition

```
type *var;
```

- `var` est l'identificateur (le nom) du pointeur
- `type` est le type de donnée de la donnée pointée
- `var` doit contenir une adresse mémoire valide (pas une valeur)

Exemple

```
int *ptr;
```

définit la variable `ptr` comme une pointeur sur un entier.
→ `ptr` devra donc contenir l'adresse d'une donnée de type `int`.

Les variables de type pointeur : Exemple

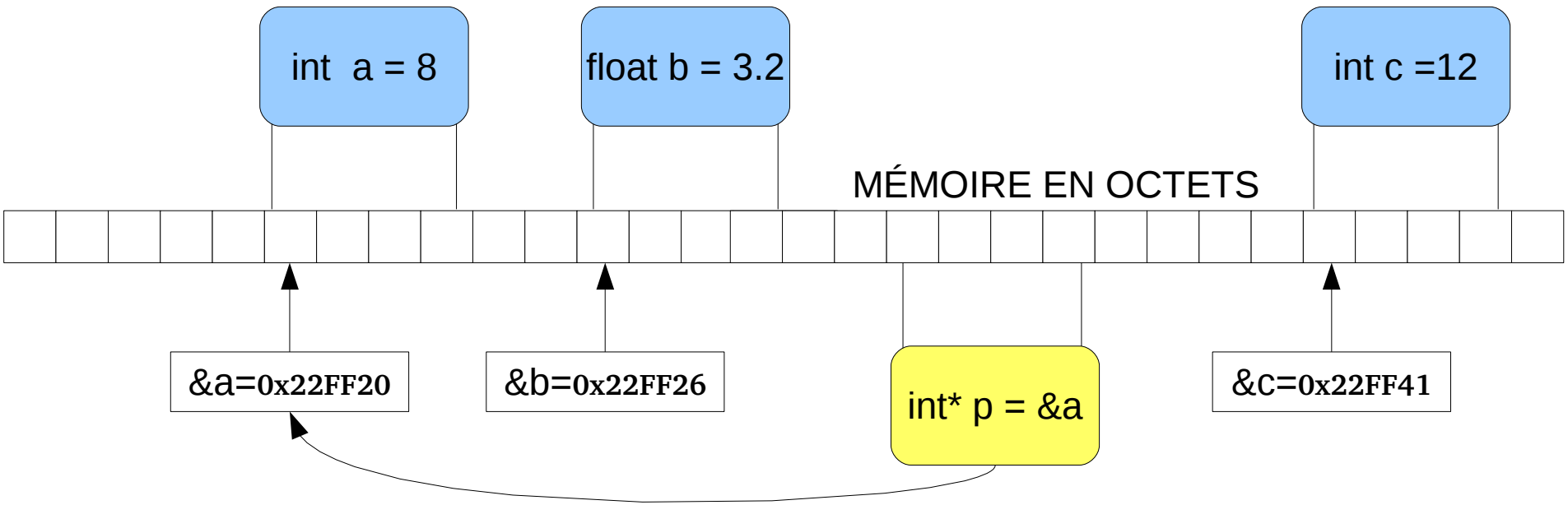
Déclaration de pointeurs :

```
1 int *Aptr;
2 double *Bptr;
3 int *Cptr;
```

Affectation de pointeurs :

```
1 int a; double b;
2
3 Aptr= &a;     // OK
4 Bptr= &b;     // OK
5 Aptr= &b;     // ERREUR
6 Bptr= &a;     // ERREUR
7
8 Aptr= Bptr;   // ERREUR
9 Aptr= Cptr;   // OK
```

Les variables de type pointeur : Vue mémoire



- `a,b,c` sont des variables normales (elles stockent des valeurs).
- `p` est une variable *dite* pointeur (elle stocke une adresse).
- `p` est de type `int*` et doit stocker l'adresse d'un entier (ici `a`).

Les variables de type pointeur : Affectation

On peut affecter une variable pointeur avec :

- l'adresse d'une variable existante *compatible*,
- avec la valeur NULL (pointeur vide de `stdlib.h`),
- avec la valeur d'une autre pointeur *compatible*,

Exemple

```
int a, *p, *q
p=&a;
q=NULL;
q=p;
```

Les variables de type pointeur : Accès aux données

Pour accéder à la zone mémoire pointée par un pointeur, il faut utiliser l'opérateur prefixe de déréférencement `*`.

Definition

`type *var;`

le déréférencement `*var` permet d'accèder à la donnée qui est stockée à l'adresse mémoire `var` (donnée pointée).

```
1 int a;
2 int *ptr;
3 ptr=&a; // affecte ptr avec l'adresse de a
4 *ptr=3; // affecte la zone mémoire d'adresse ptr
5         // avec la valeur 3 (ici a=3)
```

Manipulation des données avec les pointeurs

Lorsqu'on déréférence une variable pointeur, on peut :

- utiliser la valeur stockée dans la zone mémoire pointée
- modifier la valeur stockée dans la zone mémoire pointée

Exemple

```
int a, *p;
a=10;p=&a;
a=*p+1;
*p=13;
```

Les pointeurs : Exemple 1

```c
#include <stdio.h>
int main(){
   int A, *Aptr;
   A=150;
   Aptr=&A;
   printf("val:%8d|addr:%8X\n",A,&A);
   printf("val:%8X|addr:%8X|vap p.:%4d\n",Aptr,&Aptr,*Aptr);

   A=99;
   printf("val:%8d|addr:%8X\n",A,&A);
   printf("val:%8X|addr:%8X|vap p.:%4d\n",Aptr,&Aptr,*Aptr);

   *Aptr=45;
   printf("val:%8d|addr:%8X\n",A,&A);
   printf("val:%8X|addr:%8X|vap p.:%4d\n",Aptr,&Aptr,*Aptr);

   return 0;
}
```

Les pointeurs : Exemple 2

Attention

Les pointeurs sont dangereux et causent des erreurs dites de segmentation dans l'exécution des programmes.

```
1 #include <stdio.h>
2 int main(){
3
4    int A, *Aptr;
5    A=100;
6    *Aptr=17; // cette ligne compile correctement
7              // mais cause une erreur a l'execution
8    return 0;
9 }
```

Plan

Fonctions avec pointeurs

Definition

```
type_retour mafonction(type_param * ptr){...}
```

Dans la fonction, c'est une copie du pointeur qui est manipulée et non pas le pointeur.

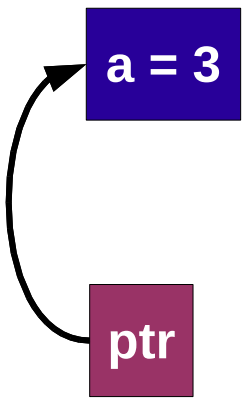
A l'intérieur de la fonction :

- on peut récupérer la donnée pointée : `*ptr`
- on peut modifier la valeur de la donnée pointée : `*ptr=...`

Fonctions avec pointeurs : Exemple 1

```
1  #include <stdio.h>
2
3  void plusUn(int *p){
4     *p= *p+1;
5  }
6
7  int main(){
8     int a;   int *ptr;
9     a=3;   ptr=&a;
10
11    plusUn(ptr);
12    printf("a= %d\n",a);
13    plusUn(&a);
14    printf("a= %d\n",a);
15
16    return 0;
17 }
```

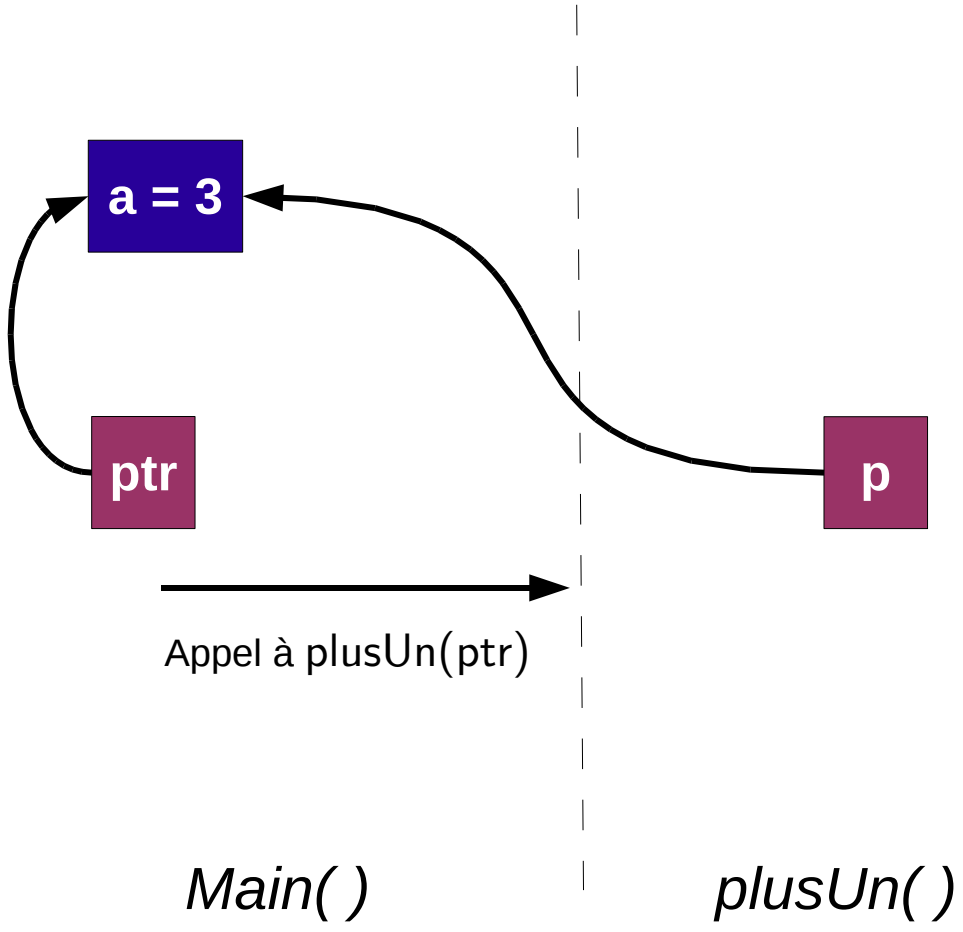
Fonctions avec pointeurs : Exemple 1



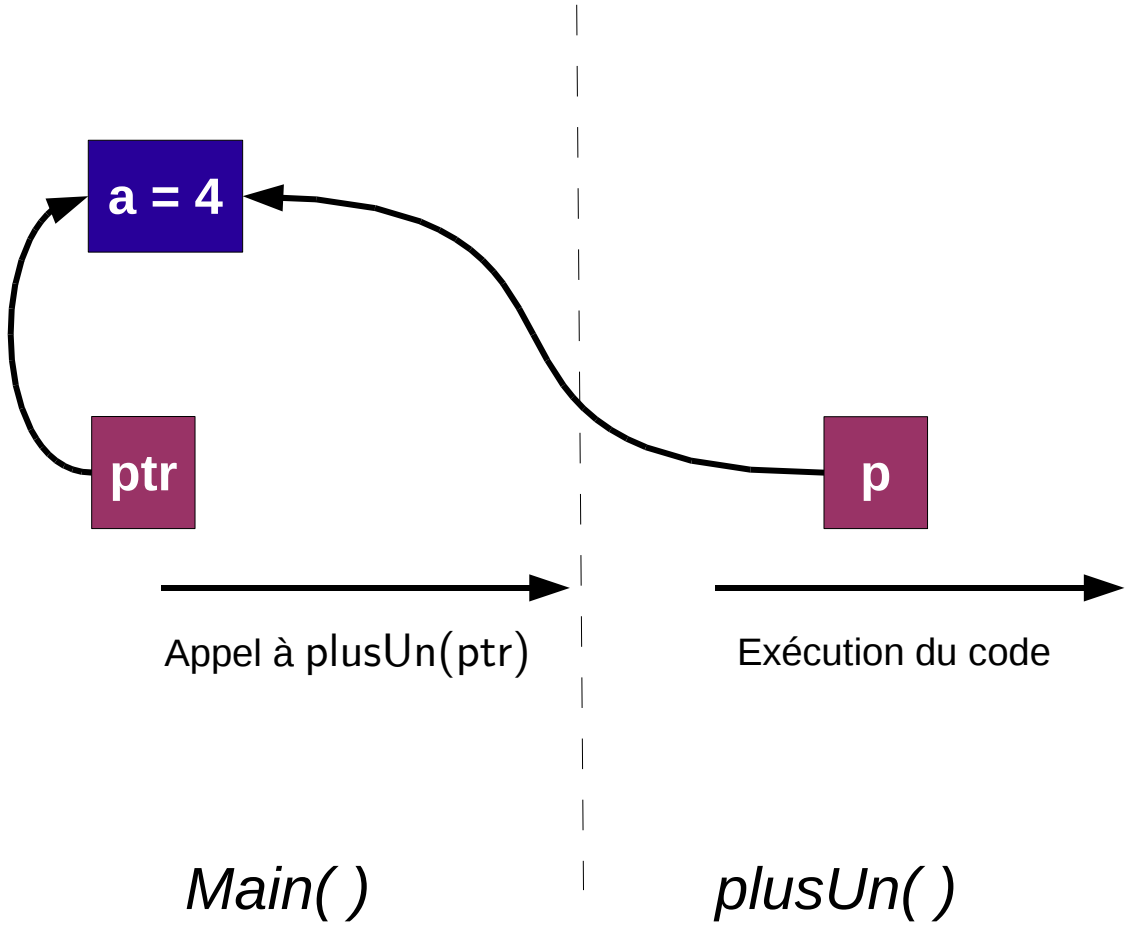
Main( )

plusUn( )

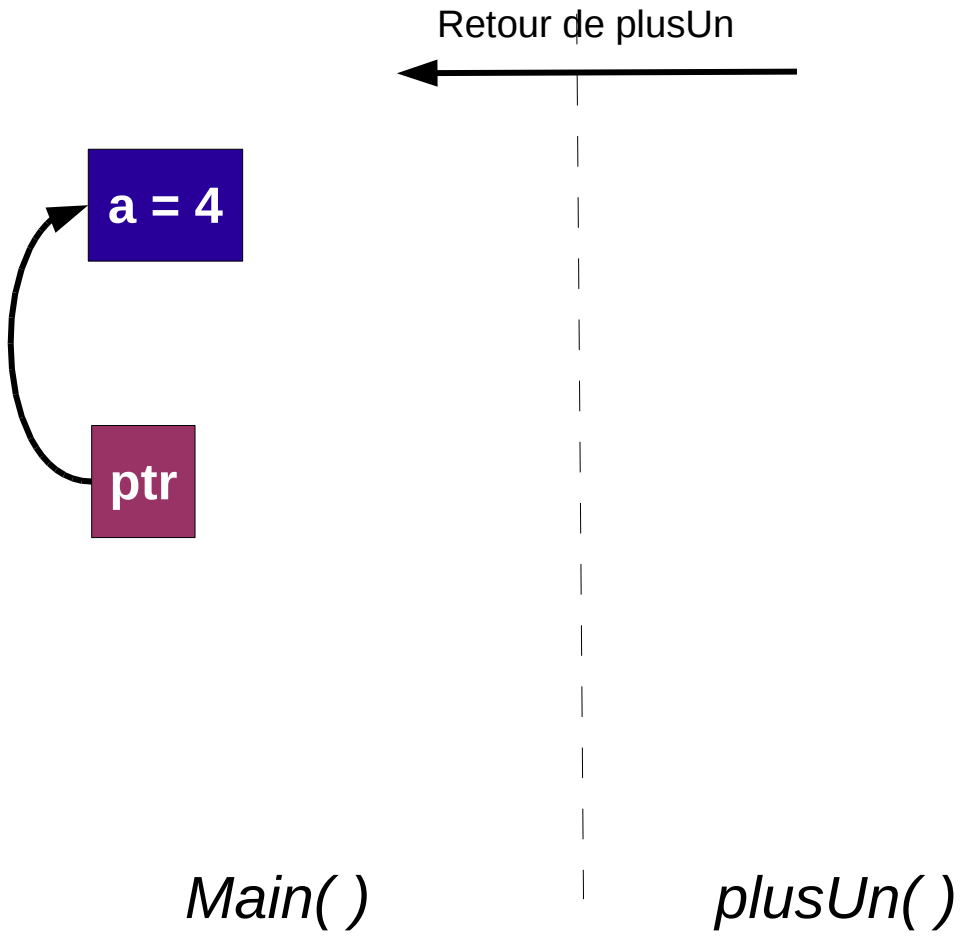
Fonctions avec pointeurs : Exemple 1



Fonctions avec pointeurs : Exemple 1



Fonctions avec pointeurs : Exemple 1



Fonctions avec pointeurs : Exemple 2

```
1  #include <stdio.h>
2
3  void ech(int *a, int *b){
4     int r;
5     r=*a;
6     *a=*b;
7     *b=r;
8  }
9  int main(){
10    int x,y;
11    x=2;y=3;
12    ech(&x,&y);
13    printf("x=%d\ny=%d\n",x,y);
14    return 0;
15 }
```

Fonction avec pointeur : paramètres résultats

En utilisant les pointeurs, on peut donc interagir dans les deux sens avec une fonction :

- donner des valeurs en entrée dans une fonction
- récupérer plusieurs valeurs en sortie d'une fonction

Exemple

```
void minmax(int a, int b, int *min, int *max){
   if (a>b)
      {*min=b; *max=a;}
   else
      {*min=a; *max=b;}
}
```

Fonction avec pointeur : Exemple 3

```
1  #include <stdio.h>
2
3  void minmax(int a, int b, int *min, int *max){
4     if (a>b)
5        { *min=b; *max=a;}
6     else
7        { *min=a; *max=b;}
8  }
9
10 int main(){
11    int a,b,min,max;
12    a=10; b=15;
13    minmax(a,b,&min,&max);
14    printf("min=%d\nmax=%d\n",min,max);
15 }
```

Fonctions avec pointeurs

L'utilisation classique des pointeurs dans une fonction ne permet pas de modifier la valeur des pointeurs.

Exemple

```
void echPtr(int *p1, int *p2){
   int *p;
   p=p1;
   p1=p2;
   p2=p;
}
```

Bien que syntaxiquement correcte, cette fonction n'échangera pas la valeur de p1 et de p2.

Passage de pointeur par copie

Le code suivant n'effectue aucun échange entre a et b car le passage des paramètres en C se fait toujours par copie !!!

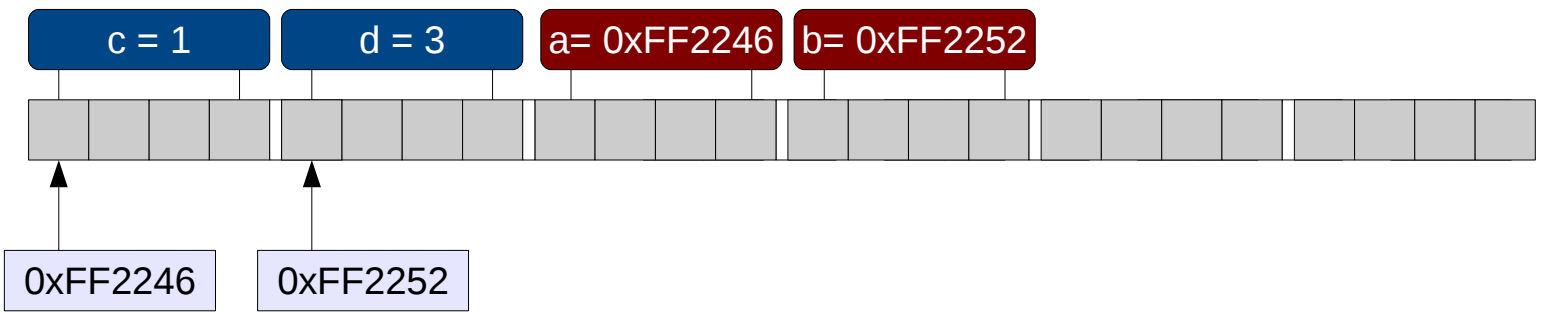
```
1  void echPtr(int *p1, int *p2){
2     int *p;
3     p=p1;   p1=p2;   p2=p;
4  }
5  int main(){
6     int *a,*b;
7     int c,d;
8     c=1;   d=3;
9     a=&c;  b=&d;
10    echPtr(a,b); // pas d'echange entre a et b
11    return 0;
12 }
```

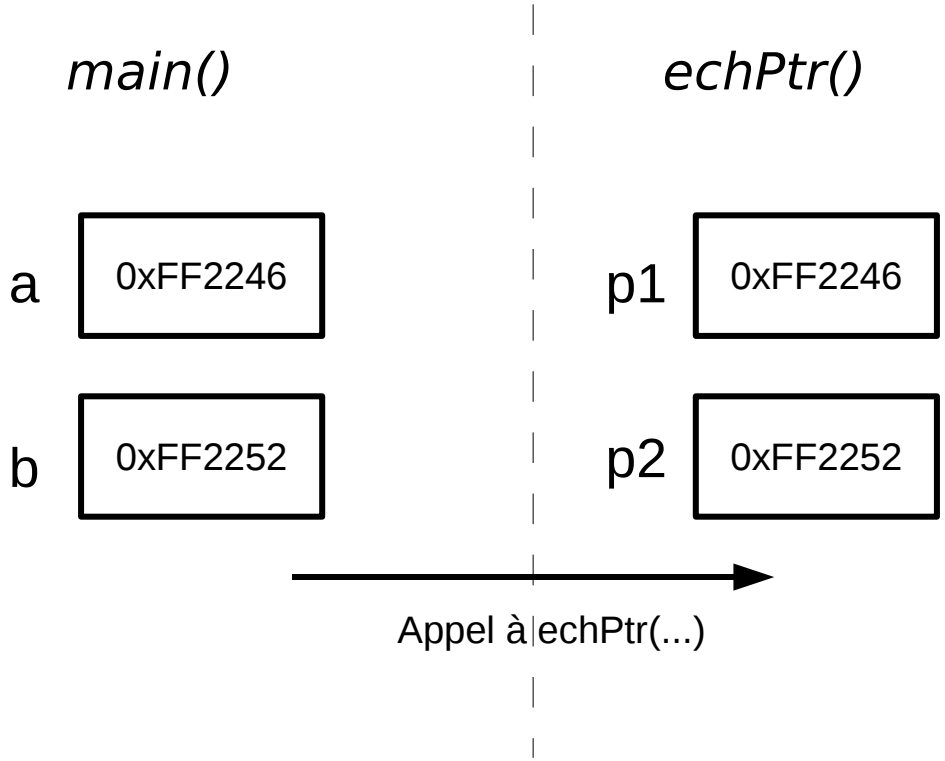
main()

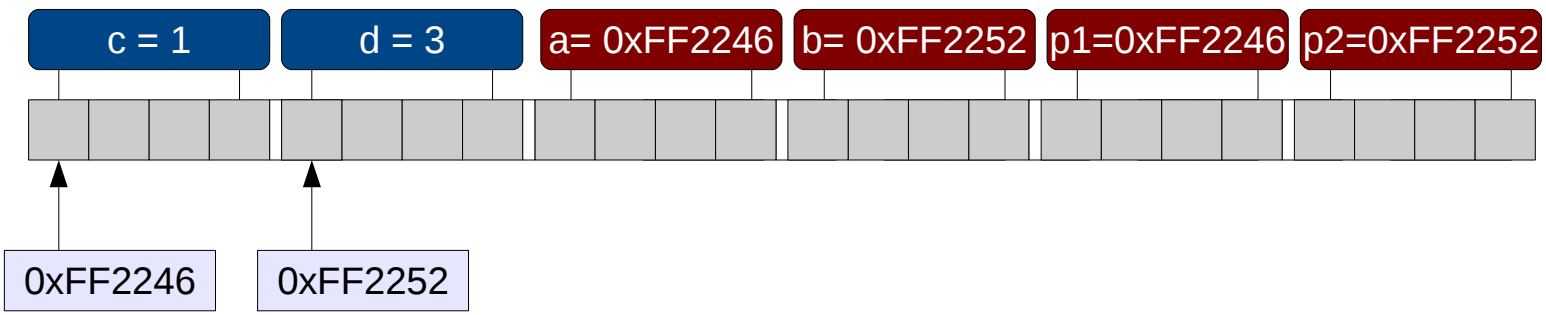
echPtr()

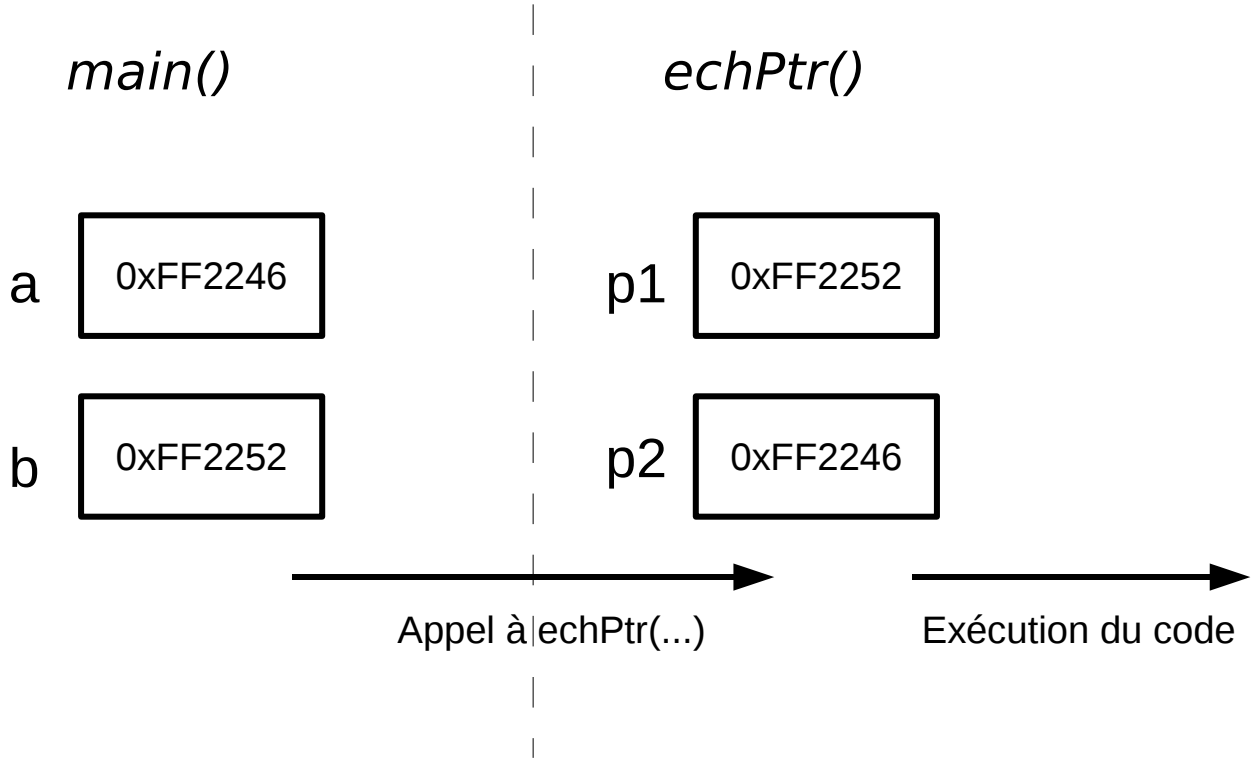
a 0xFF2246

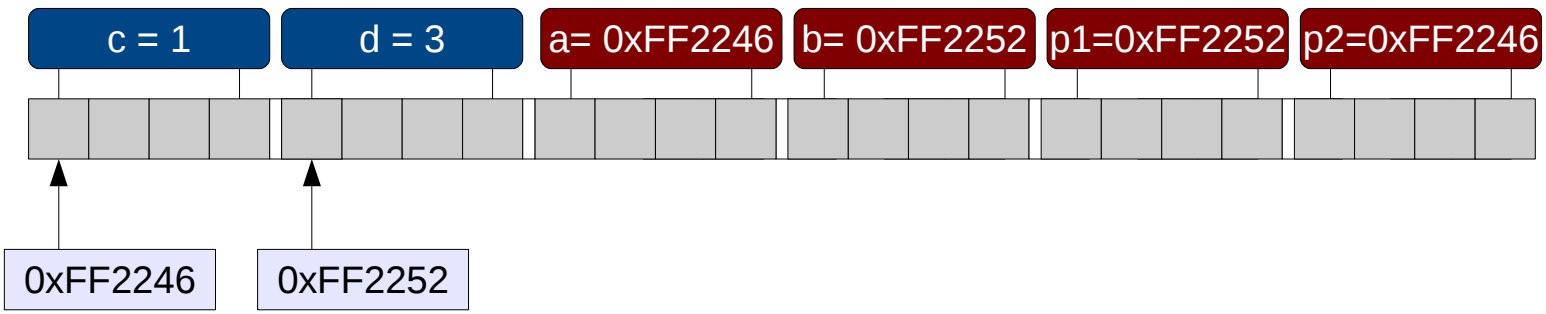
b 0xFF2252

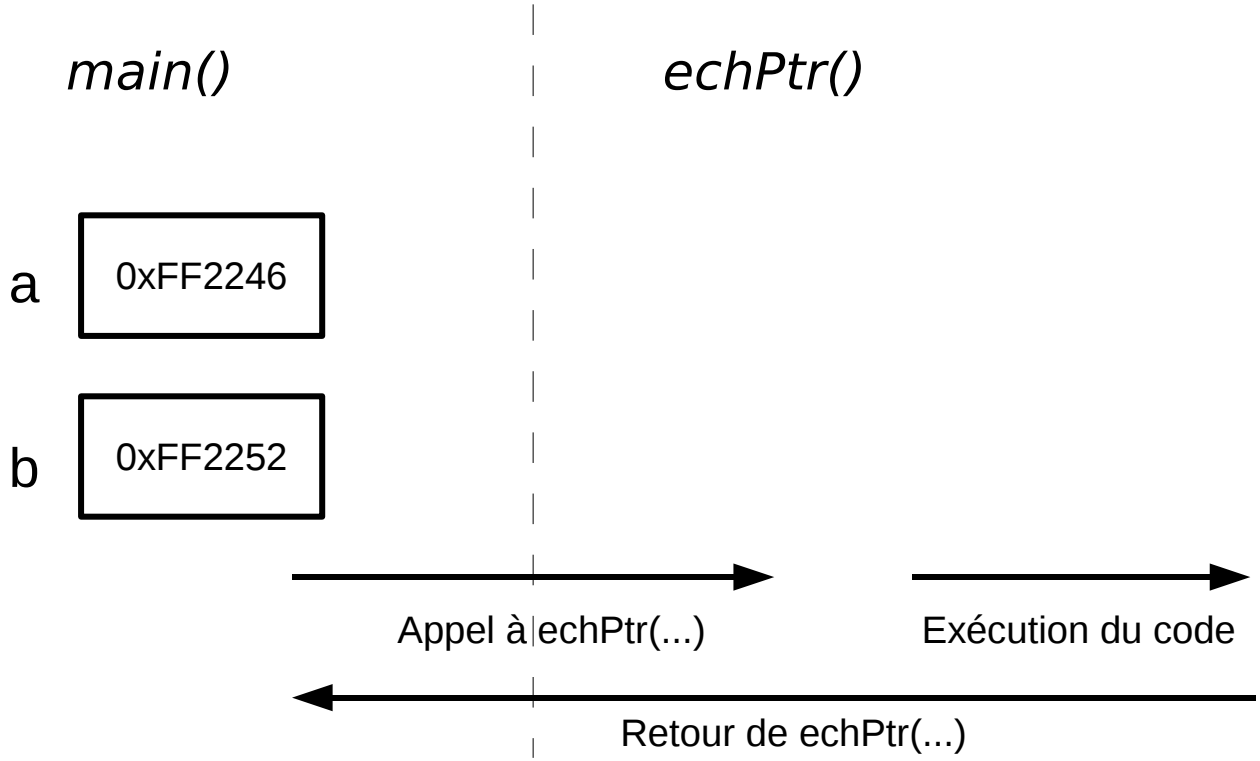


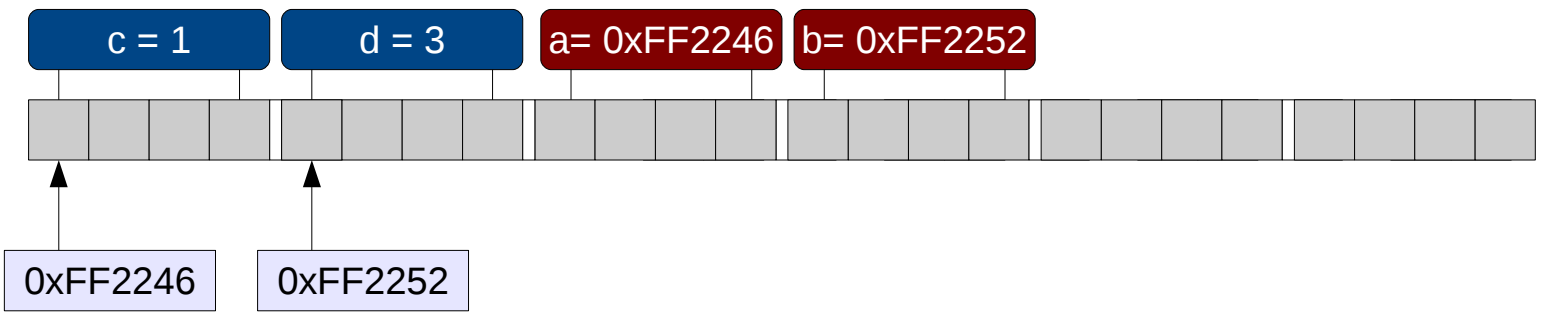

main()
echPtr()
a
0xFF2246
b
0xFF2252
p1
0xFF2246
p2
0xFF2252
Appel à echPtr(...)


c = 1
d = 3
a= 0xFF2246
b= 0xFF2252
p1=0xFF2246
p2=0xFF2252
0xFF2246
0xFF2252


main()
echPtr()
a
0xFF2246
b
0xFF2252
p1
0xFF2252
p2
0xFF2246
Appel à echPtr(...)
Exécution du code


c = 1
d = 3
a= 0xFF2246
b= 0xFF2252
p1=0xFF2252
p2=0xFF2246
0xFF2246
0xFF2252


main()
echPtr()
a
0xFF2246
b
0xFF2252
Appel à echPtr(...)
Exécution du code
Retour de echPtr(...)


c = 1
d = 3
a= 0xFF2246
b= 0xFF2252
0xFF2246
0xFF2252

Fonctions avec pointeurs modifiables

Les pointeurs étant des variables comme les autres, on peut également utiliser un pointeur pour les manipuler par leur adresse.

Definition

type **ptr_ptr ;

La variable `ptr_ptr` est donc une variable qui stocke l'adresse d'un pointeur sur un donnée de type `type`.
On parle de pointeur de pointeur.

Exemple

```
int *p;
int **pp;
pp=&p;
```

Fonctions avec pointeurs modifiables

Definition

```
type_retour mafonction(type_param **ptr){...}
```

A l'intérieur de la fonction :

- on peut récupérer le pointeur : `*ptr`
- on peut modifier le pointeur : `*ptr=...`
- et bien évidemment récupérer/modifier la donnée pointée par le pointeur : `*(*ptr)`

Fonctions avec pointeurs modifiables

Il faut donc réécrire la fonction `echPtr` comme suit :

Exemple

```
void echPtr(int **p1, int **p2){
   int *p;
   p=*p1;
   *p1=*p2;
   *p2=p;
}
```

Fonctions avec pointeurs modifiables : Exemple 1

```
1  #include <stdio.h>
2
3  void echPtr(int **p1, int **p2){
4     int *p;
5     p=*p1;  *p1=*p2; *p2=p;
6  }
7  int main(){
8     int *a,*b;
9     int c,d;
10    c=1;  d=3;
11    a=&c; b=&d;
12    echPtr(a,b);
13    return 0;
14 }
```

Pointeur de pointeur de ... de pointeur

De manière plus générale, l'imbrication de plusieurs niveau de pointeurs n'est pas limitée.

Definition

`int` $\underbrace{\texttt{*...*}}_{\text{n fois}}$ `a;`

définit la variable `a` comme `n` imbrication de pointeur sur entier.

On peut donc stocker dans `a` l'adresse d'une variable étant définie comme imbrication de `n-1` niveaux de pointeur sur un entier.

Plan

Vous avez utilisé en TP les fonctions `lire_entier()` et `lire_reel()` pour saisir des valeurs au clavier.

En réalité ces deux fonctions utilisent la fonction de lecture standard de la bibliothèque `stdlib.h` : la fonction `scanf`

La fonction scanf

Definition

```
scanf(chaine, add1, add2, ..., addn);
```

- `add1, add2, ..., addn` sont les adresses mémoires des variable dans lesquelles on écrira les valeurs saisies
- le nombre de ces adresses est quelconque

La fonction scanf

Definition

```
scanf(chaine, add1, add2, ... , addn)
```

Le scanf :

- ne connait pas le type des données représenté par leur adresse mémoire
- chaine est une chaine de caractère fixant le format d'interprêtation des valeurs saisies.
  - Attention : aucun autre texte ne doit être ajouté

scanf : exemple 1

Exemple

Saisir une variable de type entier au clavier

```
1 #include <stdio.h>
2 int main(){
3    int a;
4    scanf("%d", &a);
5    printf("l'entier saisi est %d\n",a);
6
7  return 0;
8 }
```

Ce code affichera à l'écran l'entier qui sera saisie au clavier.

scanf : les formats de saisie

Les format de saisie des valeurs sont identiques à ceux utilisés par le `printf`.

rappel des formats :

- %c → char
- %d → int
- %f → float (écriture décimale)
- %e → double ou float (écriture scientifique)
- %s → chaîne de caractère (passage variable par adresse)
- %x → codage hexadécimal

scanf : Exemple 2

On peut saisir plusieurs valeurs en même temps :

```
1 #include <stdio.h>
2 int main(){
3    int a;
4    float b;
5    scanf("%d%f",&a,&b);
6    printf("entier=%d\n reel=%f\n",a,b
7    return 0;
8 }
```

Attention

Les valeurs saisies sont séparées par un espace ou par un retour à la ligne.

scanf : maîtriser sa lecture

On peut spécifier au scanf le gabarit de lecture d'une variable (nbr caractères).

- on ajoute un entier entre le % et le format de la variable
- ex : "%3d"

L'entier spécifie le nombre maximun de caractère à saisir :

- la lecture s'arrète dès que le gabarit est atteint
- le gabarit n'a aucune action si le nbr de caractère saisie lui est inférieur

scanf : Exemple 2

```
1 #include <stdio.h>
2 int main(){
3    int a,b;
4    scanf("%2d %d",&a,&b);
5    printf("a=%d\nb=%d\n",a,b);
6    return 0;
7 }
```

Attention

Si le premier entier saisie à plus de 2 caractères, la saisie se termine en affectant les 2 premiers caractères à a et le reste à b.

scanf : maîtriser sa lecture

On peut spécifier au `scanf` les délimiteurs de saisie entre chaques variables.

- on ajoute dans la chaîne de caractère la valeur du délimiteur entre chaque format de variable
- ex : "%d ; %d"

Le délimiteur spécifie le texte attendu entre chaque saisie :

- la lecture s'arrête dès qu'une erreur survient (ex. mauvais délimiteur)
- l'utilisation d'espace autorise le retour chariot entre les délimiteurs

scanf : Exemple 3

```c
#include <stdio.h>
int main(){
   int a,b;
   scanf("%d : %d",&a,&b);
   printf("a=%d\nb=%d\n",a,b);
   return 0;
}
```

une saisie correcte est : `12 : 128`

Attention

L'utilisation d'un délimiteur particulier invalide les délimiteurs par défaut (espace et retour chariot).

Remarque

Les adresses utilisées par le `scanf` peuvent être transmises par des pointeurs...

```
1 #include <stdio.h>
2 int main(){
3    int a;
4    int ptr=&a;
5    scanf("%d",p);
6    printf("a=%d\,a);
7    return 0;
8 }
```

De manière assez générale, pointeur et adresse sont équivalents en C, l'un défini une variable quand l'autre défini une valeur.

Exercices !

Dans le programme ci-dessous, évaluez après chaque instruction (à partir de la ligne 8) les valeurs des entiers a, b et c.

```
1  #include<stdio.h>
2  int main(){
3  int a,b,c;
4  int *p1, *p2;
5  a=b=c=3;
6  p1=&a;
7  p2=&b;
8  c=a+b;
9  *p2=b+2;
10 a=*p2**p1;
11 p1=p2;
12 *p2=*p1-a;
13 p2=&(*p1);
14 *p2=*p1+ *(&(*p2));
15 a=b+*p2;
16 return 0;
17 }
```

Dans le programme ci-dessous, évaluez après chaque instruction (à partir de la ligne 8) les valeurs des entiers *a*, *b* et *c*.

instruction	a	b	c	p1	*p1	p2	*p2
	3	3	3	&a	a(3)	&b	b(3)
`c=a+b;`	3	3	6	&a	a(3)	&b	b(3)
`*p2=b+2;`	3	5	6	&a	a(3)	&b	b(5)
`a=*p2**p1;`	15	5	6	&a	a(15)	&b	b(5)
`p1=p2;`	15	5	6	&b	b(5)	&b	b(5)
`*p2=*p1-a;`	15	-10	6	&b	b(-10)	&b	b(-10)
`p2 = &(*p1);`	15	-10	6	&b	b(-10)	&b	b(-10)
`*p2=*p1+(*(&(*p2))`	15	-20	6	&b	b(-20)	&b	b(-20)
`a=b+*p2;`	-40	-20	6	&b	b(-20)	&b	b(-20)

On cherche à construire une fonction qui prend trois entiers a, b, c en paramètre et qui retourne un pointeur sur le plus grand des entiers. Est-ce que la fonction suivante répond à notre problème ?

```
1 int* max(int a, int b, int c){
2 if (a>=b && a>=c) return &a;
3 if (b>=a && b>=c) return &b;
4 if (c>=a && c>=b) return &c;
5 }
```

Si votre réponse est non, modifiez la fonction pour qu'elle fonctionne correctement.

Plan

Plan

Manipulation des données

Une seule donnée à la fois (données scalaires) :

- une variable : `int a;`
- un pointeur : `int *p;`

Comment faire pour regrouper des données et les manipuler de manière uniforme ?

Manipulation des données

Une seule donnée à la fois (données scalaires) :

- une variable : `int a;`
- un pointeur : `int *p;`

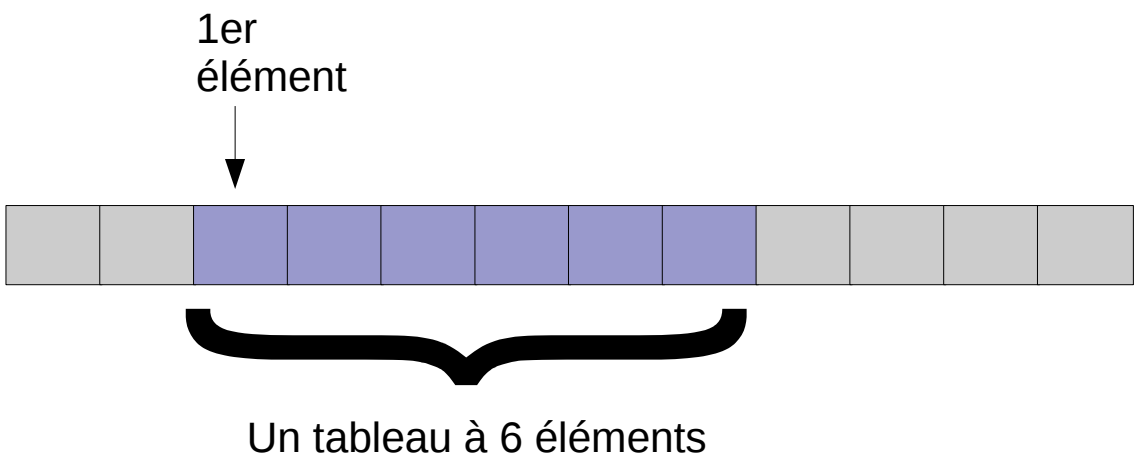
Comment faire pour regrouper des données et les manipuler de manière uniforme ?

utiliser une structure de tableau

La structure de donnée tableau

Definition

Un tableau est un ensemble de données de même type qui sont stockées de manière contigu en mémoire



- un tableau est identifiable par son 1er élément et sa taille
- les éléments sont accessibles en parcourant la mémoire à partir du 1er élément

Les tableaux en C

Definition

Un tableau en C permet de définir avec une seule variable un ensemble de variables de même type de donnée.

Un tableau C est caractérisé par :

- son identificateur (son nom)
- le type de données qu'il stocke
- sa taille (le nombre de données)

Les tableaux en C

Déclaration

```
type T[n];
```

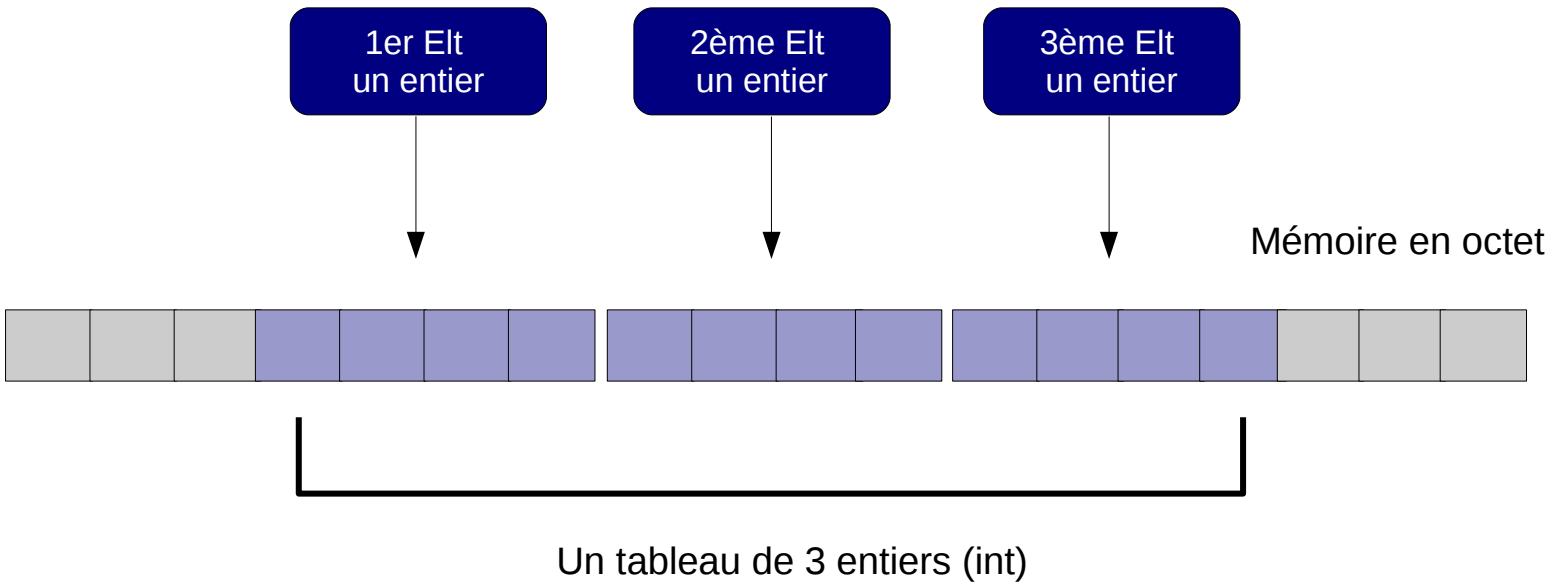
- `T` est l'identificateur du tableau
- `type` est le type de donnée des éléments du tableau
- `n` est une constante entière représentant la taille du tableau

Exemple

```
int T[3];
```

définit la variable T comme un tableau de trois entiers.

Les tableaux en C : vue mémoire



Un tableau de `n` éléments occupe `n×sizeof(type)` octets en mémoire où `type` est le type de donnée des éléments du tableau.
Ici, un tableau de 3 int *occupe 12 octets en mémoire.*

Les tableaux en C

Déclaration

```
type T[n];
```

Cette déclaration de tableau

- crée une variable `T` de type tableau sur `type`
- alloue un espace contigu en mémoire de taille suffisant pour stocker `n` données
- n'initialise pas les éléments du tableau

Attention

Il est préférable que la taille `n` du tableau soit une constante connue à la compilation.

Les tableaux en C : accès aux éléments

L'accès aux éléments d'un tableau se fait par un calcul d'adresse mémoire à partir du 1er élément : l'opérateur [ ] facilite le calcul.

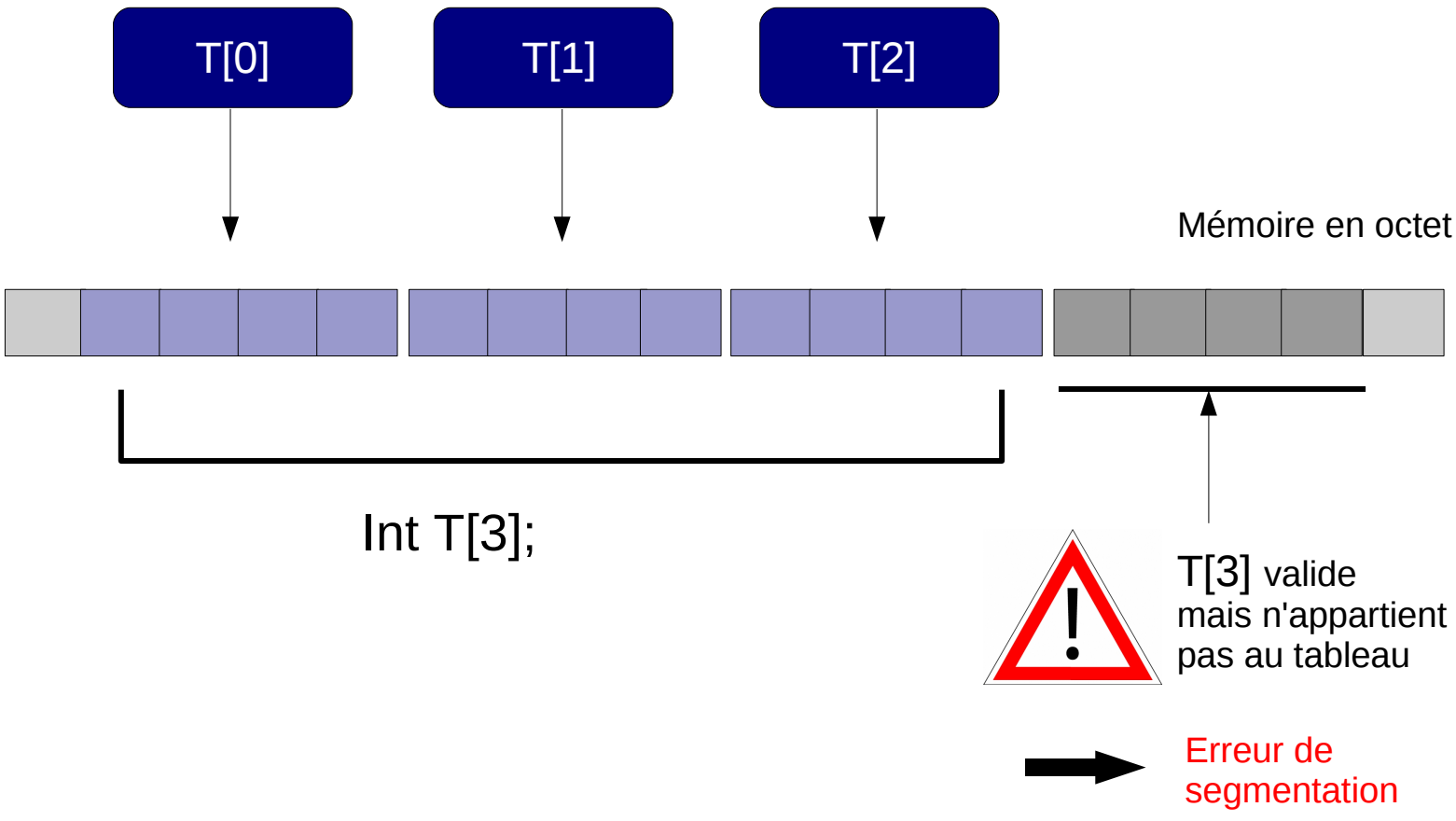
Definition

Soit la déclaration `type T[n];`
`T[i]` donne accès à la i+1-ème case du tableau `T`

Attention :

- les indices du tableau `T` commencent à 0 et se termine à n-1
- aucune vérification numérique est faite pour la valeur de `i`
- `T[i]` est un identificateur valide (une variable)

Les tableaux en C : vue mémoire



Les tableaux en C : initialisation des éléments

Deux possibilités :

- lors de la déclaration du tableau
- par initialisation successive des éléments du tableau

Les tableaux en C : initialisation des éléments

Déclaration d'un tableau avec initialisation des éléments :

Déclaration

`type T[n] =` $\{v_0, v_1, \ldots, v_{n-1}\}$`;`

Cette déclaration permet de déclarer un tableau `T` de `n` variables et d'initialiser les éléments du tableau tels que $T[i] = v_i$.

Exemple

```
int T[3]={4,17,3};
```

Les tableaux en C : initialisation des éléments

Déclaration d'un tableau avec initialisation des éléments :

Déclaration

`type T[n] =` $\{v_0, v_1, \ldots, v_{n-1}\}$`;`

Cette déclaration permet de déclarer un tableau `T` de `n` variables et d'initialiser les éléments du tableau tels que $T[i] = v_i$.

Exemple

```
int T[3]={4,17,3};
```

Remarque : *la taille* n *ou toutes les valeurs ne sont pas obligatoires.*

Les tableaux en C : initialisation des éléments

Déclaration d'un tableau avec initialisation des éléments :

Plusieurs possibilités :

- `int T[]={1,2,3,4,5};`
  ↪ tableau de 5 éléments initialisé avec les valeurs données

- `int T[5]={12,13};`
  ↪ tableau de 5 éléments initialisé avec les valeurs données, le reste est initialisé à 0

Les tableaux en C : initialisation des éléments

On peut initialiser les éléments d'un tableau dans une boucle par affectation successive :

Déclaration

```
int T[10], i;
for(i=0;i<10;i++)
   T[i]=...; ← affectation avec la valeur souhaitée
```

Les tableaux en C : initialisation des éléments

On peut initialiser les éléments d'un tableau dans une boucle par affectation successive :

Déclaration

```
int T[10], i;
for(i=0;i<10;i++)
   T[i]=...; ← affectation avec la valeur souhaitée
```

Remarque : on peut remplacer l'affectation par un scanf.

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

Les tableaux en C : parcours des données

Pour parcourir les données d'un tableau, il suffit d'accéder aux éléments de manière itérative.

Exemple

```
int T[10],i;
...
for (i=0;i<10;i++){
   instructions avec T[i]
}
```

Attention avec `T[i]`

il faut toujours garantir que $0 \leq i <$taille du tableau
le compilateur ne le fait pas pour vous ! ! !

Les tableaux en C : exemple 1

Affichage d'un tableau

```c
#include <stdio.h>

int main(){
   int T[5]={4,3,1,8,6};
   int i;
   for (i=0;i<5;i++)
       printf("%d ",T[i]);

   printf("\n");
   return 0;
}
```

Les tableaux en C : exemple 2

Calcul du plus petit élément d'un tableau

```c
#include <stdio.h>
int main(){
   int T[5]={4,3,1,8,6};
   int i,min;

   min=T[0];
   for (i=1;i<5;i++){
       if (T[i]<min)
          min=T[i];
   }
   printf("le min est %d\n",min);
   return 0;
}
```

Les tableaux en C : exemple 3

Renverser l'ordre des éléments d'un tableau

```c
#include <stdio.h>
#define TAILLE 5

int main(){
   int T[TAILLE]={4,3,1,8,6};
   int i,tmp,milieu=TAILLE/2;

  for (i=0;i<=milieu;i++){
      tmp=T[i];
      T[i]=T[TAILLE-1-i];
      T[TAILLE-1-i]=tmp;
  }

   for (i=0;i<TAILLE;i++)
       printf("%d ",T[i]);
   printf("\n");
   return 0;
}
```

Copie de tableau

Considérons le programme suivant :

```
1 int main()
2 {
3    int t1[5]={1,2,3,4,5};
4    int t2[5];
5    t2=t1;
6    return 0;
7 }
```

A la compilation, nous avons le message suivant :

error : incompatible types in assignment

Copie de tableau

Pour recopier un tableau dans un autre, il est obligatoire de passer par des affectations successives :

```
1 int main()
2 {
3    int t1[5]={1,2,3,4,5};
4    int t2[5];
5    int i;
6    for (i=0; i<5; i++)
7         t2[i]=t1[i];
8    return 0;
9 }
```

Les tableaux multidimensionnels en C

Il est possible en C de déclarer des tableaux de tableaux. On les appelle des tableaux multidimensionnels.

Exemple

```
int T[3][2];
```

T est un tableau à 3 éléments où chaque élément est un tableau de 2 entiers.

On peut voir T comme une matrice : $T = \begin{bmatrix} * & * \\ * & * \\ * & * \end{bmatrix}$

Les tableaux multidimensionnels en C

L'enchaînement des opérateur `[ ]` permet de récupérer les éléments d'un tableau multidimensionnels

Déclaration

Soit la déclaration `int T[m][n];`
`T[i][j]` donne accès au `j-ème` élément du `i-ème` tableau de `T`

Dans une vue matricielle, cela donne :

$$\texttt{int T[3][2]} = \begin{bmatrix} \texttt{T[0][0]} & \texttt{T[0][1]} \\ \texttt{T[1][0]} & \texttt{T[1][1]} \\ \texttt{T[2][0]} & \texttt{T[2][1]} \end{bmatrix}$$

Les tableaux multidimensionnels en C

Comme pour les tableaux unidimensionnels, il est possible d'initialiser les éléments du tableau lors de la déclaration.

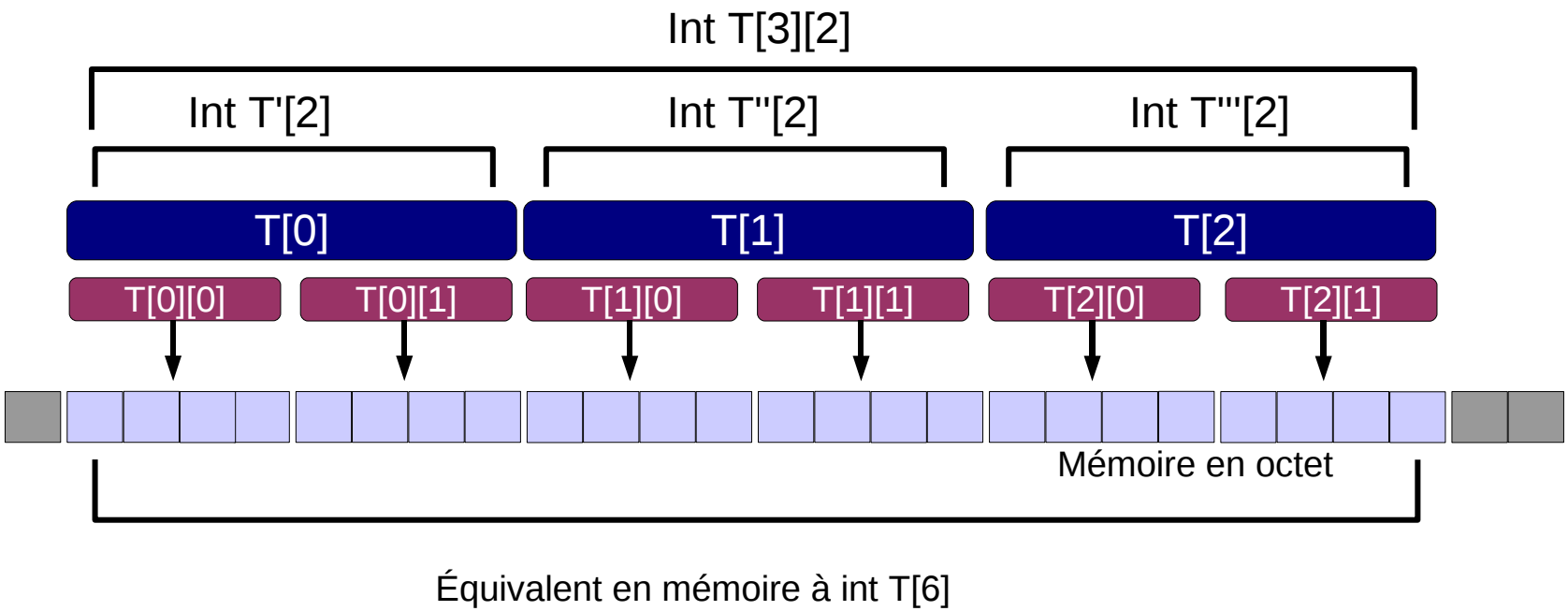
Déclaration

```
int T[3][2] = {{1,2},{3,4},{5,6}};
```

Ce qui donne matriciellement :

$$\texttt{int T} = \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix}$$

Les tableaux multidimensionnels en mémoire



Équivalent en mémoire à int T[6]

Le stockage se fait linéairement sur le tableau le plus profond.

Les tableaux multidimensionnels en C

Au vue du stockage linéaire, on peut utiliser l'initialisation des tableaux unidimensionnels : `int T[3][2] = {1,2,3,4,5,6};`

Plusieurs possibilités de déclarations :

- `int T[][2] = {{1,2},{3,4},{5,6}};`
  `int T[][2] = {1,2,3,4,5,6};`
  tableau multidimensionnel à 3 lignes et 2 colonnes initialisé avec les valeurs données
- `int T[3][2] = {{1},{3,4}};`
  tableau multidimensionnel à 3 lignes et 2 colonnes initialisé avec les valeurs données, le reste est initialisé à 0

Les tableaux multidimensionnels en C

Encore plus fort

- `int T[][2] = {{1},{},2,3,4};`

Les tableaux multidimensionnels en C

Encore plus fort

- `int T[][2] = {{1},{},2,3,4};`
  tableau multidimensionnel à 4 lignes et 2 colonnes initialisé comme la matrice :

$$\begin{bmatrix} 1 & 0 \\ 0 & 0 \\ 2 & 3 \\ 4 & 0 \end{bmatrix}$$

Les tableaux multidimensionnels en C

La notion de tableaux multidimensionnels s'étend bien évidemment à plus de deux dimensions :

Exemple

```
int T1[3][2][4][10][4];
int T2[][2][2]={1,2,3,4,5,6,7,8};
```

Seule la 1ère dimension est optionnelle

Le compilateur a besoin des autres pour calculer l'indexation linéaire des éléments du tableau.

Les tableaux multidimensionnels : Exemple 1

Calcul du déterminant de matrices 2×2

```c
#include <stdlib.h>

int main(){
   int A[2][2]={4,3,5,1};
   int det;

   det = A[0][0]*A[1][1]-A[0][1]*A[1][0];

   printf("le determinant est : %d\n",det);

   return 0;
}
```

Les tableaux multidimensionnels : Exemple 2

Calcul de la transposée d'une matrice carré

```
1  #include <stdio.h>
2  #define N 3
3
4  int main(){
5    int A[N][N];
6    int i,j,tmp;
7
8    for (i=0;i<N;i++)
9      for (j=0;j<N;j++)
10       scanf("%d",&A[i][j]);
11
12   for (i=0;i<N;i++)
13     for (j=i+1;j<N;j++){
14       tmp=A[i][j];
15       A[i][j]=A[j][i];
16       A[j][i]=tmp;
17     }
18
19   return 0;
```

Plan

Tableau en paramètre d'entrée

Syntaxe

type_retour ma_fonction(type ident[taille],...)

où

- `type_retour` peut-être `void`, `int`, `float`,...
- `type` est le type commun à tous les éléments du tableau,
- `ident` est le nom du tableau,
- `taille` est une constante entière.

Tableau en paramètre d'entrée

Syntaxe

type_retour ma_fonction(type ident[taille],...)

Exemple

- void AfficheTab(int tab[10])
- double Somme(double A[20])
- double Moyenne(int t[15])

Tableau en paramètre d'entrée

Il est aussi possible de ne pas préciser la taille.

Syntaxe

type_retour ma_fonction(type ident[],...)

Attention

Il faut bien faire attention à ne pas dépasser la taille du tableau.

Tableau en paramètre d'entrée

Il est ainsi possible d'utiliser une même fonction avec des tableaux de taille différente.
Dans ce cas là, il est important de passer également en argument la taille du tableau.

Syntaxe

```
type_retour ma_fonction(type ident[],int taille...)
```

Cela permet également de n'utiliser qu'une sous-partie du tableau.

Tableau en paramètre d'entrée

Syntaxe

type_retour ma_fonction(type ident[],int taille...)

Exemple

- `void AfficheTab(int tab[], int taille)` : Affiche les *taille* premiers éléments de *tab*
- `double Somme(double A[], int taille)` : Calcule la somme des *taille* premiers éléments de *tab*
- `double Moyenne(int t[], int taille)` : Calcule la moyenne des *taille* premiers éléments de *tab*

Tableau en paramètre d'entrée

L'accès en lecture du i-ème élément d'un tableau `tab` passé en paramètre s'écrit `tab[i]`.
La fonction d'affichage précédente s'écrit alors :

```
1 void AfficheTab(int tab[10])
2 {
3   int i;
4   for (i=0;i<10;i++)
5     printf("tab[%d]=%d\n",i,tab[i]);
6 }
```

ou bien :

```
1 void AfficheTab(int tab[], int taille)
2 {
3   int i;
4   for (i=0;i<taille;i++)
5     printf("tab[%d]=%d\n",i,tab[i]);
6 }
```

Exemple : indice du plus petit élément

On cherche le minimum d'un tableau de 10 entiers, mais on renvoie sa position et non sa valeur.

```
1 int MinTab(int t[10])
2 {
3   int i, m=0;
4   for (i=1;i<10;i++)
5     if (t[i]<t[m])
6       m=i;
7   return m;
8 }
```

Exemple : indice du plus petit élément

Je cherche maintenant le plus petit élément d'un tableau de taille quelconque.

```
1 int MinTab(int t[], int taille)
2 {
3   int i, m=0;
4   for (i=1;i<taille;i++)
5     if (t[i]<t[m])
6       m=i;
7   return m;
8 }
```

Exemple : Appartenance d'une valeur dans un tableau

On cherche à savoir si un réel x apparaît ou non dans un tableau de 20 réels.

```
1 int EstPresent(double tab[20], double x)
2 {
3   int i=0;
4   while ( (i<20) && (tab[i]!=x) )
5     i++;
6   return (i!=20);
7 }
```

Exemple : Appartenance d'une valeur dans un tableau

Je cherche à savoir si un réel x apparaît ou non dans les n premiers éléments d'un tableau de réels.

```
1 int EstPresent(double tab[], int taille, double x)
2 {
3   int i=0;
4   while ( (i<taille) && (tab[i]!=x) )
5     i++;
6   return (i!=taille);
7 }
```

Tableau en retour de fonction

Attention

Dans le langage C, les fonctions ne savent retourner que des valeurs (entiers, réels, pointeurs...), il est donc impossible de retourner comme résultat un tableau statique.

Nous verrons par la suite comment contourner ce problème et obtenir un tableau comme résultat d'une fonction.

Accès en écriture dans un tableau

L'accès en écriture d'un tableau passé en paramètre d'une fonction se fait avec l'opérateur [].

`Tab[i]=...`

```
1 void MaZ(int t[10])
2 {
3   int i;
4   for (i=0;i<10;i++)
5     t[i]=0;
6 }
```

Que se passe-t-il ?

```
1 int main()
2 {
3    int t[10]={3,5,7,2,-5,-7,4,-2,2,1};
4    AfficheTab(t);
5    MaZ(t);
6    AfficheTab(t);
7 }
```

Ques se passe-t-il ?

```
1 int main()
2 {
3    int t[10]={3,5,7,2,-5,-7,4,-2,2,1};
4    AfficheTab(t);
5    MaZ(t);
6    AfficheTab(t);
7 }
```

Le tableau t a été modifié. Pourquoi ?

Plan

Tableau et pointeurs

Comment peut-on modifier une variable ? Avec un pointeur sur celle-ci.

Tout se passe donc comme si le paramètre de la fonction n'était pas le tableau, mais un pointeur sur le tableau.

Remarque

En C, le passage se fait par valeur. Si nous passions effectivement un tableau en paramètre, celui-ci serait intégralement recopié (case par case) dans un tableau local à la fonction. Trop coûteux !

Tableau et pointeurs

Quand on déclare un tableau de 10 entiers :

```
int tab[10];
```

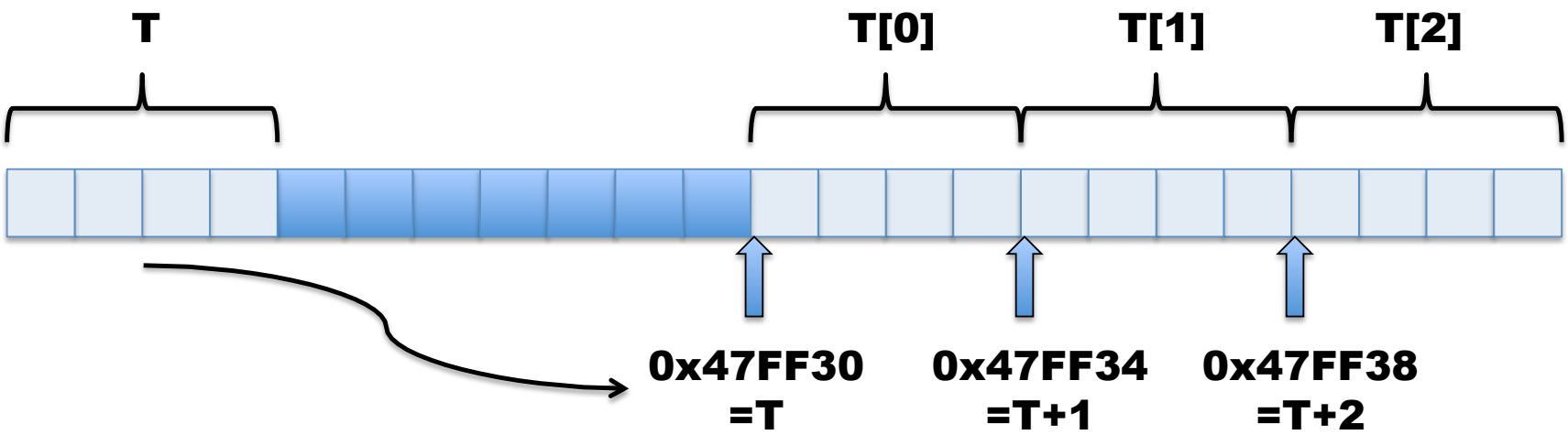
on alloue une zone contigue en mémoire de 10 entiers.
On alors 10 variables :

`tab[0]`, `tab[1]`,...,`tab[9]`

Mais quelle est la nature de `tab` ?

`tab` est un pointeur sur la première case du tableau.

Structure d'un tableau



- `T` contient l'adresse du premier élément du tableau (`&T[0]`)
- les adresses des autres éléments du tableau se déduisent

Arithmétique de pointeurs

si dans `AfficheTab`, on ne connait que l'adresse du premier élément du tableau, comment on trouve `tab[i]` ?
Comme on sait, par la déclaration, que l'on a affaire à un tableau d'entiers, on peux en déduire :

- l'adresse de `tab[1]` est celle de `tab[0]` plus la place que prend un entier,
- ...
- l'adresse de `tab[i]` est celle de `tab[0]` plus la place que prend i entiers.

Le compilateur est donc capable de retrouver l'emplacement de toutes les cases du tableau à partir de la première.

Arithmétique de pointeurs

Peut-on manipuler directement les adresses pour accéder aux éléments du tableau ? oui

Si `tab` est l'adresse du premier élément, alors l'adresse du i-ème élément est

```
tab+i
```

et on peut accéder à un élément en faisant :

```
*(tab+i)
```

Arithmétique de pointeurs

```
1 void AfficheTab(int tab[10])
2 {
3   int i;
4   for (i=0;i<10;i++)
5     printf("%d ",*(tab+i));
6   printf("\n");
7 }
```

Ou encore, en n'utilisant que des pointeurs :

```
1 void AfficheTab(int tab[10])
2 {
3   int *p;
4   for (p=tab;p<tab+10;p++)
5     printf("%d ",*p);
6   printf("\n");
7 }
```

Exemple complet : Tri d'un tableau généré aléatoirement

Une fonction qui modifie un tableau en le remplissant de valeurs aléatoires comprises entre 0 et 20.

```
1 void InitTab(int tab[10])
2 {
3     int i;
4     for (i=0; i<10; i++)
5         tab[i]=rand() % 21;
6 }
```

Exemple complet

Une fonction qui trie un tableau de taille quelconque.

```
1  int Tri(int t[], int taille)
2  {
3      int i,j,m;
4      for (i=0;i<taille-1;i++)
5      {
6          m=i;
7          for (j=i;j<taille;j++)
8              if (t[j]<t[m])
9                  m=j;
10         echange(t+i,t+m);
11     }
12 }
```

La fonction `echange` étant celle du cours précédent.

Exemple complet

L'utilisation de tableau aléatoire permet de tester facilement son algorithme sur de nombreux cas différents.

```
1  int main()
2  {
3     int t[10];
4     srand(time());
5     InitTab(t);
6     AfficheTab(t);
7     Tri(t,5);
8     AfficheTab(t);
9     Tri(t,10);
10    AfficheTab(t);
11    return 0;
12 }
```

Exemple complet

A la première exécution, on obtient (par exemple) :

```
7  15  8  12  12  5  6  17  2  1
7  8  12  12  15  5  6  17  2  1
1  2  5  6  7  8  12  12  15  17
```

A la seconde, on a :

```
7  5  4  20  14  11  17  1  7  7
4  5  7  14  20  11  17  1  7  7
1  4  5  7  7  7  11  14  17  20
```

Plan

Cas des tableaux 2D

Pour utiliser un tableau en paramètre d'une fonction, la syntaxe est identique :

Syntaxe

type_retour ma_fonction(type ident[taille_1][taille_2],...)

Exemple

- `void AfficheMatrice(int tab[3][3])`
- `float DetMatrice(float C[3][3])`
- ...

Attention

Attention

Comme pour l'initialisation de tableau, seule la première dimension peut être omise.

On peut écrire

- `void AfficheMatrice(int tab[][3])`
- `float DetMatrice(float C[][3])`

Mais pas

- `void AfficheMatrice(int tab[][])`
- `float DetMatrice(float C[][])`

Tableau et pointeurs

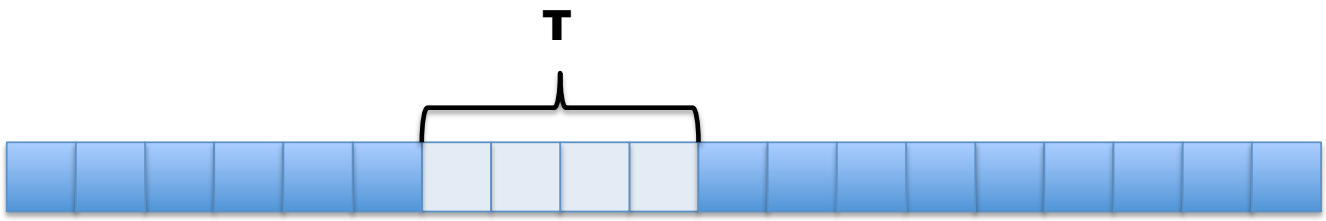
Comme dans le cas 1D, un tableau est toujours un pointeur vers le premier élément. Les éléments du tableau sont donc toujours modifiable dans une fonction.

Attention

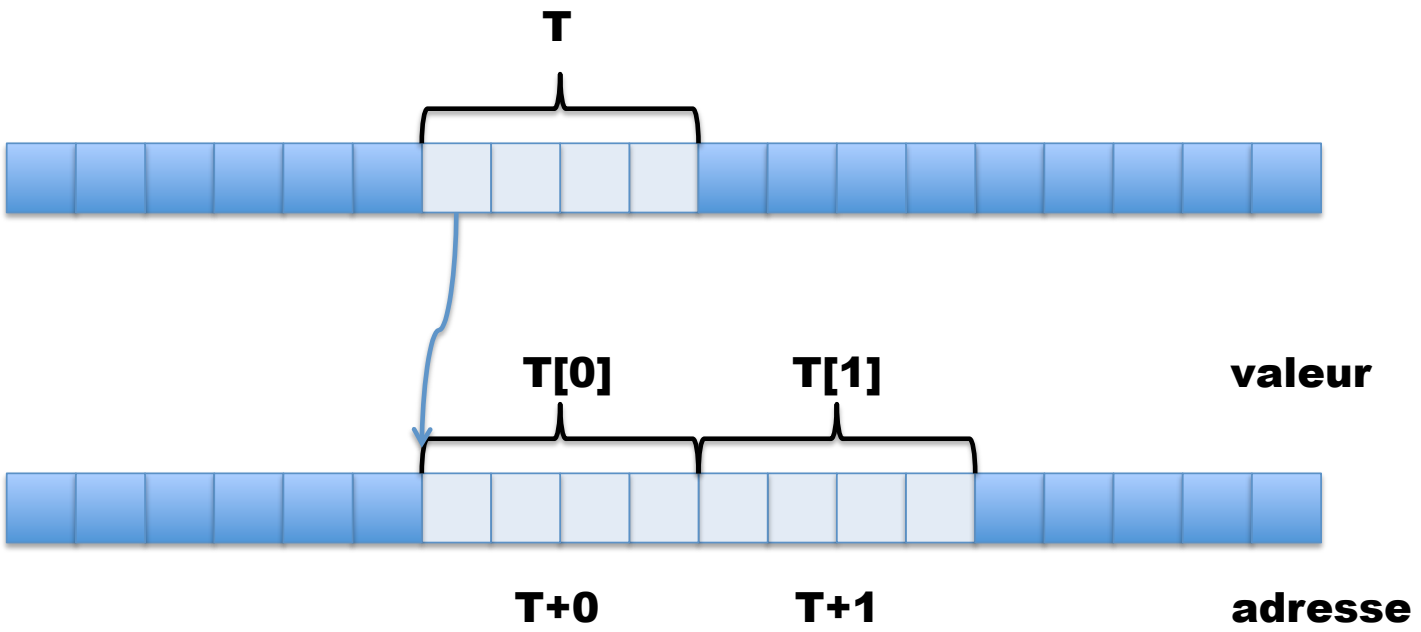
Si T est un tableau 3×3, alors le premier élément de T est T[0] et donc un tableau de 3 éléments.

L'arithmétique des pointeurs est alors plus complexe.

Structure d'un tableau 2D



Structure d'un tableau 2D



Structure d'un tableau 2D

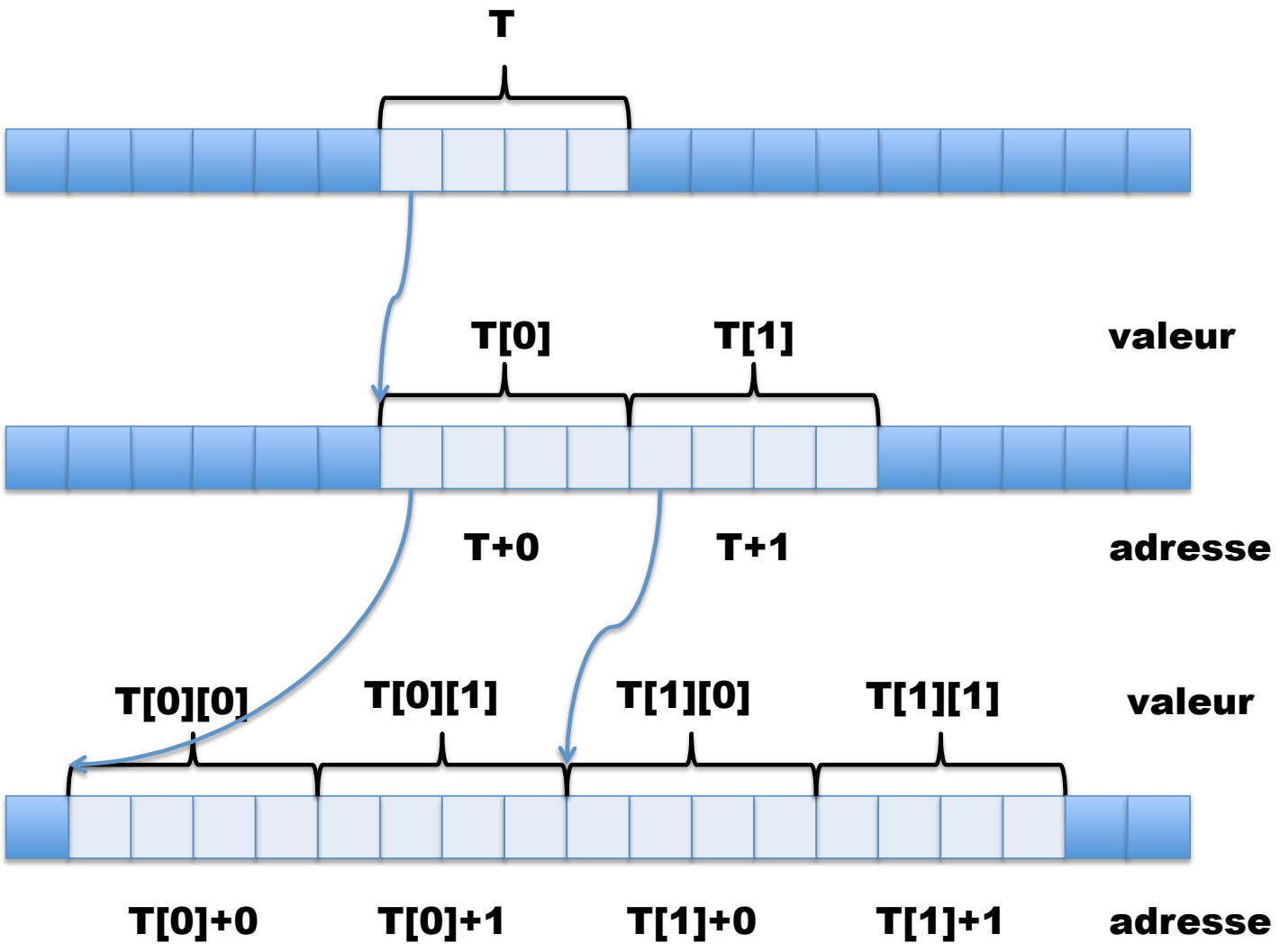


Tableau et pointeurs

- `T` est un pointeur sur `T[0]` (lui même un pointeur sur `T[0][0]`),
- `T+i` est un pointeur sur `T[i]` (lui même un pointeur sur `T[i][0]`),
- `*(T+i)` est donc un pointeur sur `T[i][0]`,
- `*(T+i)+j` est donc un pointeur sur `T[i][j]`.

On a donc équivalence entre `T[i][j]` et `*(*(T+i)+j)`

Tableau et pointeurs : Exemple

En utilisant l'arithmétique des pointeurs, on peut écrire une fonction d'affichage d'un tableau 3×3 comme suit :

```
1 int affiche(int tab[3][3])
2 {
3     int i,j;
4     for (i=0;i<3;i++)
5         for (j=0;j<3;j++)
6             printf("%d ",*(*(tab+i)+j));
7 }
```

Tableau et pointeurs : Exemple

Sur l'exemple précédent, l'arithmétique des pointeurs n'apporte rien (si ce n'est rendre le programme moins lisible).

Par contre, on peut tirer profit de la linéarité de la mémoire pour ne faire qu'une boucle :

```
1 int affiche(int tab[3][3])
2 {
3     int * p;
4     for (p=tab;p<*tab+9;p++)
5         printf("%d ",*p);
6 }
```

`tab` pointe sur `tab[0][0]` et à partir de lui, tous les autres éléments se suivent.

Exemple

Une fonction qui calcule le produit de matrice 3×3.

```
1  void ProdMat(int A[3][3], int C[3][3], int C[3][3])
2  {
3     int i,j,k;
4     for (i=0; i<3; i++)
5        for (j=0; j<3; j++)
6        {
7           C[i][j]=0;
8           for (k=0; k<3; k++)
9              C[i][j]=C[i][j]+A[i][k]*B[k][j];
10       }
11 }
```

Plan

Modèle mémoire

- Données statiques
  - On connaît à l'avance (lors de l'écriture du programme) la taille des données,
  - La mémoire peut donc être réservée dès le début du programme.
- Données dynamiques
  - On connaît la taille des données lors de l'exécution du programme
  - La réservation de la mémoire doit donc se faire dynamiquement au cours de l'exécution.

Remarques

Sur les tableaux statiques :

- ils sont alloués dans un espace mémoire particulier (la pile)
- cette pile est très limitée
- on ne peut pas allouer beaucoup de tableaux, ni de grands tableaux

Sur les tableaux dynamiques

- ils sont alloués dans une autre zone mémoire (le tas)
- cet espace mémoire est très vaste
- on peut y allouer beaucoup de grands tableaux

Outils

- Nécessité d'avoir des outils de gestion de le mémoire :
  - Réservation de l'espace mémoire :
    - d'une taille donnée
    - d'un type donnée
  - Libération de l'espace mémoire

- Utilisation des pointeurs pour accéder à ces espaces mémoires

Allocation mémoire

La bibliothèque `<stdlib.h>` fournit la fonction adéquate :

```
void * malloc(size_t taille)
```

La fonction `malloc` renvoie :

- un pointeur générique sur une zone mémoire de taille *taille* octets.
- NULL si l'allocation a échoué

`size_t` est un type entier non signé utilisé pour les tailles.

Exemples

Remarque

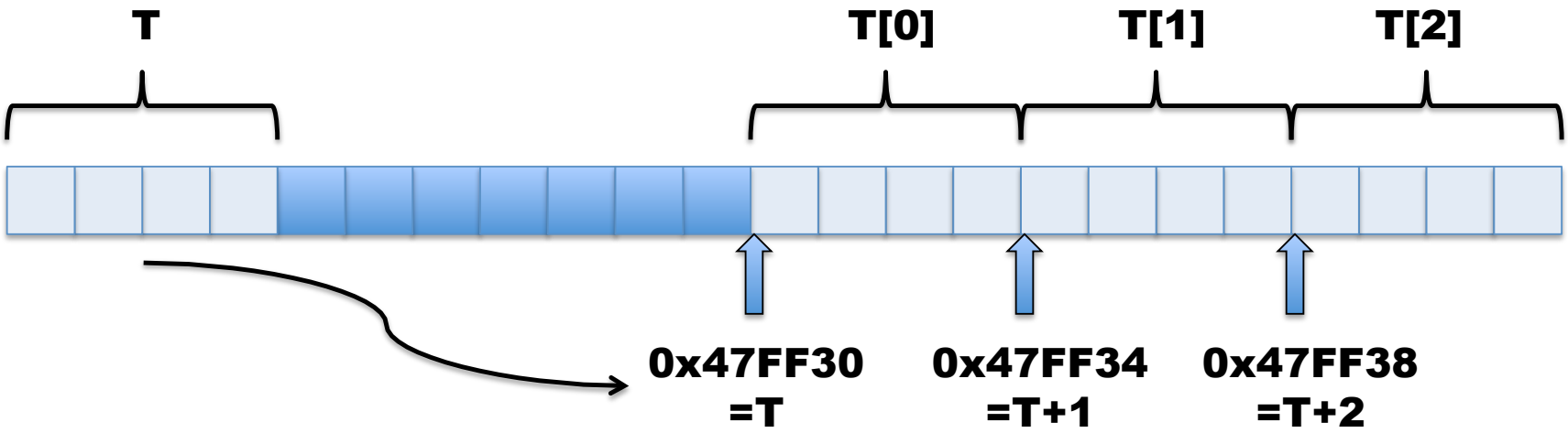
La fonction `sizeof` donne la taille en octets occupée par un objet.

Exemple

- `malloc(50*sizeof(int))` renvoie un pointeur sur une zone mémoire capable de contenir 50 entiers
- `malloc(10*sizeof(double))` renvoie un pointeur sur une zone mémoire capable de contenir 10 réels (des doubles)

Représentation en mémoire

Quand on utilise malloc, on alloue une zone mémoire et on obtient en retour un pointeur vers le début de cette zone mémoire.



Utilisation

Une fois cette zone mémoire allouée, on peut la voir comme un tableau et utiliser l'opérateur [] ou utiliser de l'arithmétique des pointeurs.

```
1  int main()
2  {
3      float * tab_dyn;
4      tab_dyn = malloc(40*sizeof(float));
5
6      for (i=0; i<40; i++)
7          tab_dyn[i]=1./i;
8
9      for (i=0; i<40; i++)
10         printf("%f ",tab_dyn+i);
```

Exemple

On veut qu'un utilisateur rentre ses valeurs dans un tableau (pour ensuite, par exemple, en faire la moyenne...)

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *adr;
    int i, N;
    printf("Entrez le nombre de valeurs : ");
    scanf("%d",&N);
    adr=malloc(N*sizeof(int));
    for (i=0;i<N; i++){
        printf("Entrez la valeur %d : ",i);
        scanf("%d",&adr[i]);
    }
    return 0;
}
```

Exemple : avec arithmétique des pointeurs

Comme on sait que la variable `adr[i]` est situé à l'adresse `adr+i`, on peut modifier le code précédent :

```
1     for (i=0;i<N; i++)
2     {
3         printf("Entrez la valeur %d : ",i);
4         scanf("%d",adr+i);
5     }
```

Mémoire pleine

- Lorsque la mémoire a complétement été allouée, `malloc` renvoie la valeur `NULL`.

- Il FAUT tester cette valeur à chaque allocation de mémoire !

Exemple correct

Avant d'aller plus loin dans l'exécution du programme, on doit vérifier que l'allocation a pu être faite :

```
1 int main()
2 {
3     int * tab;
4     tab = malloc(10*sizeof(int));
5     if (tab==NULL)
6         exit(0);
7     // Suite du programme
8     return 0;
9 }
```

Bibliothèque assert

La biliothèque <assert.h> fournit une fonction permettant d'interrompre conditionnellement un programme.

Cette fonction s'appelle assert(cond) et permet de continuer le programme unqiuement quand l'expression cond est vrai.

```
1 #include <stdio.h>
2 #include <assert.h>
3 int main()
4 {    int i=0;
5      assert(i!=0);
6      printf("i n'est pas nul !!!\n");
7      return 0;}
```

A l'exécution, on obtient le message suivant :

Assertion failed : (i!=0), function main, file test.c, line 5.

Bonne façon de faire (1)

La bonne manière finalement d'utiliser l'allocation dynamique de mémoire est donc la suivante :

```
1  #include <stdio.h>
2  #include <assert.h>
3  int main()
4  {
5      int * tab;
6      tab = malloc(10*sizeof(int));
7      assert(tab!=NULL);
8      // Suite du programme
9      return 0;
10 }
```

Libération de la mémoire

Nous sommes capables de réserver de la mémoire. Celle-ci est limitée, il faut donc pouvoir la libérer.

```
free(void *p)
```

- libère l'espace mémoire pointé par le pointeur `p`.
- le pointeur `p` est toujours utilisable.
- La mémoire doit avoir été préalablement allouée dynamiquement.

Bonne façon de faire (2)

La bonne manière finalement (encore une fois) d'utiliser l'allocation dynamique de mémoire est donc la suivante :

```c
#include <stdio.h>
#include <assert.h>
int main()
{
    int * tab;
    tab = malloc(10*sizeof(int));
    assert(tab!=NULL);
    // Suite du programme
    free(tab); //je n'ai plus besoin du contenu de tab
    // Suite du programme qui peut utiliser tab
    return 0;
}
```

Initialisation

Pour initialiser un tableau, on ne peut pas le faire à la déclaration, il n'a pas encore d'espace mémoire propre.

On ne peut le faire qu'après le `malloc` par affectations successives.

```
1      int * tab, i;
2      tab = malloc(N*sizeof(int));
3      assert(tab!=NULL);
4
5      for (i=0;i<N;i++)
6          tab[i]=i;
```

Exemple : extraire les éléments pairs d'un tableau

On suppose avoir un tableau de 40 entiers et on cherche à garder que les pairs.

```
1  int i, c=0, j=0;
2  int * tab;
3  for (i=0;i<40;i++)
4       if (T[i]%2==0)
5            c++;
6  tab = malloc(c*sizeof(int));
7  assert(tab!=NULL);
8  for (i=0;i<40;i++)
9       if (T[i]%2==0)
10      {
11           tab[j]=T[i];
12           j++;
13      }
```

Manque de place

Il peut arriver que la mémoire allouée se retrouve insuffisante. Imaginons un programme de gestion de notes et des étudiants qui s'inscrivent en retard...

```
void * realloc(void *p, size_t taille)
```

La fonction `realloc` modifie la taille de la zone pointée par `p`.

- renvoie l'adresse de la nouvelle zone
- `NULL` si la réallocation a échoué : `p` n'est alors pas modifié.

Fonction `realloc`

Son fonctionnement :

- Si nouvelle taille $<$ ancienne taille alors seuls les éléments du début de l'ancien contenu sont conservés.
- Si nouvelle taille $>$ ancienne taille alors l'ancien contenu est intégralement conservé dans les premières positions de la nouvelle zone mémoire.
- La zone doit avoir été une première fois allouée à l'aide de `malloc`.

Exemple

```
1  int main()
2  {
3      int i, N, *adr;
4      printf("Taille de votre tableau : ");
5      scanf("%d",&N);
6      adr=malloc(N*sizeof(int));
7      assert(adr!=NULL);
8      // Je me sers du tableau adr...
9      printf("Nouvelle taille de votre tableau : ");
10     scanf("%d",&N);
11     adr=realloc(adr, N);
12     assert(adr!=NULL);
13     // le contenu de adr n'a pas changé,
14     // mais j'ai plus de places disponibles.
15     free(adr);
16     return 0;
17 }
```

Extraire les éléments pairs d'un tableau de taille 40

```
1  int N,j;
2  int * tab;
3  for (i=0;i<40;i++)
4       if (T[i]%2==0)
5            N++;
6  tab = malloc(N*sizeof(int));
7  j=0;
8  for (i=0;i<40;i++)
9       if (T[i]%2==0)
10           {
11                tab[j]=T[i];
12                j++;
13           }
```

Une autre solution

```
1  int N=0, Nmax=10;
2  int * tab;
3  tab = malloc(Nmax*sizeof(int));
4  for (i=0;i<40;i++)
5       if (T[i]%2==0)
6       {
7            tab[N]=T[i];
8            N++;
9            if (N==Nmax)
10           {
11                Nmax=Nmax+5;
12                tab=realloc(tab,Nmax);
13           }
14      }
```

Encore une autre

```
1  int N=0, Nmax=10;
2  int * tab;
3  tab = malloc(40*sizeof(int));
4  for (i=0;i<40;i++)
5       if (T[i]%2==0)
6       {
7            tab[N]=T[i];
8            N++;
9       }
10 tab=realloc(tab, N*sizeof(int));
```

Comparation des solutions

Solution 1 : Deux parcours du tableau initial, long et couteux...

Solution 2 : Potentiellement, beaucoup de realloc effectués

Solution 3 : Potentiellement, on crée un tableau beaucoup trop grand pour rien

La bonne solution va dépendre des cas d'utilisation : à vous de voir !

Copie de tableau

```
1  int main()
2  {
3      int i, *t1, *t2;
4      t1=malloc(20*sizeof(int));
5      t2=malloc(20*sizeof(int));
6      for (i=0;i<20;i++)
7          t1[i]=i;
8      t2=t1;
9      for (i=0;i<20;i++)
10         printf("%d ",t2[i]);
11 }
```

Copie de tableau

A l'exécution, nous obtenons bien les valeurs de 0 à 19 qui s'affichent.

Avons-nous effectué une copie de tableau ?

NON : c'est une copie virtuelle, `t2` n'est pas une copie de `t1`. Les 2 pointeurs indiquent la même zone mémoire. Une modification de `t1` modifiera également `t2`.

Copie de tableau

La bonne solution pour recopier `t1` dans `t2` est de procéder par affectation successive.

```
1  int main()
2  {
3      int i, *t1, *t2;
4      t1=malloc(20*sizeof(int));
5      t2=malloc(20*sizeof(int));
6      for (i=0;i<20;i++)
7          t1[i]=i;
8      for (i=0;i<20;i++)
9          t2[i]=t1[i];
10 }
```

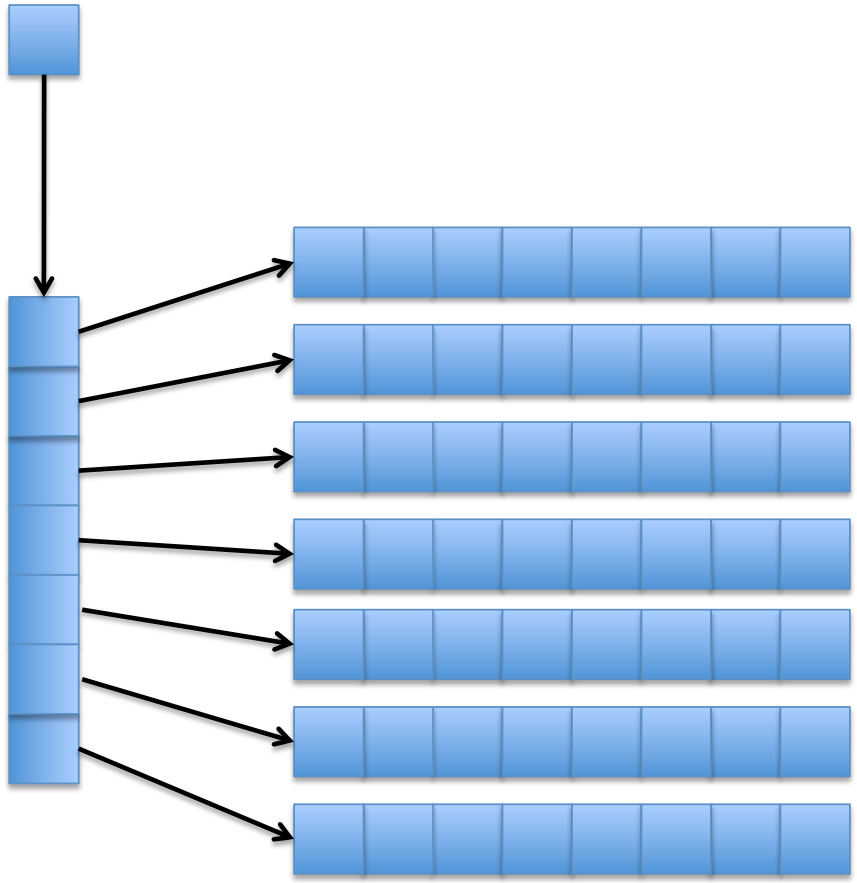
Plan

Tableaux bi-dimensionnels

- On veut utiliser un tableau de taille $N \times M$ entiers
- N et M ne seront connus qu'à l'exécution.
- Comment faire ?

Tableau 7×8

- Un pointeur sur un tableau de 7 cases
- Chaque case contient un pointeur
- Chacun de ses pointeurs indiquent une zone de 8 cases



Mise en oeuvre

```
1  int ** tableau;
2  int i,j;
3
4  tableau=malloc(N*sizeof(int *));
5  for(i=0;i<N;i++)
6      tableau[i]=malloc(M*sizeof(int));
7
8  for(i=0;i<N;i++)
9      for(j=0;j<N;j++)
10         tableau[i][j]=0;
```

Libération de la mémoire

Pour libérer la mémoire, il faut libèrer chacune des zones précédemment allouées :

```
1 for(i=0;i<N;i++)
2       free(tableau[i]);
3
4 free(tableau);
```

Plan

Tableau en paramètre d'entrée

Pour passer un tableau dynamique en paramètre, on doit passer en argument :

- le pointeur sur la zone mémoire correspondante
- la taille du tableau

Tableau en paramètre d'entrée

Syntaxe

type_retour ma_fonction(type * ident, int taille...)

où

- `type_retour` peut-être `void`, `int`, `float`,...
- `type` est le type commun à tous les éléments du tableau,
- `ident` est le nom du tableau dynamique,
- `taille` est un entier.

Exemple

Syntaxe

type_retour ma_fonction(type * ident, int taille...)

Exemple

- `void AfficheTab(int * tab, int n)` : Affiche le tableau *tab* de taille n
- `double Somme(double * A, int m)` : Calcule la somme des *m* éléments de *tab*
- `double Moyenne(int * t, int taille)` : Calcule la moyenne des *taille* éléments de *tab*

Tableau en paramètre d'entrée

L'accès en lecture du i-ème élément d'un tableau `tab` passé en paramètre peut s'écrire `tab[i]`.

```
1 void AfficheTab(int * tab, int n)
2 {
3   int i;
4   for (i=0;i<n;i++)
5     printf("tab[%d]=%d\n",i,tab[i]);
6 }
```

Tableau en paramètre de sortie

Nous savons qu'il est possible de renvoyer un pointeur. Nous allons donc pouvoir renvoyer un tableau dynamique.

Syntaxe

type * ma_fonction(...)

On peut donc imaginer les exemples suivants :
`int * creeTableau(int n)` qui va créer un tableau de taille n et renvoyer son adresse.

Tableau en paramètre de sortie

```
1  int * creeTableau(int n)
2  {
3      int *t, i;
4      t=malloc(n*sizeof(n));
5      for (i=0;i<n; i++)
6          t[i]=0;
7      return t;
8  }
9  int main()
10 {
11     int taille=20;
12     int * tab;
13     tab=creeTableau(taille);
14     printf("tab[4]=%d\n",tab[4]);
15     free(tab);
16     return 0;
17 }
```

Tableau modifiable

Comme les tableaux statiques, les tableaux dynamiques ne sont que des pointeurs.
On en déduit donc que les tableaux dynamiques sont modifiables par les fonctions :

```
1 void MaZ(int * t, int n)
2 {
3     int i;
4     for (i=0;i<n; i++)
5         t[i]=0;
6 }
```

va effectivement modifier le tableau passé en paramètre et le mettre à 0.

Exemple : extraire les nombres premiers d'un tableau

On suppose avoir la fonction `premier(int)` qui renvoie 1 si n est premier et 0 sinon.

On souhaite écrire une fonction qui, à partir d'un tableau, va renvoyer un tableau ne contenant que les premiers.

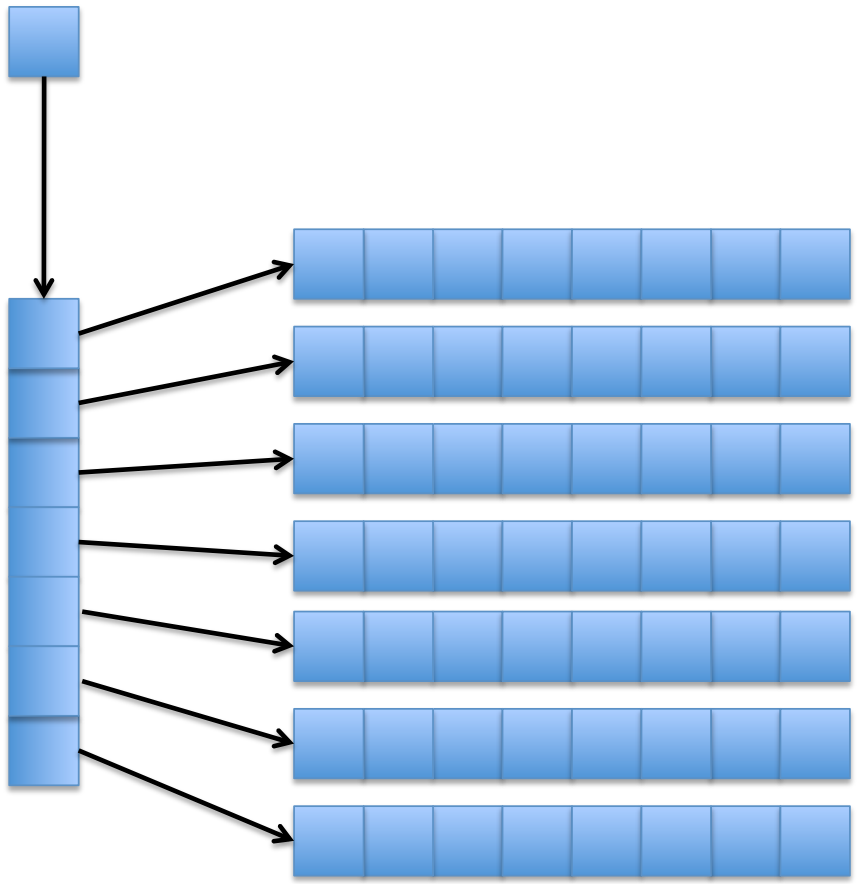
Exemple : extraire les nombres premiers d'un tableau

```
1  int * tabPremier(int * t, int n)
2  {
3      int i, j=0, * r;
4      r=malloc(n*sizeof(int));
5      for (i=0; i<n; i++)
6      {
7          if (Premier?(t[i])==1)
8              {
9                  r[j]=t[i];
10                 j++;
11             }
12     }
13     r=realloc(r, j*sizeof(int));
14     return r;
15 }
```

Cas de la dimension 2

On suppose avoir choisi comme représentation celle-ci :

- Un pointeur sur un tableau de N cases
- Chaque case contient un pointeur
- Chacun de ses pointeurs indiquent une zone de M cases



Cas de la dimension 2

Pour passer un tel objet en paramètre, on doit écrire :

Syntaxe

type_retour ma_fonction(type ** ident, int N, int M,...)

où

- `type_retour` peut-être `void`, `int`, `float *`,...
- `type` est le type commun à tous les éléments du tableau,
- `ident` est le nom du tableau dynamique,
- `N` est un entier représentant la première dimension.
- `M` est un entier représentant la seconde dimension.

Exemple : pluviométrie

Nous allons nous intéresser à l'écriture de différentes fonctions sur des relevés de précipitations.

Ceux-ci seront stockées dans un tableau 2D (mois, jours).

Pour commencer, nous allons commencer par créer un tableau `mois` qui contiendra le nombre de jours du mois correspondant :

```
int mois[12]={31,28,31,30,31,30,31,31,30,31,30,31};
```

Exemple : pluviométrie

Comment créer un tableau `pluie` 2D tel que la ligne i contienne $mois[i]$ cases ?

```
1 int ** pluie;
2 int i;
3 pluie=malloc(12*sizeof(int *));
4 for(i=0;i<12;i++)
5       pluie[i]=malloc(mois[i]*sizeof(int));
```

Exemple : pluviométrie

Première question : calculer le total de précipitations par mois et renvoyer le résultat dans un tableau

```
1  int * pluieMois(int ** pluie, int m[12])
2  {
3      int i, j, *t;
4      t=malloc(12*sizeof(int));
5      for (i=0;i<12;i++)
6      {
7          t[i]=0;
8          for (j=0;j<m[i]; j++)
9          {
10             t[i]+=pluie[i][j];
11         }
12     }
13     return t;
14 }
```

Exemple : pluviométrie

Deuxième question : afficher le jour de l'année où il pleut le plus

```
1  void pluieMax(int ** pluie, int m[12])
2  {
3      int i, j;
4      int mi=0,mj=0;
5      for (i=0;i<12;i++)
6          for (j=0;j<m[i]; j++)
7          {
8              if (pluie[i][j]>pluie[mi][mj])
9                  {
10                     mi=i;
11                     mj=j;
12                 }
13         }
14     printf("il pleut le plus le %d/%d\n",mj+1,mi+1);
15 }
```

Exemple : pluviométrie

Dernière question : calculer le cumulé de la pluviométrie sur l'année

```
1  int pluieAnnuelle(int ** pluie, int m[12])
2  {
3      int i, s=0, *t;
4      t=pluieMois(pluie,m);
5      for (i=0;i<12;i++)
6          {
7              s=s+t[i];
8          }
9      return s;
10 }
```