Cours de Langage C
Pointeurs et tableaux

Objectifs de la séance

- Savoir comment sont stockées les variables en mémoire
- Notion d'adresse
- Connaître les avantages et inconvénients du passage par valeur et du passage par adresse pour les arguments de fonction

Rappel : Stockage des variables

- En base 10 : on exprime les nombres avec 10 chiffres de 0 à 9

  $1984 = 1.10^3 + 9.10^2 + 8.10^1 + 4.10^0$

- En base 2 : on a seulement 2 chiffres 0 et 1

  $1011 = 1.2^3 + 0.2^2 + 1.2^1 + 1.2^0$

  soit 11 en décimal

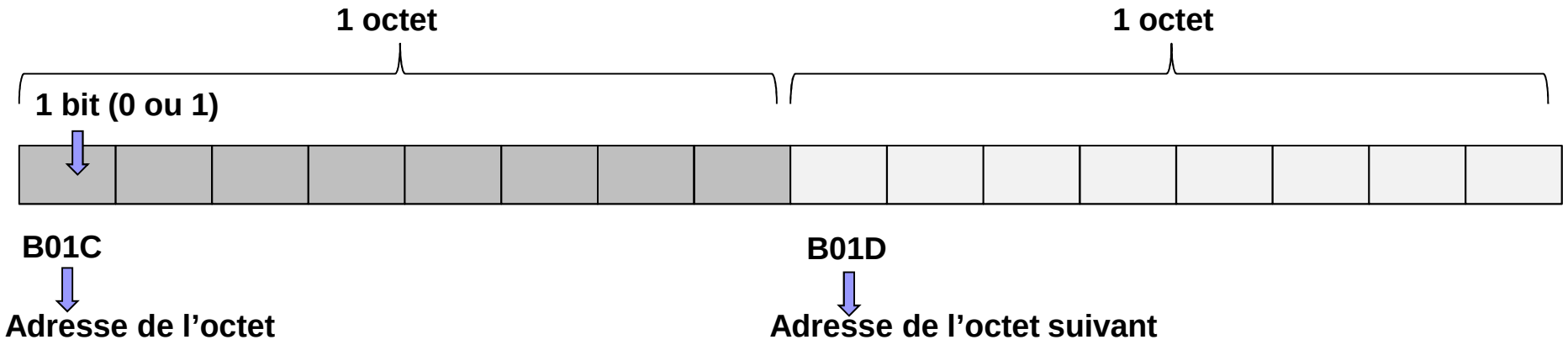
- En base 16 ou en hexadécimal : on exprime les nombres avec 10 chiffres de 0 à 9 et 6 lettres de A à F

  $10AF = 1.16^3 + 0.16^2 + 10.16^1 + 15.16^0$

  soit 4271 en décimal

Stockage et adresse

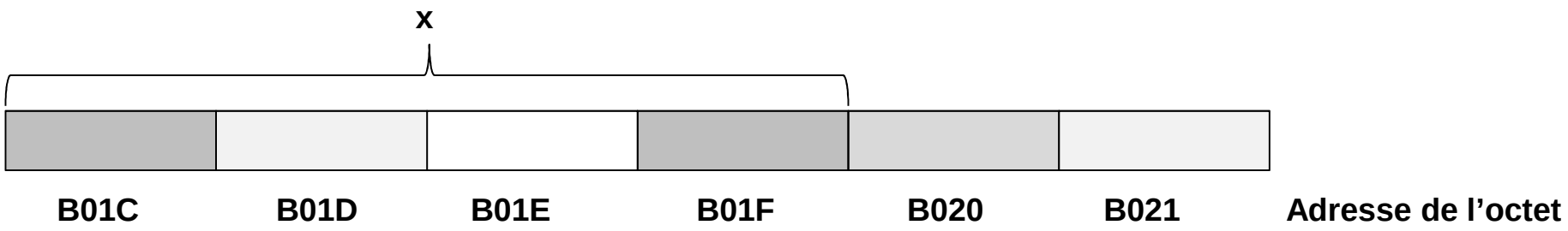
- Dans la mémoire de l'ordinateur, les données sont stockées sous forme binaire.
- La mémoire est divisée en « cases » de taille 8 bits, appelées octets (bytes en anglais).
- Chaque octet est repéré par son adresse, qui est souvent donnée par un nombre hexadécimal.



Introduction aux pointeurs

int x;

- Cette « déclaration de variable » réalise deux opérations :
  - Définition d'une variable x pouvant être utilisée dans le programme pour manipuler des données.
  - Réservation ou allocation d'un espace mémoire où sera stocké le contenu de la variable.
- La variable x est stockée sur 4 octets. Son adresse est celle du 1er octet, soit ici B01C.



Les pointeurs

int x;

- La variable x a une adresse en mémoire.
  - Dans l'exemple précédent c'est B01C.
- Pour avoir accès à l'adresse de la variable x, on utilise l'opérateur « **&** ».
- **&x** représente l'adresse de la variable **x**.
- Pour manipuler les adresses on définit un nouveau type de variable : les pointeurs.

Un pointeur est une variable qui contient l'adresse d'une autre variable. C'est une adresse typée.

Les pointeurs

- **Déclaration d'un pointeur :** double *p ;
  - **p** est un « pointeur sur une variable de type double ».
  - **p** est une adresse typée.
  - C'est l'adresse de la première case mémoire contenant la donnée. On dit que **p** pointe sur une variable.
- **Utilisation = accès à la variable pointée par p :** on utilise *
  - *p représente le contenu de la variable pointée par p
- **Initialisation d'un pointeur :**
  - On peut donner l'adresse d'une variable déjà existante :

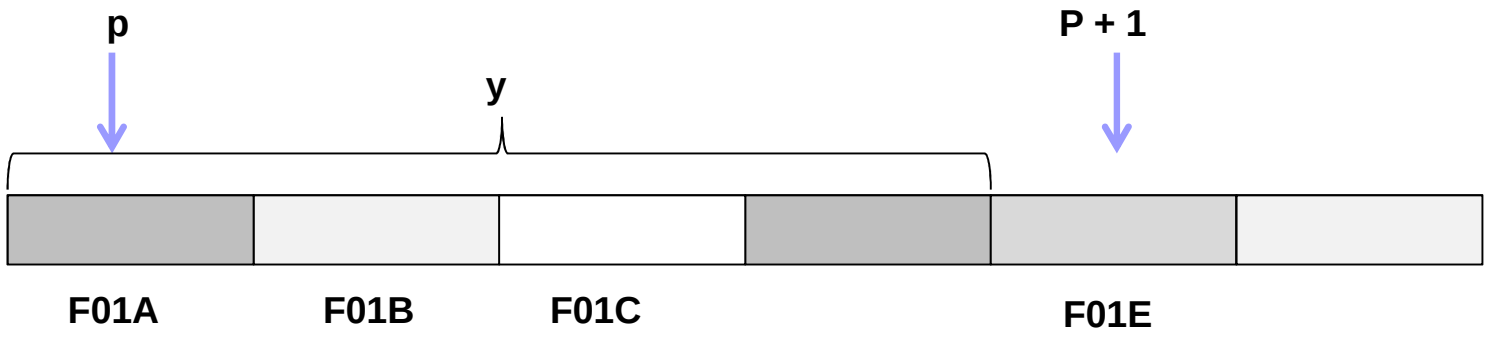
```
int y ;
int *p ;
p = &y ;
```

Les pointeurs

- **Pointer sur la case suivante :**
  - **p+1** pointe sur le double suivant en mémoire.
- **Contenu de la case mémoire suivante :**
  - ***(p+1)** est le contenu de la variable pointée par **p+1.**

```
int y ;
int *p ;
p = &y ;
```

La variable **y** est stockée sur 8 octets.
L'adresse de **y** est F01A. **p** vaut donc F01A.



Variables et pointeurs

double x ;

&x existe à la déclaration

x = 5 ;

OK

double *p ;

p n'existe pas encore à la déclaration

En mémoire, il ne s'est rien passé

***p = 5 ;**

Pas d'erreur à la compilation
(warning), mais plantage à OK

double x ;

p=&x ;

OK, initialisation correcte de p

***p = 5 ;**

Exemples

```
int n;
Exemples
int *p;
n=5;
printf("%d\n",n);
p=&n;
printf("%d\n",*p);
*p=1;
printf("%d\n",n);
printf("%d\n",*p);
printf("%x %x\n",p,&n);
printf("%x %x\n",p+1,&n+2 );
```

5

5

1

1

BC01 BC01

BC05 BC09

*p désigne le contenu de
la case mémoire pointée par p

%x est le format pour les
adresses en hexadécimal

Retour sur les fonctions

- En C, une fonction travaille sur des copies des variables.
- On parle de « *passage d'argument par valeur* »

```
void ma_fonction(int a)
{
a = 0 ;
}
main( )
{
int b = 5 ;
printf("%d",b) ;
ma_fonction(b) ;
printf("%d",b) ;
}
```

Affiche **5**

Affiche **5** : ma fonction n'a pas modifié la variable **b** définie dans le main … Pourquoi ?

Retour sur les fonctions

- En C, une fonction travaille sur des copies des variables.
- On parle de « *passage d'argument par valeur* »

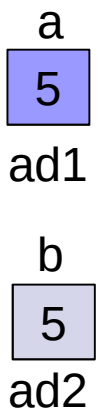
```
void ma_fonction(int a)
{
a = 0 ;
}

main( )
{
int b = 5 ;
printf("%d",b) ;
ma_fonction(b) ;
printf("%d",b) ;
}
```

2 – création d'une variable « a » à l'adresse ad1

1 – création et initialisation à 5 d'une variable « b » à l'adresse ad2

3 – recopie de la variable « b » dans la variable « a » (donc a vaut 5)



Retour sur les fonctions

- En C, une fonction travaille sur des copies des variables.
- On parle de « *passage d'argument par valeur* »

```
void ma_fonction(int a)
{
a = 0 ;
}
main( )
{
int b = 5 ;
printf("%d",b) ;
ma_fonction(b) ;
printf("%d",b) ;
}
```

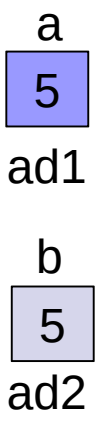
2 – création d'une variable « a » à l'adresse ad1

4 – a vaut 0

1 – création et initialisation à 5 d'une variable « b » à l'adresse ad2

3 – recopie de la variable « b » dans la variable « a » (donc a vaut 5)

5 – il ne s'est rien passé à l'adresse de la variable « b » qui n'est donc pas modifiée !



Passage d'arguments par adresse

- Un des intérêts des pointeurs : modifier la valeur de variables dont l'adresse est passée comme argument d'une fonction.
  - Exemple : échanger 2 variables

```
#include<stdio.h>
void echange(int x, int y)
{
int temp;
temp = y;
y = x;
x = temp;
}
main()
{
int a = 3,b = 25;
echange(a, b);
printf("%d\t%d\n", a, b);
}
```

Idée intuitive , mais qui ne marche pas !

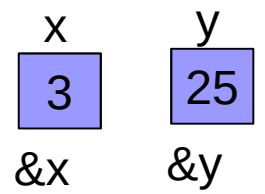
Passage d'arguments par adresse

- Un des intérêts des pointeurs : modifier la valeur de variables dont l'adresse est passée comme argument d'une fonction.
  - Exemple : échanger 2 variables

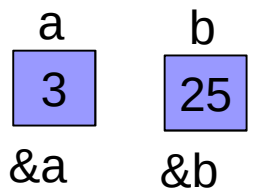
```
#include<stdio.h>
void echange(int x, int y)
{
int temp;
temp = y;
y = x;
x = temp;
}
main()
{
int a = 3, b = 25;
echange(a, b);
printf("%d\t%d\n", a, b);
}
```

2 – création de 2 variables x et y

3 – recopie de a et b dans x et y



1 – création et initialisation de 2 variables a et b



Passage d'arguments par adresse

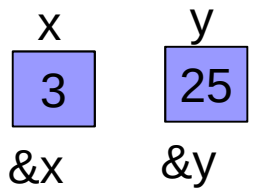
- Un des intérêts des pointeurs : modifier la valeur de variables dont l'adresse est passée comme argument d'une fonction.
  - Exemple : échanger 2 variables

```
#include<stdio.h>
void echange(int x, int y)
{
int temp;
temp = y;
y = x;
x = temp;
}
main()
{
int a = 3, b = 25;
echange(a, b);
printf("%d\t%d\n", a, b);
}
```

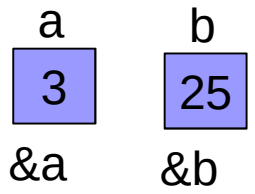
2 – création de 2 variables x et y

3 – recopie de a et b dans x et y

4 – Echange des valeurs de x et y



1 – création et initialisation de 2 variables a et b



⇨**a et b n'ont pas été modifiées !**

Passage d'arguments par adresse

- Un des intérêts des pointeurs : modifier la valeur de variables dont l'adresse est passée comme argument d'une fonction.
  - Exemple : échanger 2 variables

```
#include<stdio.h>
void echange(int *x, int *y)
{
int temp;
temp = *y;
*y = *x;
*x = temp;
}
main()
{
int a = 3,b = 25;
echange(&a, &b);
printf("%d\t%d\n", a, b);
}
```

Bonne méthode !

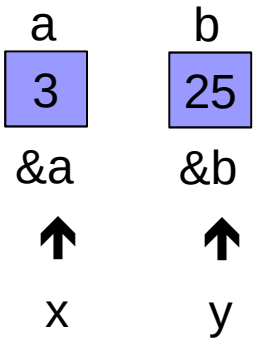
Passage d'arguments par adresse

- Un des intérêts des pointeurs : modifier la valeur de variables dont l'adresse est passée comme argument d'une fonction.
  - Exemple : échanger 2 variables

```
#include<stdio.h>
void echange(int *x, int *y)
{
int temp;
temp = *y;
*y = *x;
*x = temp;
}
main()
{
int a = 3,b = 25;
echange(&a, &b);
printf("%d\t%d\n", a, b);
}
```

2 – création de 2 pointeurs x et y initialisés à &a et &b
« x pointe sur a » et « y pointe sur b »

3 – l'échange se fait bien à &a et &b



1 – création et initialisation de 2 variables a et b

⇨**a et b ont bien été modifiées !**

Exemple : la fonction *triple*

- **Le code :**

a = 3*a ;

est codé dans une

fonction *triple(a)*

Le code est :

```
int triple(int a)
{
a = 3*a ;
return (a) ;
}
```

Et l'appel a = triple(a)

modifie bien a !

- **Le code :**

a = 3*a ; b = 3*b ; c=3*c ;

est codé dans une fonction

triple(a,b,c)

Le code est :

```
int triple(int a,int b,int c)
{
a = 3*a ; b = 3*b ; c=3*c ;
return (a) ;
}
```

Et l'appel : a = triple(a,b,c)

modifie a mais ne modifie ni b, ni c !

Exemple : la fonction *triple*

- **Le nouveau code pour modifier a, b et c simultanément est :**

```
void triple(int *a, int *b, int *c)
{
*a = 3*(*a) ; *b = 3*(*b) ; *c = 3*(*c);
}
```

Et l'appel :

triple(&a, &b, &c) **modifie bien les variables a, b et c**

Conclusion : une fonction peut modifier plusieurs variables grâce aux pointeurs

Tableaux et pointeurs

int tab[100];

- Cette déclaration fait deux choses :
  - **Définition d'une variable** pouvant être utilisée dans le programme pour manipuler des données.
  - **Allocation d'un espace mémoire** de 100 cases contigües de 4 octets.
- En fait, **un tableau n'est rien d'autre qu'un pointeur**
- La variable tab est un **pointeur** qui est l'adresse du 1er élément du tableau. En d'autres termes, le nom d'un tableau est l'adresse de son 1er élément.

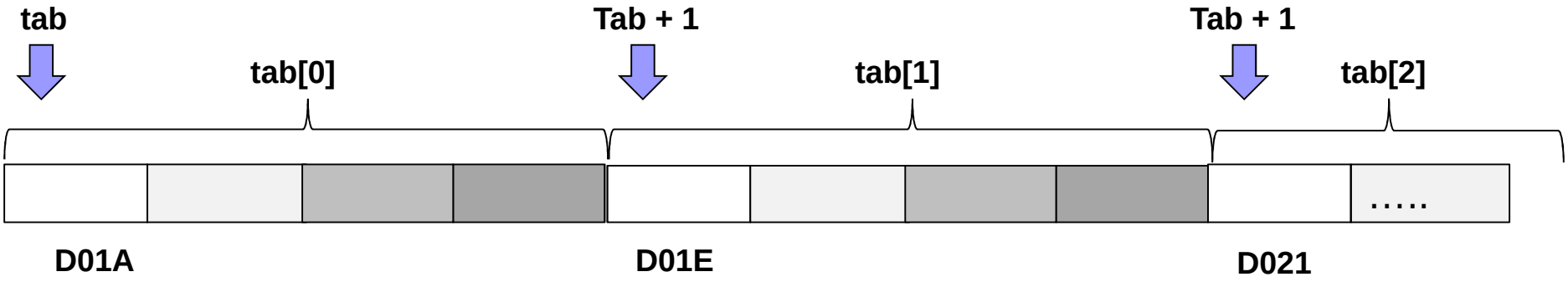
Tableaux et pointeurs

int tab[100];

- **tab** est l'adresse de **tab[0]**
- **tab+i** est l'adresse de **tab[i]**

  Il y a équivalence entre **tab+i** et **&tab[i]**

  et entre ***(tab+i)** et **tab[i]**



Exemple

```
int tab[100];
for(i = 0; i < n; i++) tab[i] = i;
printf("%n", tab[0]);
printf("%d\n", *tab);
printf("%d %d\n", *(tab+1), tab[1]);
printf("%x %x\n", tab, &tab[0];
```

⇨ Affiche les mêmes valeurs

⇨Même valeur !

⇨ Affiche la même adresse !

Tableaux et pointeurs dans les fonctions

- Il y a équivalence entre :

```
void initialise_tableau(int tab[ ], int dim);

main()
{
int T[100] ;
initialise_tableau(T, 100) ;
}
void initialise_tableau(int tab[ ], int dim)
{
int i ;
for (i=0 ; i<dim ; i++) tab[i] = 0;
}
```

Tableaux et pointeurs dans les fonctions

- et :

```
void initialise_tableau(int *tab, int dim);

main()
{
int t[100] ;
initialise_tableau(t, 100) ;
}
void initialise_tableau(int *tab, int dim)
{
int i ;
for (i=0 ; i<dim ; i++) tab[i] = 0;
}
```

En résumé : Pour bien concevoir une fonction utilisant les pointeurs

- **Déterminer ce que doit retourner la fonction :**
  - Souvent **void** (la fonction a déjà modifié les valeurs)
  - Parfois un type standard (**int**, **double**, …)
  - Certaines fonctions retournent des adresses donc de type pointeur sur quelque chose : **int*** , **double***, ...
- **Typer la fonction en conséquence**
- **Identifier ce qu'on passe à la fonction :**
  - Des valeurs ? → **fonction(int a, double b)**
  - Des adresses ? → **fonction(int *a, double *b)**