

Chapitre 9

Pointeurs

Les pointeurs

Concept

Un pointeur **est une variable qui contient une adresse mémoire**

Permet de **manipuler directement la mémoire**

Utile dans de nombreux scénarios

Un des plus **importants concepts du langage C**

Les pointeurs

Utilité

Passage des paramètres d'une fonction par adresse

Gestion de certains types (*e.g.* tableaux, chaînes de caractères, pointeurs sur fonctions)

Structures dynamiques

Performance ► éviter les copies de données

Désavantage

⚠ La mémoire doit toujours être utilisée avec grande attention

9. Table des matières

- 1. Accès mémoire direct et indirect**
2. Pointeurs
 1. Adresse d'une variable
 2. Déclaration de pointeur
 3. Opérateur de déréférencement
3. Paramètres formels d'une fonction
4. Tableaux et pointeurs

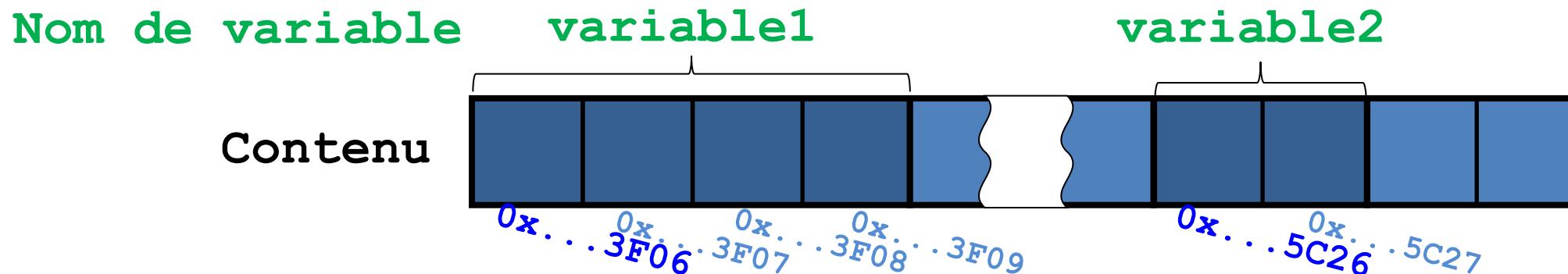
Accès à la mémoire – 2 modes d'adressage

Adressage direct

Accès au contenu d'une variable par le nom de la variable

Adressage indirect

Accès au contenu d'une variable, par **l'adresse de la variable** ou **par un pointeur** (variable qui contient l'adresse)



Accès à la mémoire – 2 modes d'adressage

Adressage direct par le nom de la variable

```
x = 40;  
printf("%d", x);
```

Adressage indirect par *, opérateur d'indirection ou de déréférencement (*contenu de*)

```
*address64 = 40;  
printf("%d", *address64);
```

Avec **address64** =
0x...3F06

Nom de variable

x

Contenu

40



0x...3F06

9. Table des matières

1. Accès mémoire direct et indirect

2. Pointeurs

1. Adresse d'une variable

2. Déclaration de pointeur

3. Opérateur de déréférencement

3. Paramètres formels d'une fonction

4. Tableaux et pointeurs

Les pointeurs

Définition simple

Les pointeurs sont des variables qui mémorisent les adresses physiques de la mémoire.

Ils donnent accès à un emplacement de la mémoire interne de l'ordinateur

Une **variable** permet d'accéder à un emplacement mémoire
variable=5



Un **pointeur** permet d'accéder à n'importe quel emplacement mémoire



Les pointeurs

Définition

- a) Un pointeur est une **variable**
- b) Un pointeur est une **variable qui contient une adresse**
- c) Un pointeur **peut contenir l'adresse d'une autre variable**

Si un pointeur P contient l'adresse d'une variable A, on dit que
«P pointe sur A»

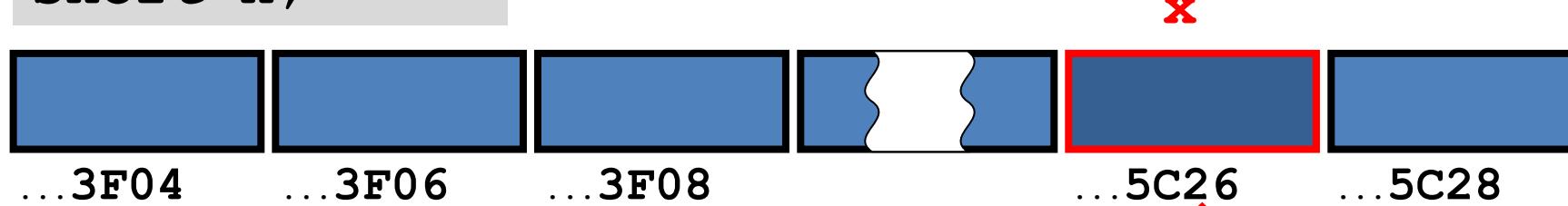
9. Table des matières

1. Accès mémoire direct et indirect
2. Pointeurs
 - 1. Adresse d'une variable**
 2. Déclaration de pointeur
 3. Opérateur de déréférencement
3. Paramètres formels d'une fonction
4. Tableaux et pointeurs

Adresse d'une variable

Déclaration de variable

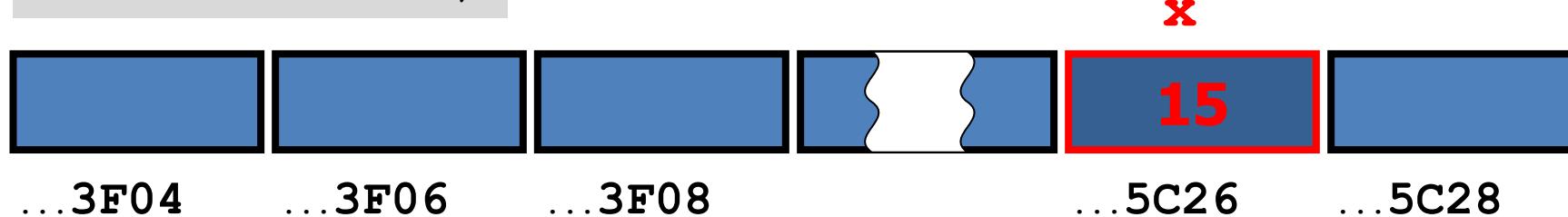
```
short x;
```



short sur 2 octets soient 16 bits

Initialisation

```
short x = 15;
```



Adresse d'une variable

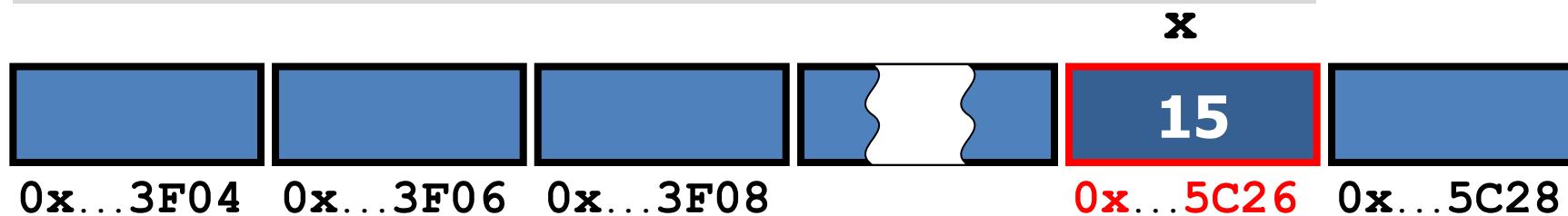
L'adresse de la variable x est obtenue par l'opérateur &

&x

→ **adresse de la variable x**

Exemple

`printf("%p", &x); → 0x 00FA E201 0223 5C26`



9. Table des matières

1. Accès mémoire direct et indirect
2. Pointeurs
 1. Adresse d'une variable
 - 2. Déclaration de pointeur**
 3. Opérateur de déréférencement
3. Paramètres formels d'une fonction
4. Tableaux et pointeurs

Syntaxe de déclaration d'un pointeur

```
<type> *<variable1>, *<variable2>, ... ;
```

Le type correspond au type de la case mémoire pointée.

Pour différencier un pointeur d'une variable ordinaire, **on fait précéder son nom du signe '*' lors de sa déclaration**

Exemple `double *ptrD1, *ptrD2;`

→ on déclare 2 variables `ptrD1` et `ptrD2`, dont le contenu est une adresse (*) pointant sur un `double`

Déclaration d'un pointeur

short *ptrX;

ptrX

Réserve un emplacement pour stocker une adresse mémoire, pointant sur un entier de type **short**

x



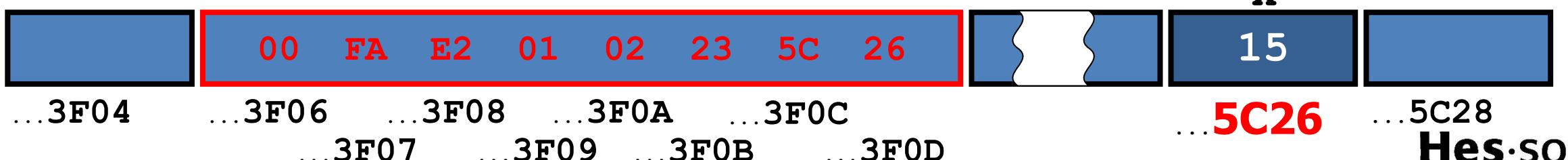
Adresse sur 8 bytes. Exemple : **0x 00FA E201 0223 5C26**

ptrX = &x;

ptrX

Écrit **l'adresse de x** dans la variable **ptrX**

On dit alors que **ptrX pointe sur x**



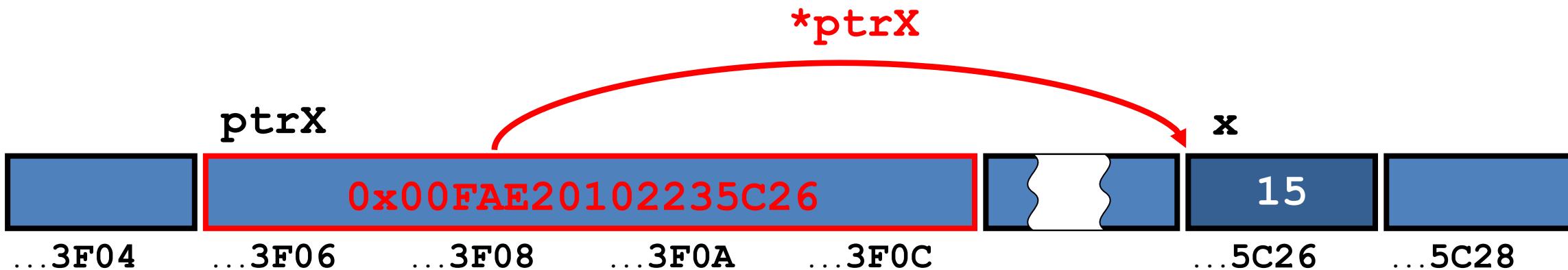
9. Table des matières

1. Accès mémoire direct et indirect
2. Pointeurs
 1. Adresse d'une variable
 2. Déclaration de pointeur
- 3. Opérateur de déréférencement**
3. Paramètres formels d'une fonction
4. Tableaux et pointeurs

Opérateur de déréférencement *

Affiche la valeur de **x** par pointeur déréférencé

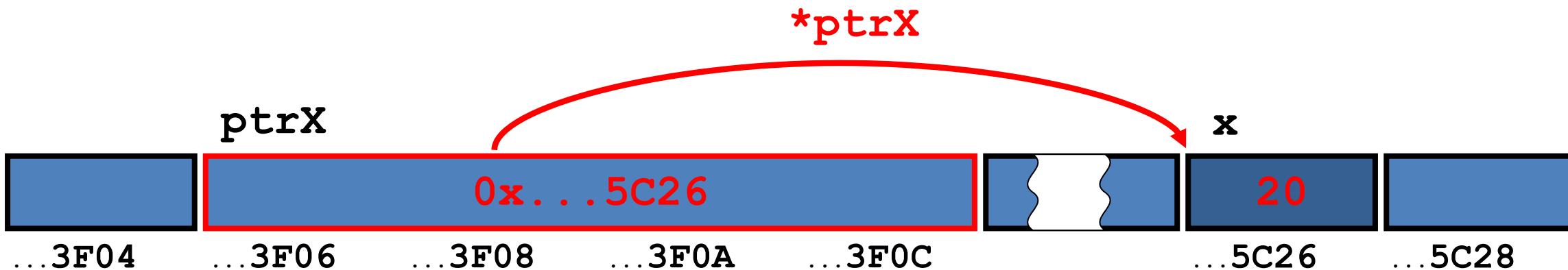
```
printf("%d", *ptrX); → 15
```



Opérateur de déréférencement *

Affecte une valeur à **x** par pointeur déréférencé

```
*ptrX = 20;
```



Remarques

Les pointeurs et les noms de variables donnent accès à un emplacement de la mémoire. **Mais :**

un pointeur est une variable qui peut 'pointer' sur **différentes adresses**

le nom d'une variable reste toujours lié à **la même adresse**

⚠ Ne pas confondre l'opérateur * de déréférencement avec la syntaxe déclarative

```
char *ptrP;
```

```
...
```

```
*ptrP = 'f';
```

Déclaration de pointeur ***** : la variable **ptrP** contient l'adresse d'un **char**

Opérateur de déréférencement ***** : accès au contenu de la mémoire pointée par **ptrP**

Résumé : les opérations sur les pointeurs

L'utilisation de pointeurs nécessite :

<type>

*ptrX, *ptrY;

une **syntaxe de déclaration** pour pouvoir
déclarer un pointeur

&_x

un opérateur unaire pour obtenir **l'adresse d'une variable**. L'**opérateur «adresse de» &**

*ptrX

un opérateur unaire pour **accéder au contenu** de la mémoire, par son adresse. L'**opérateur de déréférencement *** (**«contenu de»**)

Opérateur	Sens d'évaluation	Priorité
<code>++ -- (postfixé) ()</code>	→	haute
<code>& (adresse) * (déref.) + - (unaire) ++ --</code> <code>(préfixé) (<Type>) ! ~</code>	←	
<code>* / %</code>	→	
<code>+ -</code>	→	
<code><< >></code>	→	
<code>< > <= <=</code>	→	
<code>== !=</code>	→	
<code>&</code>	→	
<code>^</code>	→	
<code> </code>	→	
<code>&&</code>	→	
<code> </code>	→	
<code>? :</code>	←	
<code>= += -= *= /= %= >>= etc</code>	←	basse

Exemples

`ptrX = &x;`

Si un pointeur `ptrX` pointe sur une variable `x` alors `*ptrX` peut être utilisé partout où on peut écrire `x`

Les expressions suivantes sont donc équivalentes

<code>y = *ptrX+1</code>	\equiv	<code>y = x+1</code>
<code>*ptrX = *ptrX+10</code>	\equiv	<code>x = x+10</code>
<code>*ptrX += 2</code>	\equiv	<code>x += 2</code>
<code>++*ptrX</code>	\equiv	<code>++x</code>
<code>(*ptrX)++</code>	\equiv	<code>x++</code>

Pointeurs particuliers

```
int *ptrP = NULL;
```

La constante `NULL` est utilisée pour indiquer que `ptrP` ne contient pas encore d'adresse valide.

Bonne pratique : initialiser tous les pointeurs à `NULL`

Le type `void*` est utilisé quand on ne sait pas encore le type sur lequel va pointer le pointeur → type retourné par `malloc`

```
void *ptrP;
```

Vue d'ensemble

```
int a = 6;
```

Constante

6 est une **constante** qui est déterminée lors de la **compilation** ; elle n'est **pas modifiable** par le programme

```
int a;  
a = 6;
```

Variable

L'adresse de a est une **constante** qui est déterminée lors de la **compilation**. Le contenu de a, sa valeur, est **modifiable** par le programme

```
int *ptr = NULL;  
ptr = &a
```

Pointeur

L'adresse de ptr est une **constante** qui est déterminée lors de la **compilation** ; le contenu de ptr, l'adresse de la variable a, est **modifiable** par le programme

9. Table des matières

1. Accès mémoire direct et indirect

2. Pointeurs

 1. Adresse d'une variable

 2. Déclaration de pointeur

 3. Opérateur de déréférencement

3. Paramètres formels d'une fonction

4. Tableaux et pointeurs

Passage de paramètres

En programmation, il existe 2 modes de passage de paramètres

Passage par valeur

en entrée : le compilateur passe la valeur à la fonction

Passage par adresse

en entrée/sortie : le compilateur passe l'adresse de la variable

→ la fonction peut lire et modifier la valeur de la variable

En C, le passage de paramètres ne se fait que par valeur !

Passage de paramètres

Principe

1. La fonction appelante fait une **copie des valeurs passées en paramètre** dans la pile (`«stack»`)
2. La fonction appelée utilisera ces copies comme paramètres
3. Ces copies disparaîtront lors du retour à la fonction appelante

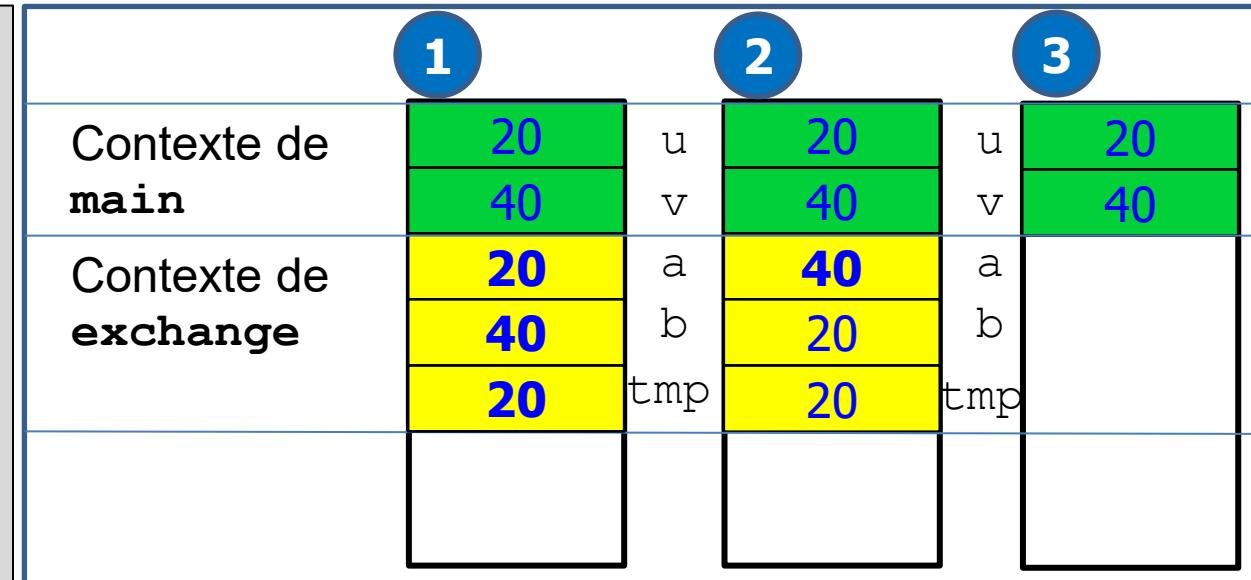
Passage par valeur

```

void exchange(int a,int b)
{
    int tmp = a;
    1 a = b;
    2 b = tmp;

    int u = 20, v = 40;
    printf("u = %d v = %d\n",u,v);
    exchange(u, v);
    3 printf("u = %d v = %d\n",u,v);
    return 0;
}

```



u=20 v=40

u=20 v=40

**Les valeurs u et v
ne sont
PAS échangées !**

Comment échanger, avec une fonction en langage C, les valeurs de 2 variables ?

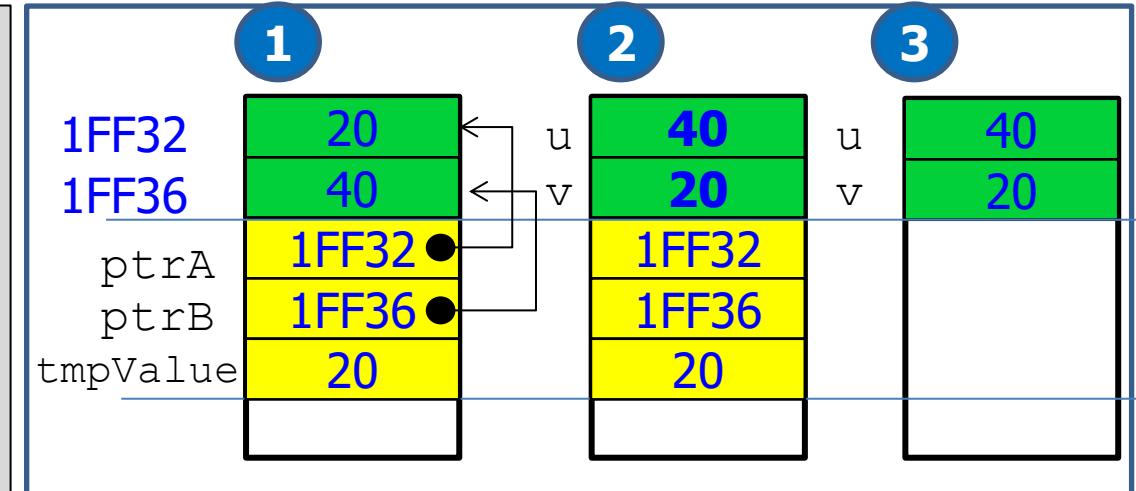
Puisque, le seul mode de passage des paramètres (en C) se fait par copie, pour modifier a ou b depuis la fonction **exchange**, on doit accéder à leur valeur de manière indirecte, avec un pointeur !

Passage par valeur

```
void exchange(int *ptrA, int *ptrB)
{
    int tmpValue = *ptrA;
    1 *ptrA = *ptrB;
    2 *ptrB = tmpValue ;
}

int main(void)
{
    int u = 20, v = 40;
    printf("u = %d v = %d\n", u, v);
    exchange(&u, &v);
    3 printf("u = %d v = %d\n", u, v);
    return 0;
}
```

1FF32 1FF36



u:20 v:40

Les valeurs u et v sont maintenant échangées !

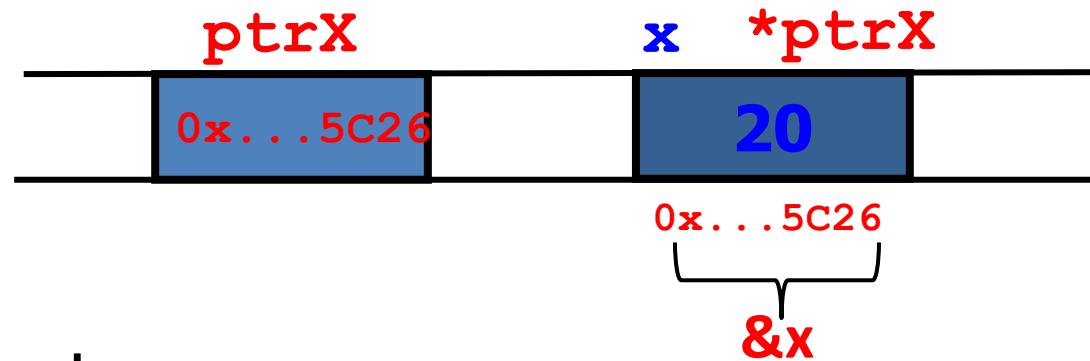
9. Table des matières

1. Accès mémoire direct et indirect
2. Pointeurs
 1. Adresse d'une variable
 2. Déclaration de pointeur
 3. Opérateur de déréférencement
3. Paramètres formels d'une fonction
- 4. Tableaux et pointeurs**

9.4 Tableaux et pointeurs

Résumé

```
int x=20;  
int *ptrX;  
ptrX = &x;
```



x désigne le **contenu** de **x**

&x désigne **l'adresse** de **x**

ptrX contient **l'adresse** de **x** → **ptrX** pointe sur **x**

***ptrX** désigne le **contenu** de **x**

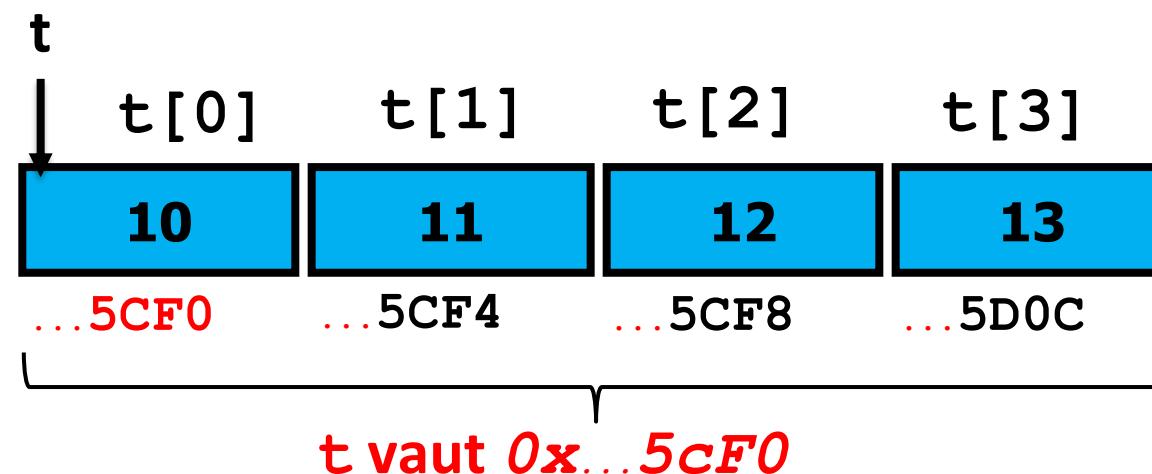
9.4 Lien entre pointeur et tableau

Le nom d'un tableau, ici *t*, représente en réalité son adresse !

```
int t[4];
printf("%p" ,t); → 0x00000000000000005CF0
```

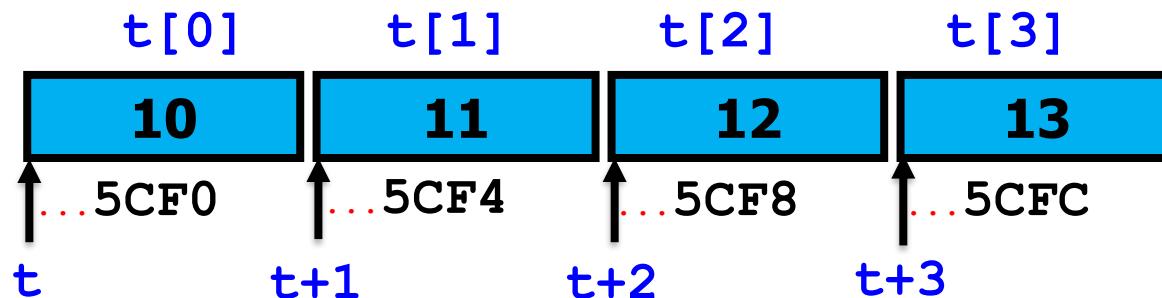
%p → format pour afficher une adresse en hexadécimal

⚠ dépend du compilateur et de la plateforme



9.4 Lien entre pointeur et tableau

```
int t[4];
printf("%p", t); → 0x...5CF0
printf("%p", t+1); → 0x...5CF4
printf("%p", t+2); → 0x...5CF8
printf("%p", t+3); → 0x...5CFC
```



`t[0]` est équivalent `*t`
`t[1]` est équivalent `* (t+1)`
`t[2]` est équivalent `* (t+2)`
`t[3]` est équivalent `* (t+3)`

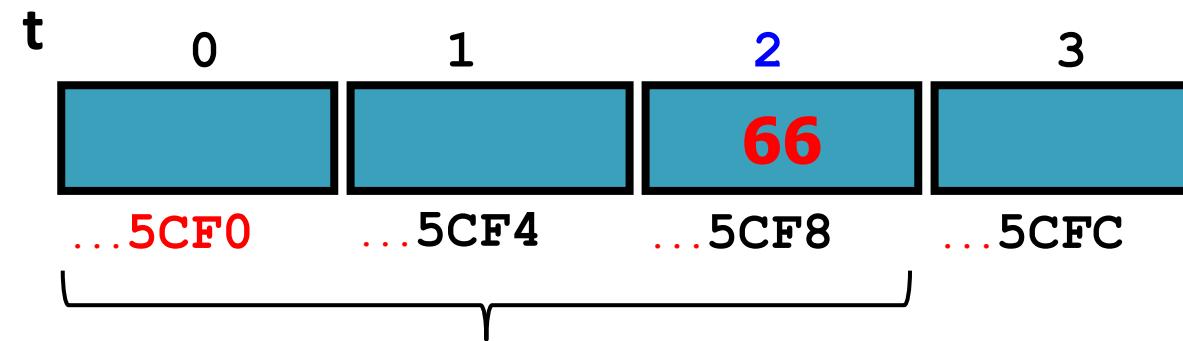
9.4 Lien entre pointeur et tableau

Formalisme tableau vs formalisme pointeur

$t[i]$ est équivalent $* (t+i)$

Exemple

```
int t[4]; t[2] = 66
```



$t+2 \rightarrow 5CF0 + 2 * \text{sizeof}(int) = 5CF8$
 $* (t+2) \rightarrow * (5CF8) \rightarrow 66$

Différence entre tableau et pointeur



```
int t[10];  
  
int *ptr;  
....  
ptr = t;
```

A red arrow points from the word "constante" to the identifier "t". A blue arrow points from the word "variable" to the identifier "ptr".

Un pointeur est une **variable**, donc des opérations comme **ptr=t;** ou **ptr++** sont **permises**.

Le nom d'un tableau est une **constante**, donc des opérations comme **t=ptr** ou **t++** sont **impossibles**

Différence entre tableau et pointeur

```
int t[10]; // sizeof(t) → 40
```

Un tableau est constitué d'un **pointeur constant**, qui pointe sur le premier élément, et d'une **zone mémoire** pour stocker ses données

```
int *ptr = t; // sizeof(ptr) → 4 / 8
```

Un **pointeur** est une variable qui contient une **adresse**

Formalisme tableau et formalisme pointeur

```
int main(void)
{
    int t[10] = {-3, 4, 0, -7, ...};
    int pos[10];
    int i, j=0;
    for (i=0; i<10; i++)
        if (t[i] > 0)
    {
        pos[j] = t[i];
        j++;
    }
    return 0;
}
```

```
int main(void)
{
    int t[10] = {-3, 4, 0, -7, ...};
    int pos[10];
    int i, j=0;
    for (i=0; i<10; i++)
        if (*(t+i) > 0)
    {
        *(pos+j) = *(t+i);
        j++;
    }
    return 0;
}
```

Exemple d'implémentation de strcpy()

```
void mystrcpy(char *ptrCh1,  
              const char *ptrCh2)  
{  
    int i;  
    i=0;  
    while ((ptrCh1[i] = ptrCh2[i]) != '\0')  
        i++;  
}
```

Exemple d'implémentation de strcpy()

```
void mystrcpy(char *ptrCh1,  
              const char *ptrCh2)  
{  
    int i;  
    i=0;  
    while( (* (ptrCh1+i) = * (ptrCh2+i) ) != '\0')  
        i++;  
}
```

Exemple d'implémentation de strcpy()

```
void mystrcpy(char *ptrCh1,  
              const char *ptrCh2)  
{  
    while ( (*ptrCh1 = *ptrCh2) != '\0' )  
    {  
        ptrCh1++;  
        ptrCh2++;  
    }  
}
```

Exemple d'implémentation de strcpy()

```
void mystrcpy(char *ptrCh1,  
              const char *ptrCh2)  
{  
    while (*ptrCh1++ = *ptrCh2++) ;  
}
```

Programmer avec des pointeurs : résumé

Déclaration

```
int var, tab[5];  
int *ptrT=NULL;  
int *ptrV=NULL;
```

Initialisation

```
ptrV = &var;  
ptrT = tab;
```

Utilisation

```
*ptrV = 6;  
*(ptrT+1)=4;  
*ptrT = *ptrV;
```

Fonction avec un argument de type pointeur

Déclaration

```
void foo(int *ptr);
```

appel

```
foo( ptr );  
foo( &var );
```

Résumé : le pointeur

Un pointeur

- 1. est une variable**
- 2. contient une adresse mémoire**
- 3. est associé à une arithmétique**

Exercices



Exercices du chapitre 09