Eléments d'Informatique

Cours 9 – Pointeurs et tableaux

Catherine Recanati

UNIVERSITÉ PARIS 13 NORD

Plan général

- **Représentation des nombres. Notion de variable.**
- **Programme. Expressions.**
- **Architecture des ordinateurs: langage machine, langage assembleur, AMIL.**
- **Systèmes d'exploitation : fichiers, processus, compilation.**
- **Instructions de contrôle: boucles et branchements.**
- **Programme. Définition de fonction. Appel fonctionnel.**
- **Tableaux de variables et fonctions d'arguments de type tableau.**
- **Sens d'un programme, pile d'exécution, compilation.**
- **Pointeurs et tableaux.**
- Chaines de caractères, bibliothèque <string.h>.
- Allocation dynamique, liste chaînées.
- Révisions.

- Cours 9 – Pointeurs et tableaux

- Pointeurs comme adresses
- opérateur & et opérateur *
- passage de variable par référence (par adresse)
- Echange des valeurs de deux variables
- Fonctions à argument de type tableau
- Parcours de tableaux avec des pointeurs
- Tri des valeurs d'un tableau

Rappel: dans un programme écrit en langage C

Une variable `x` désigne une valeur

Cette valeur est mémorisée
(c'est-à-dire stockée)
à l'adresse de la variable

Adresse d'une variable x

```
int x;          // déclaration d'une variable x
                // de type int

x = 3 ;         // affectation de la valeur 3
                // à la variable x
```


```
printf("x :  %d", x );   // imprime 3
printf("&x : %p", &x);   // imprime
                         // 0x7fff53edeb8c
```

Vocabulaire

- Un *pointeur* est une variable dont la valeur est une adresse (en général celle d'une autre variable).
- si p est une variable dont la valeur est l'adresse d'une autre variable x, on dit que p est *un pointeur sur x* ou que p *pointe sur* x.

L'opérateur d'adresse &

Placé devant une variable,
il désigne l'adresse en mémoire
de la variable

On l'appelle *opérateur d'adresse*
(ou opérateur de dé-référence)

Il se prononce « et commercial »
ou « esperluette »

L'opérateur étoile *

C'est l'opérateur inverse
de l'opérateur d'adresse.

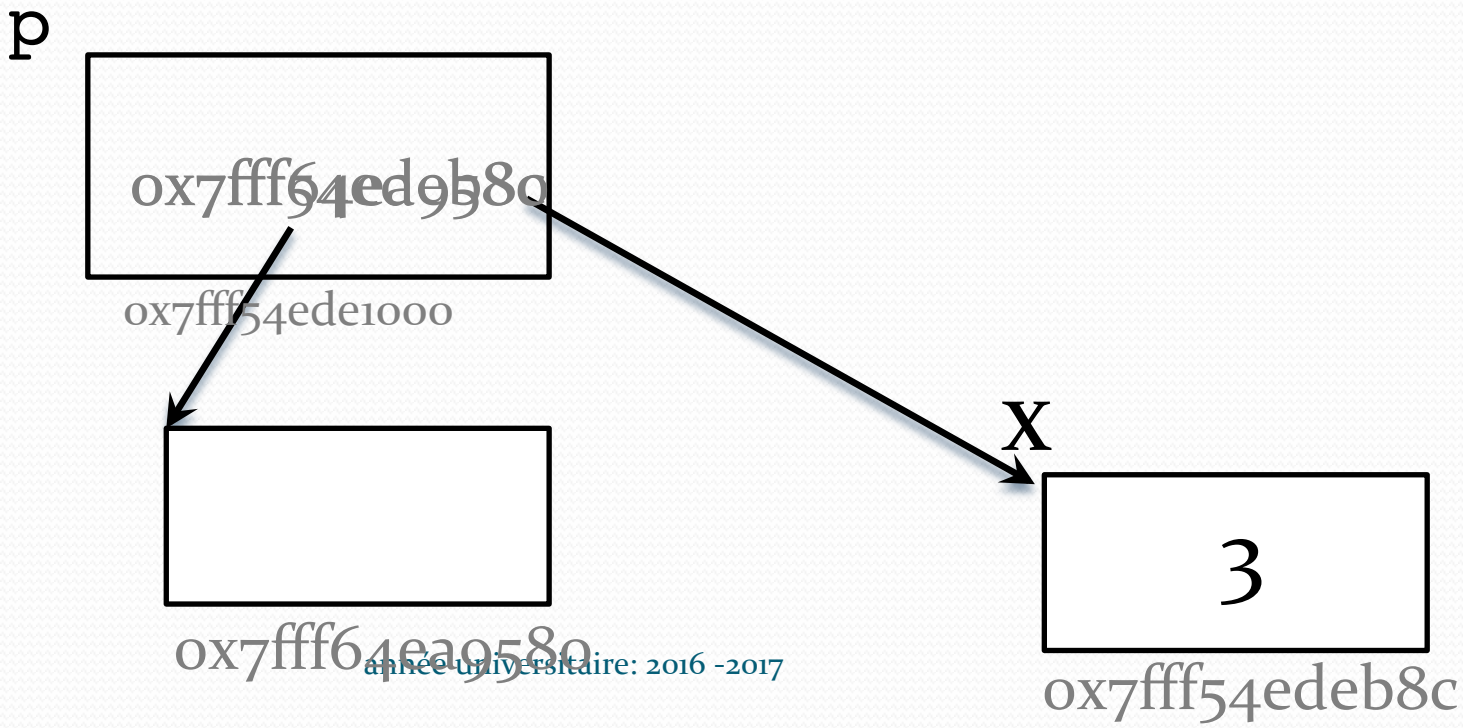
On a x = *(&x)

Appliqué à une adresse,
il désigne la valeur située à cette adresse
(c.à.d. le contenu de cette mémoire)

Appliqué à un pointeur il désigne
la valeur pointée par ce pointeur

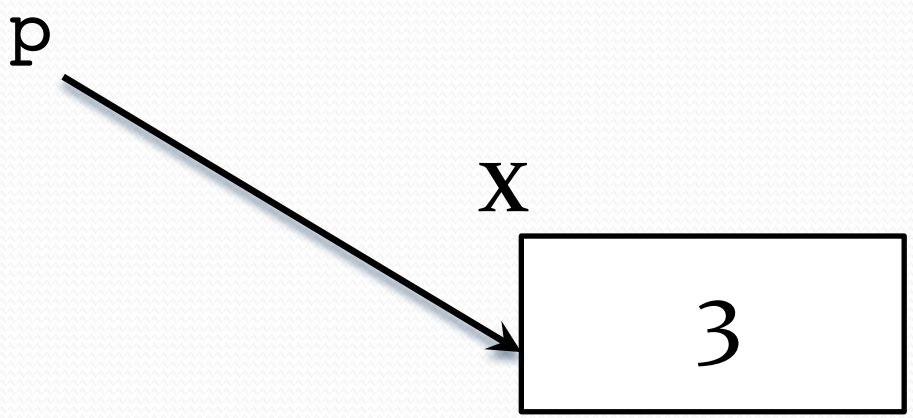
```
int x = 3;   // déclaration d'une variable x de type int
             //   initialisée à la valeur 3
int* p ;     // déclaration d'un pointeur p sur un int
             // on peut aussi écrire  int   *p;

   p = &x ;    // initialisation du pointeur p  par l'adresse de
               // la variable x.  On dit que  p  pointe sur x
```



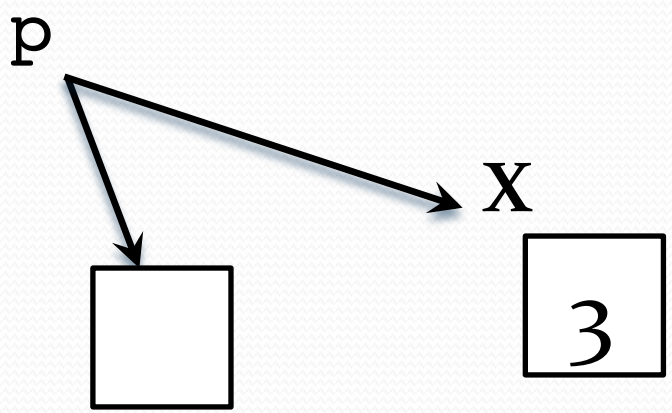
```
int x = 3;    // déclaration d'une variable x de type int
              //   initialisée à la valeur 3
int* p;       // déclaration d'un pointeur p sur un int
              // il pointe sur une valeur de type int  *p

   p = &x ;     // affectation de l'adresse de la variable x
                // au pointeur pt. On dit que  p  pointe sur x
                // et on a  *p  ==  3
```



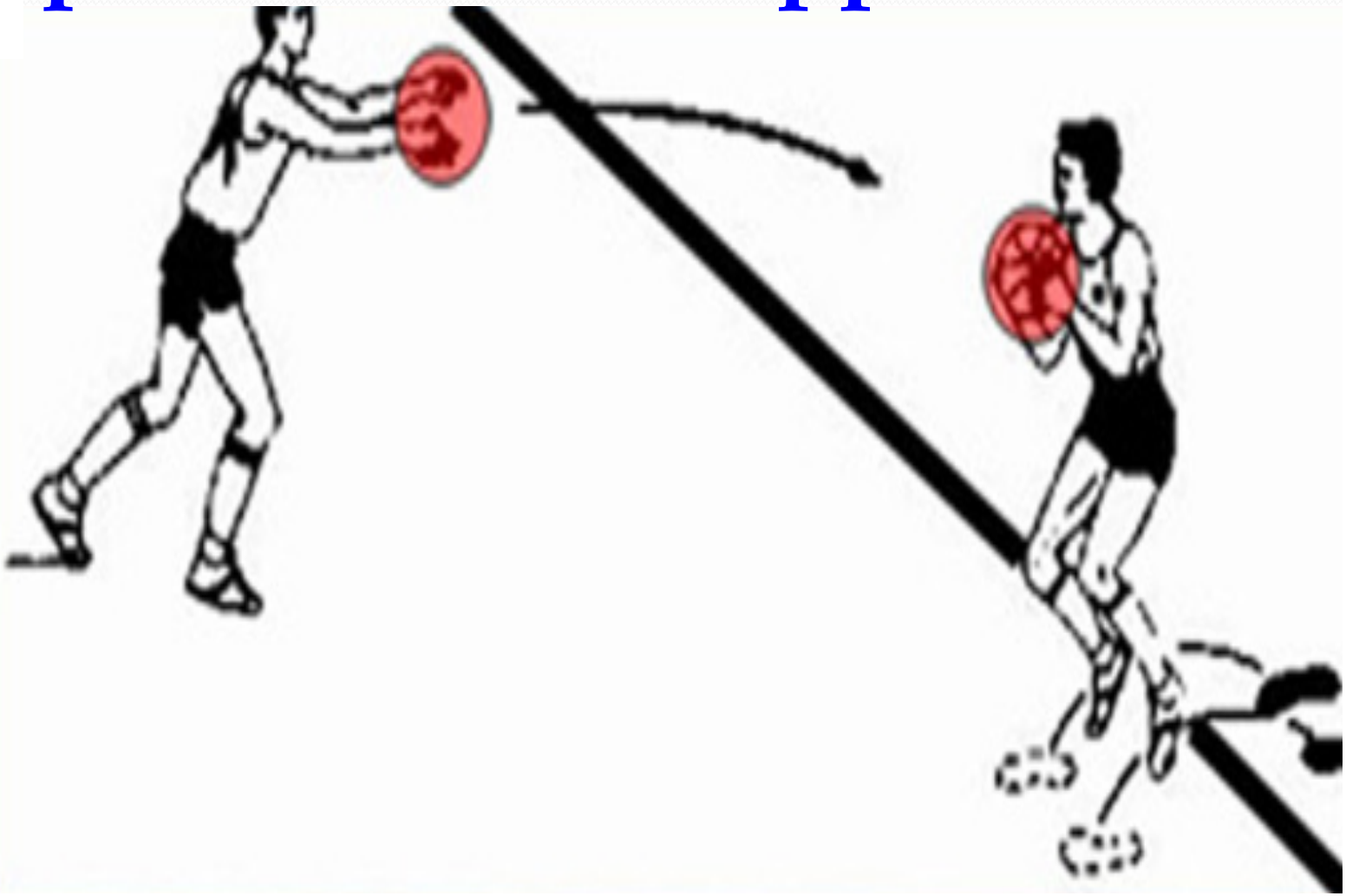
```
int x = 3;    // déclaration d'une variable x de type int
              //   initialisée à la valeur 3
int* p;       // déclaration d'un pointeur p sur un int
              // il pointe sur une valeur de type int  *p

   p = &x ;     // affectation de l'adresse de la variable x
                // au pointeur pt. On dit que  p  pointe sur x
```



Plan

Passage de variable x par référence : appel f (&x)



Plan

Lors d'un appel de fonction,
les *valeurs*
des expressions-arguments
sont transmises (i.e. affectées)
aux paramètres formels
de la fonction (avant exécution)

Lors d'un appel de `f (expression);`
avec f définie par

```
f (a){
    ... corps où figure a ...
}
```

Le paramètre formel *a* sera initialisé par la valeur de retour du calcul de *expression*, avant l'exécution du corps de la fonction.

Le passage d'une adresse de variable permet d'accéder à cette variable durant l'exécution de la fonction, et donc d'en modifier la valeur.

Les modifications effectuées pendant le calcul de la fonction subsisteront après le retour de calcul de l'appel à la fonction. On parle d'effet de bord de la fonction quand la valeur de la variable a été modifiée.

Résumé sur les appels f(x) et f(&x)

- f (x) : **passage de** la variable **x** **par valeur.**

La variable x elle-même **n'est pas modifiable** par le calcul de f.

- f (&x) : **passage de** la variable **x** **par adresse** (ou **par référence**)

La variable x peut être modifiée par le calcul de f.

Passage de variable par référence

Inconvénients :

- complexité syntaxique: opérateur &, *, etc.
- effets de bords possiblement imprévus, car les affectations sur la variable ne sont pas visibles depuis le programme principal.

Passage de variable par référence

Avantages :

- efficacité en cas de variable structurée de valeur longue à recopier (cas des tableaux, des listes, etc.)
- permet de modéliser des fonctions multiples, c'est-à-dire des fonctions qui retournent non pas une, mais plusieurs valeurs

Modèles mathématiques de fonction

- Fonction mathématique usuelle

  $y = f(x)$;

- Fonction de plusieurs arguments

  $y = f(x1, ..., xN)$;

- Fonction qui renvoie plusieurs valeurs $y1, ..., yN$

  (void) $f(x1, ..., xN, \&y1, ..., \&yN)$;

- Pour rendre visible le fait qu'une variable rapporte une valeur en retour de fonction

  > ajouter _return à son nom

Exemple :

```
int  LookupDefColor (name, &def_return)
   char *       name ;
   Color        def_return;
```

Echange des valeurs de 2 variables

Pour échanger les valeurs de deux variables `x` et `y`, il faut utiliser une troisième variable. En effet, si l'on écrit directement

```
x = y;
y = x;
```

ça ne marche pas, car la valeur de `x` aura été perdue dès la première affectation (elle est « écrasée » par la valeur de `y`).

Echange des valeurs de 2 variables

```
int x, y, sauve;


sauve = x;
x = y;
y = sauve;
```

sont des lignes de C qui permettent de programmer l'échange des valeurs de `x` et de `y`.

```
void echange(int a, int b) {
    int sauve;
    sauve = a;
    a = b;
    b = sauve;
}
int main() {
    int x=0, y=4;
    echange (x,y);
    printf( "x=%d, y=%d », x, y);
}
```

```
void echange(int *a, int *b) {
    int sauve;
    sauve = ...;
    ...
    ...   = sauve;
}
int main() {
    int x=0, y=4;
    echange (&x, &y);
    printf( "x=%d, y=%d », x, y);
}
```

```
void echange(int *a, int *b) {
    int sauve;
    sauve = *a;
    *a  = *b;
    *b = sauve;
}
int main() {
    int x=0, y=4;
    echange (&x, &y);
    printf( "x=%d, y=%d », x, y);
}
```

Fonctions à argument de type tableau

On a vu que dans le cas d'un tableau dont les valeurs des variables indexées sont modifiées par le calcul de la fonction, ces valeurs restent modifiées après le calcul de la fonction.

Les fonctions prenant en paramètre une variable de type tableau sont aussi un exemple de passage d'argument par adresse, mais cela n'est pas visible syntaxiquement (il n'y a pas de &).

Un tableau tab est un pointeur (constant) dont la valeur est l'adresse de la 1ere variable à laquelle il donne accès.

On a donc l'égalité `tab == &tab[0]` et si on cherche à afficher `tab` au format pointeur

```
printf("tab = %p", tab);
```

On obtiendra par exemple

```
tab = 0x7fff58140c20
```

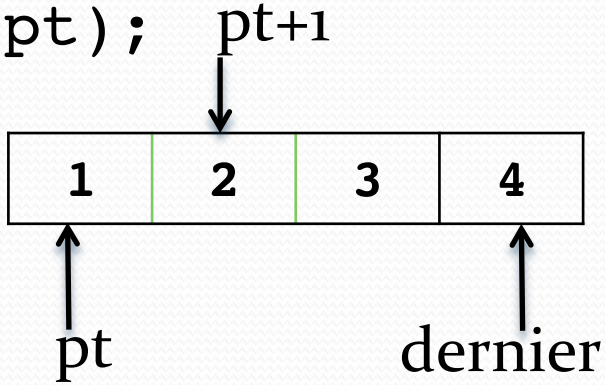
Quand on passe un tableau en argument à une fonction, on ne voit pas apparaître l'opérateur & explicitement, mais ses valeurs peuvent être modifiées.

Parcours de tableau

Les pointeurs permettent de parcourir les valeurs d'un tableau. Exemple :

```
int tab[]= {1,2,3,4};
int *pt = &tab[0]; // pt pointe sur le début
                   /* équivallent à int *pt = tab */
int *dernier = &tab[3]; // pointe sur la fin

for ( ; pt <= dernier; ) {
     printf( "%d ", *pt);
     pt = pt+1;
}
```

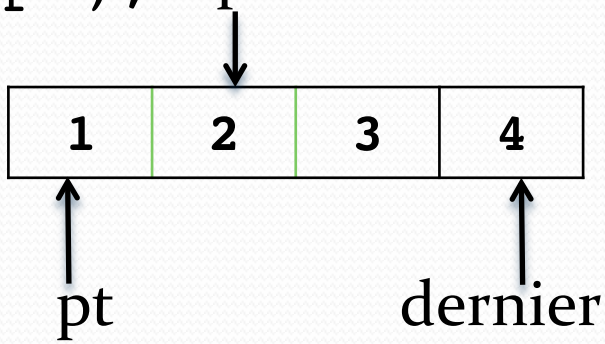


Parcours de tableau

Les pointeurs permettent de parcourir les valeurs d'un tableau. Exemple :

```
int tab[]= {1,2,3,4};
int *pt = tab; // pt pointe sur le début
int *dernier = &tab[3]; // pointe sur la fin
                // équivalent à: int dernier= tab + 3

for ( ; pt <= dernier; pt++) {
     printf( "%d ", *pt);
}
```

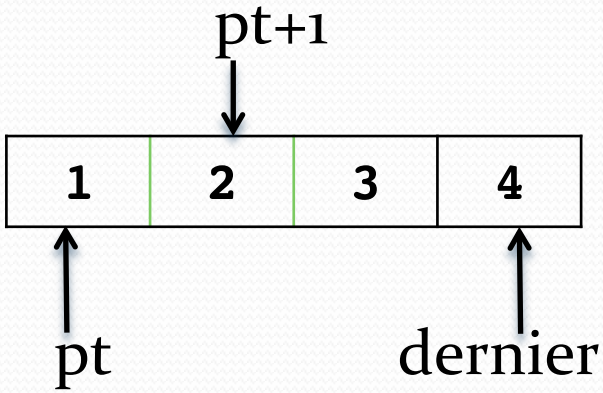


Parcours de tableau

Les pointeurs permettent de parcourir les valeurs d'un tableau. Exemple :

```
int tab[]= {1,2,3,4};
int *pt = tab; // pt pointe sur le début
int *dernier = tab +3; // pointe sur la fin

for ( ; pt <= dernier; )      {
     printf( "%d ", *pt++);
}
```



Tri (des valeurs) d'un tableau

Un algorithme possible est le suivant (ce n'est pas le seul):

Si le nb d'éléments du tableau est 1 ou 0

alors (ne rien faire) retourner void.

Sinon parcourir le tableau et pour chaque i

comparer tab[i] avec tab[j] pour j > i

(il doit être plus petit pour que le tableau soit trié)

si on trouve un j avec tab[j] < tab[i]

échangez les valeurs tab[i] et tab[j].

```
void trieTab(int tab[], int nb){
   int i, j;

   if ((nb == 0) || (nb == 1))
       return;
   for (i=0; i < nb-1; i++)
     for (j=i+1; j < nb; j++)
        if (tab[i] > tab[j])
           echange(&tab[i],
                   &tab[j]);
   return;
}
```

```
void trieTab(int tab[], int nb){
 int *p , *q;
 int *der = tab + nb-1;

  if ((nb == 0) || (nb == 1))
      return;
  for (p = tab; p <= der; p++)
    for (q = p+1; q <= der; q++)
       if (*p > *q)
          echange(p,q);
  return;
}
```

```
void trieTab(int tab[], int nb){
   int *p, *q;
   int *out= tab + nb; // en dehors

    if ((nb == 0) || (nb == 1))
        return;
   for (p = tab; p < out; p++)
     for (q = p+1; q < out; q++)
         if (*p > *q)
             echange(p,q);
   return;
}
```

Remarque : il faut passer le nombre d'éléments du tableau en paramètre, car à l'intérieur de la fonction on ne peut pas utiliser `sizeof(tab)/sizeof(tab[0])` pour connaître le nombre d'éléments du tableau passé en argument.

En effet, le paramètre formel de la fonction, déclaré sans dimension, n'a pas d'espace mémoire réservé (seule la valeur de début du tableau est passée en paramètre), et la formule précédente, serait appliquée au paramètre formel, et donnerait toujours `1`.

Plan

Merci pour votre attention !

Des questions ?