

TD 5 LES POINTEURS

d'après le site de F. Faber http://www.ltam.lu/Tutoriel_Ansi_C

Définition: Pointeur

*Un **pointeur** est une variable spéciale qui peut contenir l'adresse d'une autre variable.*

En C, chaque pointeur est limité à un type de données. Il peut contenir l'adresse d'une variable simple de ce type ou l'adresse d'une composante d'un tableau de ce type.

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

'P pointe sur A'.

Remarque

Les pointeurs et les noms de variables ont le même rôle: Ils donnent accès à un emplacement dans la mémoire interne de l'ordinateur. Il faut quand même bien faire la différence:

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

- Le **nom d'une variable** reste toujours lié à la même adresse.
-

1. Généralités sur les pointeurs

Les opérateurs de base

Lors du travail avec des pointeurs, nous avons besoin

- d'un opérateur 'adresse de': **&**
pour obtenir l'adresse d'une variable.
- d'un opérateur 'contenu de': *****
pour accéder au contenu d'une adresse.
- d'une syntaxe de déclaration
pour pouvoir déclarer un pointeur.

L'opérateur 'adresse de' : &

&<NomVariable>
fournit l'adresse de la variable <NomVariable>

L'opérateur **&** nous est déjà familier par la fonction **scanf**, qui a besoin de l'adresse de ses arguments pour pouvoir leur attribuer de nouvelles valeurs.

Exemple

```
int N;  
printf("Entrez un nombre entier : ");
```

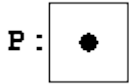
```
scanf ("%d", &N) ;
```

Attention !

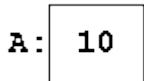
L'opérateur **&** peut seulement être appliqué à des objets qui se trouvent dans la mémoire interne, c.-à-d. à des variables et des tableaux. Il ne peut pas être appliqué à des constantes ou des expressions.

Représentation schématique

Soit P un pointeur non initialisé



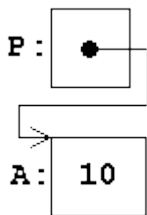
et A une variable (du même type) contenant la valeur 10 :



Alors l'instruction

```
P = &A;
```

affecte l'adresse de la variable A à la variable P. En mémoire, A et P se présentent comme dans le graphique à la fin du chapitre 9.1.2. Dans notre représentation schématique, nous pouvons illustrer le fait que 'P pointe sur A' par une flèche:



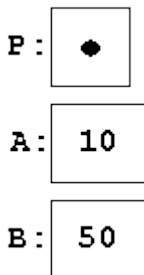
L'opérateur 'contenu de' : *

***<NomPointeur>**

désigne le contenu de l'adresse référencée par le pointeur <NomPointeur>

Exemple

Soit A une variable contenant la valeur 10, B une variable contenant la valeur 50 et P un pointeur non initialisé:

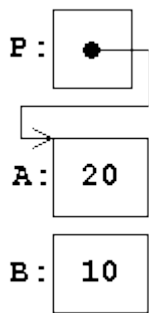


Après les instructions,

```
P = &A;  
B = *P;  
*P = 20;
```

- P pointe sur A,

- le contenu de A (référéncé par *P) est affecté à B, et
- le contenu de A (référéncé par *P) est mis à 20.



Déclaration d'un pointeur

<Type> *<NomPointeur>

déclare un pointeur <NomPointeur> qui peut recevoir des adresses de variables du type <Type>

Une déclaration comme

```
int *PNUM;
```

peut être interprétée comme suit:

*"*PNUM est du type **int**"*

ou

*"PNUM est un pointeur sur **int**"*

ou

*"PNUM peut contenir l'adresse d'une variable du type **int**"*

Exemple

Le programme complet effectuant les transformations de l'exemple ci-dessus peut se présenter comme suit:

| | | |
|--|---------|--|
| <pre> main() { /* déclarations */ short A = 10; short B = 50; short *P; /* traitement */ P = &A; B = *P; *P = 20; return 0; }</pre> | ou bien | <pre> main() { /* déclarations */ short A, B, *P; /* traitement */ A = 10; B = 50; P = &A; B = *P; *P = 20; return 0; }</pre> |
|--|---------|--|

Remarque

Lors de la déclaration d'un pointeur en C, ce pointeur est lié explicitement à un type de données.

Ainsi, la variable PNUM déclarée comme pointeur sur **int** ne peut pas recevoir l'adresse d'une variable d'un autre type que **int**.

Nous allons voir que la limitation d'un pointeur à un type de variables n'élimine pas seulement un grand nombre de sources d'erreurs très désagréables, mais permet une série d'opérations très pratiques sur les pointeurs

En travaillant avec des pointeurs, nous devons observer les règles suivantes:

Priorité de * et &

* Les opérateurs * et & ont la même priorité que les autres opérateurs unaires (la négation !, l'incrément ++, la décrémentation --). Dans une même expression, les opérateurs unaires *, &, !, ++, -- sont évalués de droite à gauche.

* Si un pointeur P pointe sur une variable X, alors *P peut être utilisé partout où on peut écrire X.

Exemple

Après l'instruction

```
P = &X;
```

les expressions suivantes, sont équivalentes:

```
Y = *P+1    ⇔ Y = X+1
*P = *P+10  ⇔ X = X+10
*P += 2     ⇔ X += 2
++*P        ⇔ ++X
(*P)++      ⇔ X++
```


Dans le dernier cas, les parenthèses sont nécessaires:

Comme les opérateurs unaires * et ++ sont évalués *de droite à gauche*, sans les parenthèses le pointeur P serait incrémenté, *non pas l'objet* sur lequel P pointe.

On peut uniquement affecter des adresses à un pointeur.

Le pointeur NUL

Seule exception: La valeur numérique 0 (zéro) est utilisée pour indiquer qu'un pointeur ne pointe 'nulle part'.

P : 

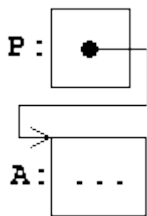
```
int *P;
P = 0;
```

Finalement, les pointeurs sont aussi des variables et peuvent être utilisés comme telles. Soit P1 et P2 deux pointeurs sur **int**, alors l'affectation

```
P1 = P2;
```

copie le contenu de P2 vers P1. P1 pointe alors sur le même objet que P2.

Résumons:



Après les instructions:

```
int A;
int *P;
P = &A;
```

A désigne le contenu de A

&A désigne l'adresse de A

P désigne l'adresse de A

***P** désigne le contenu de A

En outre:

&P désigne l'adresse du pointeur P

***A** est illégal (puisque A n'est pas un pointeur)

Exercice 1

```
main()
{
    int A = 1;
    int B = 2;
    int C = 3;
    int *P1, *P2;
    P1=&A;
    P2=&C;
    *P1=(*P2)++;
    P1=P2;
    P2=&B;
    *P1-=*P2;
    ++*P2;
    *P1*=*P2;
    A=++*P2**P1;
    P1=&A;
    *P2=*P1/=*P2;
    return 0;
}
```

Copiez le tableau suivant et complétez-le pour chaque instruction du programme ci-dessus.

| | <u>A</u> | <u>B</u> | <u>C</u> | <u>P1</u> | <u>P2</u> |
|-------|----------|----------|----------|-----------|-----------|
| Init. | 1 | 2 | 3 | / | / |

| | | | | | |
|--------------|---|---|---|----|---|
| P1=&A | 1 | 2 | 3 | &A | / |
| P2=&C | | | | | |
| *P1=(*P2)++ | | | | | |
| P1=P2 | | | | | |
| P2=&B | | | | | |
| *P1-=*P2 | | | | | |
| ++*P2 | | | | | |
| *P1*=*P2 | | | | | |
| A=++*P2**P1 | | | | | |
| P1=&A | | | | | |
| *P2=*P1/=*P2 | | | | | |

Tableaux et pointeurs

le nom d'un tableau représente l'adresse de son premier élément. En d'autre termes:

&tableau[0] et **tableau**

sont une seule et même adresse.

En simplifiant, nous pouvons retenir que *le nom d'un tableau est un **pointeur constant** sur le premier élément du tableau.*

Exemple

En déclarant un tableau A de type **int** et un pointeur P sur **int**,

```
int A[10];
int *P;
```

l'instruction:

P = A; est équivalente à **P = &A[0];**



Si P pointe sur une composante quelconque d'un tableau, alors P+1 pointe sur la composante suivante. Plus généralement,

P+i pointe sur la i-ième composante derrière P et

P-i pointe sur la i-ième composante devant P.

Ainsi, après l'instruction,

```
P = A;
```

le pointeur P pointe sur A[0], et

*** (P+1)** désigne le contenu de A[1]

*** (P+2)** désigne le contenu de A[2]

...

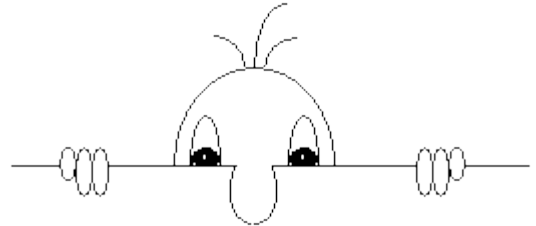
*** (P+i)** désigne le contenu de A[i]

Remarque

Au premier coup d'oeil, il est bien surprenant que P+i n'adresse pas le i-ième *octet* derrière P, mais la i-ième *composante* derrière P ...

Ceci s'explique par la stratégie de programmation 'défensive' des créateurs du langage C:

Si on travaille avec des pointeurs, les erreurs les plus perfides sont causées par des pointeurs malplacés et des adresses mal calculées. En C, le compilateur peut calculer automatiquement l'adresse de l'élément P+i en ajoutant à P la grandeur d'une composante multipliée par i. Ceci est possible, parce que:



- chaque pointeur est limité à un seul type de données, et
- le compilateur connaît le nombre d'octets des différents types.

Exemple

Soit A un tableau contenant des éléments du type **float** et P un pointeur sur **float**:

```
float A[20], x;  
float *P;
```

Après les instructions,

```
P = A;  
x = *(P+9);
```

x contient la valeur du 10-ième élément de A, (c.-à-d. celle de A[9]). Une donnée du type **float** ayant besoin de 4 octets, le compilateur obtient l'adresse P+9 en ajoutant $9 * 4 = 36$ octets à l'adresse dans P.

Rassemblons les constatations ci dessus :

Comme A représente l'adresse de A[0],

*** (A+1)** désigne le contenu de A[1]

*** (A+2)** désigne le contenu de A[2]

...

*** (A+i)** désigne le contenu de A[i]

Attention !

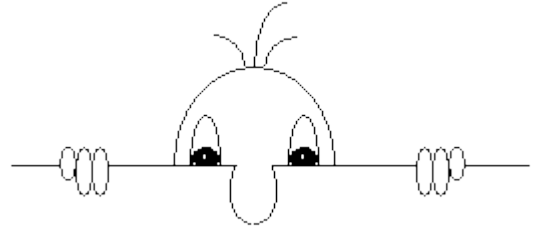
Il existe toujours une différence essentielle entre un pointeur et le nom d'un tableau:

- Un *pointeur* est une variable, donc des opérations comme **P = A** ou **P++** sont permises.
- Le *nom d'un tableau* est une constante,

donc des opérations comme **A = P** ou **A++** sont impossibles.

Ceci nous permet de jeter un petit coup d'oeil derrière les rideaux:

Lors de la première phase de la compilation, toutes les expressions de la forme **A[i]** sont traduites en ***(A+i)**. En multipliant l'indice **i** par la grandeur d'une composante, on obtient un indice en octets:



$$\langle \text{indice en octets} \rangle = \langle \text{indice élément} \rangle * \langle \text{grandeur élément} \rangle$$

Cet indice est ajouté à l'adresse du premier élément du tableau pour obtenir l'adresse de la composante **i** du tableau. Pour le calcul d'une adresse donnée par une adresse plus un indice en octets, on utilise un mode d'adressage spécial connu sous le nom '**adressage indexé**':

$$\langle \text{adresse indexée} \rangle = \langle \text{adresse} \rangle + \langle \text{indice en octets} \rangle$$

Presque tous les processeurs disposent de plusieurs registres spéciaux (*registres index*) à l'aide desquels on peut effectuer l'adressage indexé de façon très efficace.

Résumons Soit un tableau **A** d'un type quelconque et **i** un indice pour les composantes de **A**, alors

A désigne l'adresse de **A[0]**

A+i désigne l'adresse de **A[i]**

***(A+i)** désigne le contenu de **A[i]**

Si **P = A**, alors

P pointe sur l'élément **A[0]**

P+i pointe sur l'élément **A[i]**

***(P+i)** désigne le contenu de **A[i]**

Formalisme tableau et formalisme pointeur

A l'aide de ce bagage, il nous est facile de 'traduire' un programme écrit à l'aide du '*formalisme tableau*' dans un programme employant le '*formalisme pointeur*'.

Exemple

Les deux programmes suivants copient les éléments positifs d'un tableau **T** dans un deuxième tableau **POS**.

Formalisme tableau

```
main()
{
    int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9};
    int POS[10];
    int I,J; /* indices courants dans T et POS */
    for (J=0,I=0 ; I<10 ; I++)
        if (T[I]>0)
        {
            POS[J] = T[I];
            J++;
        }
    return 0;
}
```

Nous pouvons remplacer systématiquement la notation **tableau[I]** par ***(tableau + I)**, ce

qui conduit à ce programme:

Formalisme pointeur

```
main()
{
  int T[10] = {-3, 4, 0, -7, 3, 8, 0, -1, 4, -9};
  int POS[10];
  int I,J; /* indices courants dans T et POS */
  for (J=0,I=0 ; I<10 ; I++)
    if (*(T+I)>0)
    {
      *(POS+J) = *(T+I);
      J++;
    }
  return 0;
}
```

Sources d'erreurs

Un bon nombre d'erreurs lors de l'utilisation de C provient de la confusion entre soit contenu et adresse, soit pointeur et variable. Revoyons donc les trois types de déclarations que nous connaissons jusqu'ici et résumons les possibilités d'accès aux données qui se présentent.

Les variables et leur utilisation `int A;`

déclare une ***variable simple*** du type **int**

A désigne le contenu de A

&A désigne l'adresse de A

`int B[];`

déclare un ***tableau*** d'éléments du type **int**

B désigne l'adresse de la première composante de B.

(Cette adresse est toujours constante)

B[i] désigne le contenu de la composante i du tableau

&B[i] désigne l'adresse de la composante i du tableau

en utilisant le formalisme pointeur:

B+i désigne l'adresse de la composante i du tableau

***(B+i)** désigne le contenu de la composante i du tableau

`int *P;`

déclare un ***pointeur*** sur des éléments du type **int**.

P peut pointer sur des variables simples du type **int** ou
sur les composantes d'un tableau du type **int**.

P désigne l'adresse contenue dans P
(Cette adresse est variable)

***P** désigne le contenu de l'adresse dans P

Si P pointe dans un tableau, alors

P désigne l'adresse de la première composante

P+i désigne l'adresse de la i-ième composante derrière P
*** (P+i)** désigne le contenu de la i-ième composante derrière P

Exercice 2

Écrire un programme qui lit deux tableaux A et B et leurs dimensions N et M au clavier et qui ajoute les éléments de B à la fin de A. Utiliser le formalisme pointeur à chaque fois que cela est possible.

Exercice 3

Écrire un programme qui lit un entier X et un tableau A de type **int** au clavier et élimine toutes les occurrences de X dans A en tassant les éléments restants. Le programme utilisera des pointeurs pour parcourir le tableau.

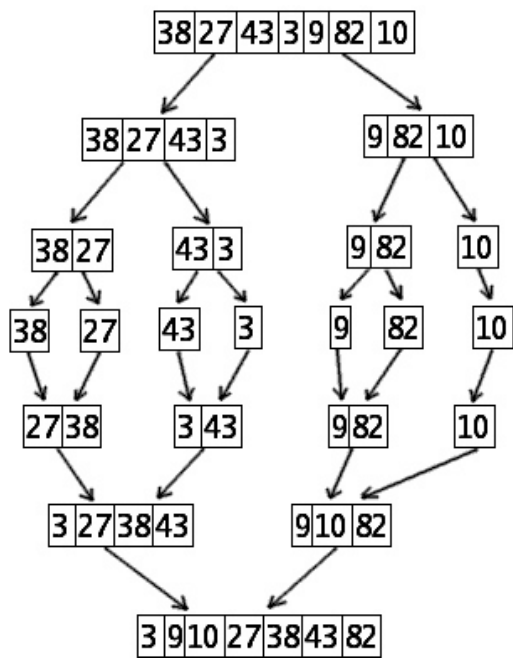
Exercice 4

Le **tri fusion** est un [algorithme de tri](#).

Le tri repose sur le fait que pour fusionner deux [listes](#)/tableaux trié(e)s dont la somme des longueurs est n , $n-1$ comparaisons au maximum sont nécessaires.

L'algorithme se déroule en trois étapes principales :

1. On trie les éléments deux à deux
2. On fusionne les listes obtenues
3. On recommence l'opération précédente jusqu'à ce qu'on ait une seule liste triée



Ecrire la fonction void mergesort(int * t, int n) qui trie le tableau t de taille n. Utilisez un tableau intermédiaire.