SUP'MANAGEMENT

ECOLE SUPERIEURE DE MANAGEMENT DE COMMERCE ET D'INFORMATIQUE

Reconnue par l'Etat

Les Pointeurs

Cours Algorithmique et Programmation C Avancées

Module Informatique I I

Plan

- Adressage de variables
- Les pointeurs
- Pointeurs et tableaux
- Tableaux de pointeurs
- Allocation dynamique de mémoire
- Exercices d'application

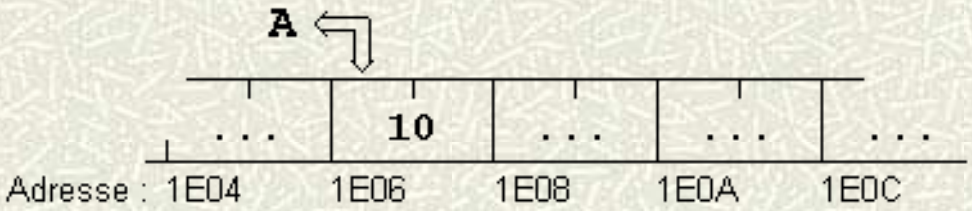
1. Adressage de variables

- **1.1. Adressage direct**

Adressage direct: Accès au contenu d'une variable par le nom de la variable.

Exemple

```
short A;
A = 10;
```

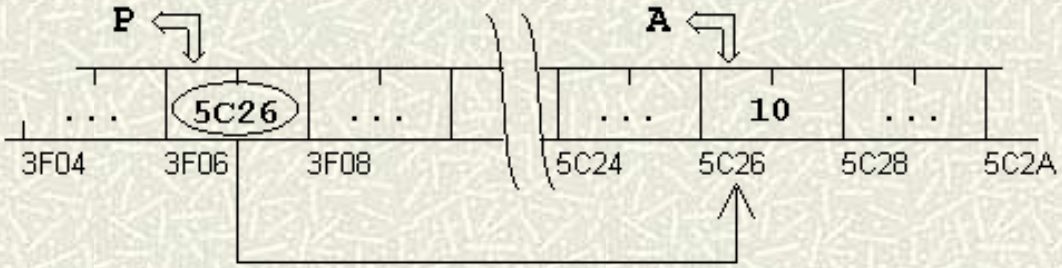


1. Adressage de variables

- **1.2. Adressage indirect**

Adressage indirect: Accès au contenu d'une variable, en passant par un pointeur qui contient l'adresse de la variable.

Exemple : Soit A une variable contenant la valeur 10 et P un pointeur qui contient l'adresse de A. En mémoire, A et P peuvent se présenter comme suit:



2. Les pointeurs

Définition:

*Un **pointeur** est une variable spéciale qui 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'.

2. Les pointeurs

- ***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.

2.1. Les opérateurs de base

- ***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 :

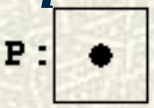
```
int N;
printf("Entrez un nombre entier : ");
scanf("%d", &N);
```

2.1. Les opérateurs de base

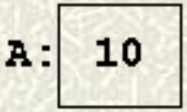
- ***Attention !***
  - L'opérateur **&** peut seulement être appliqué à des objets qui se trouvent dans la mémoire interne, c.-à-d. à des variables et des tableaux. Il ne peut pas être appliqué à des constantes ou des expressions.

- ***Représentation schématique***

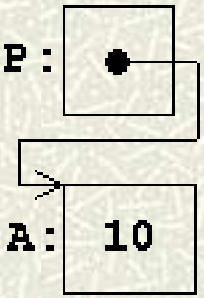
Soit P un pointeur non initialisé



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



Alors l'instruction **P = &A;**



2.1. Les opérateurs de base

- ***L'opérateur 'contenu de' : ****

  ***<NomPointeur>**

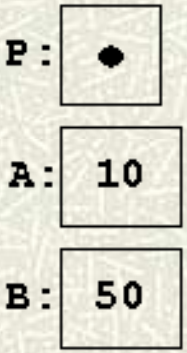
  désigne le contenu de l'adresse référencée par le pointeur <NomPointeur>

Exemple

A une variable contenant la valeur 10,

B une variable contenant la valeur 50

P un pointeur non initialisé:



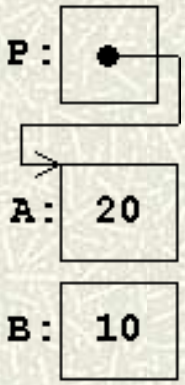
2.1. Les opérateurs de base

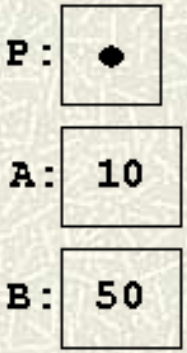
Après les instructions,

P = &A;

B = *P;

***P = 20;**





- P pointe sur A,

- le contenu de A (référencé par *P) est affecté à B, et

- le contenu de A (référencé par *P) est mis à 20.

2.1. Les opérateurs de base

- ***Déclaration d'un pointeur***

<Type> *<NomPointeur>

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

2.1. Les opérateurs de base

Une déclaration comme

int *PNUM;

peut être interprétée comme suit:

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

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

ou

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

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

2.2. Les opérations élémentaires sur pointeurs

- ***Priorité de * et &***
  - Les opérateurs ***** et **&** ont la même priorité que les autres opérateurs unaires (la négation **!**, l'incrémentation **++**, 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.

2.2. Les opérations élémentaires sur pointeurs

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++**

2.2. Les opérations élémentaires sur pointeurs

- ***Le pointeur NUL***

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

int *P;

P = 0;

- 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.

Exercice 1

```
Void 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;}
```

	A	B	C	P1	P2
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					

Exercice 1

	A	B	C	P1	P2
Init.	1	2	3	/	/
P1=&A	1	2	3	&A	/
P2=&C	**1**	**2**	**3**	**&A**	**&C**
***P1=(*P2)++**	**3**	**2**	**4**	**&A**	**&C**
P1=P2	**3**	**2**	**4**	**&C**	**&C**
P2=&B	**3**	**2**	**4**	**&C**	**&B**
***P1-=*P2**	**3**	**2**	**2**	**&C**	**&B**
++*P2	**3**	**3**	**2**	**&C**	**&B**
***P1*=*P2**	**3**	**3**	**6**	**&C**	**&B**
A=++*P2P1**	**24**	**4**	**6**	**&C**	**&B**
P1=&A	**24**	**4**	**6**	**&A**	**&B**
***P2=*P1/=*P2**	**6**	**6**	**6**	**&A**	**&B**

3. Pointeurs et tableaux

- **Adressage des composantes d'un tableau**

Exemple

int A[10];

int *P;

l'instruction: **P = A;** est équivalente à **P = &A[0];**

Adressage des composantes d'un tableau

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]**

Adressage des composantes d'un tableau

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.

3.2. Arithmétique des pointeurs

Toutes les opérations avec les pointeurs tiennent compte automatiquement du type et de la grandeur des objets pointés.

- **Affectation par un pointeur sur le même type**

  Soient P1 et P2 deux pointeurs sur le même type de données, alors l'instruction

  P1 = P2;

  fait pointer P1 sur le même objet que P2

3.2. Arithmétique des pointeurs

- **Addition et soustraction d'un nombre entier**

Si P pointe sur l'élément A[i] d'un tableau, alors

P+n pointe sur A[i+n]

P-n pointe sur A[i-n]

3.2. Arithmétique des pointeurs

- **Incrémentation et décrémentation d'un pointeur**

Si P pointe sur l'élément A[i] d'un tableau, alors après l'instruction

- **P++;** P pointe sur A[i+1]
- **P+=n;** P pointe sur A[i+n]
- **P--;** P pointe sur A[i-1]
- **P-=n;** P pointe sur A[i-n]

3.2. Arithmétique des pointeurs

- ***Domaine des opérations***

L'addition, la soustraction, l'incrémentation et la décrémentation sur les pointeurs sont seulement définies *à l'intérieur d'un tableau*.

- ***Soustraction de deux pointeurs***

P1-P2 fournit le nombre de composantes comprises entre P1 et P2.

- ***Comparaison de deux pointeurs***

On peut comparer deux pointeurs par <, >, <=, >=, ==, **!**=.

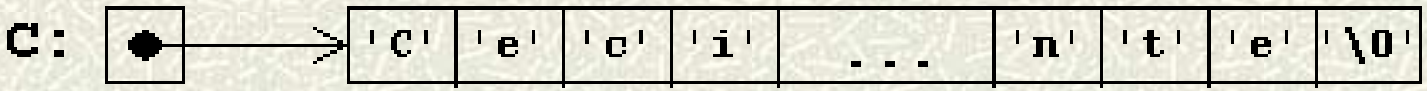
3.3. Pointeurs et chaînes de caractères

- **Pointeurs sur char et chaînes de caractères constantes**

Affectation et Initialisation

char *C;

C = "Ceci est une chaîne de caractères constante";



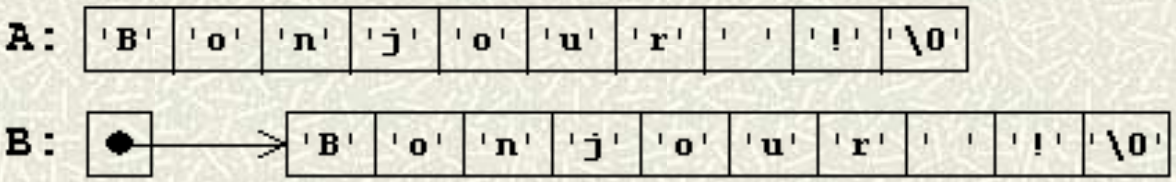
3.3. Pointeurs et chaînes de caractères

- ***Attention !***

Il existe une différence importante entre les deux déclarations:

char A[] = "Bonjour !"; /* un tableau */

char *B = "Bonjour !"; /* un pointeur */



3.3. Pointeurs et chaînes de caractères

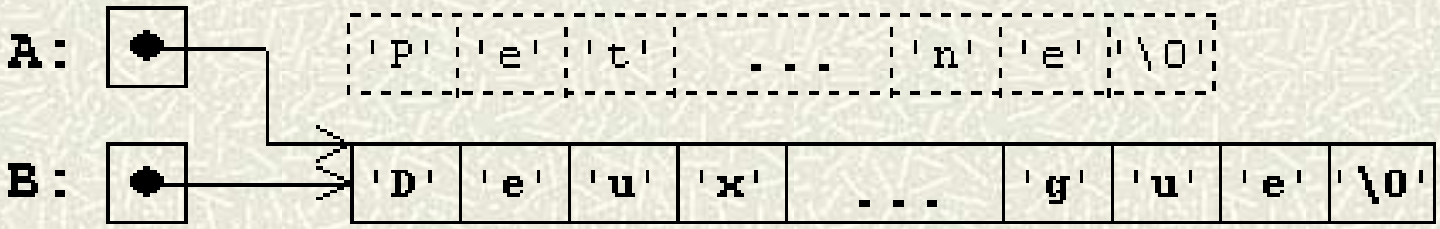
- ***Modification***

Exemple

char *A = "Petite chaîne";

char *B = "Deuxième chaîne un peu plus longue";

A = B;



3.3. Pointeurs et chaînes de caractères

Conclusions:

- Utilisons des *tableaux de caractères* pour déclarer les chaînes de caractères que nous voulons modifier.
- Utilisons des *pointeurs sur* ***char*** pour manipuler des chaînes de caractères constantes (dont le contenu ne change pas).
- Utilisons de préférence des *pointeurs* pour effectuer les manipulations à l'intérieur des tableaux de caractères.

4. Tableaux de pointeurs

- ***Déclaration d'un tableau de pointeurs***

  <Type> *<NomTableau>[<N>]

  déclare un tableau <NomTableau> de <N> pointeurs sur des données du type <Type>.

Exemple

double *A[10];

4. Tableaux de pointeurs

- ***Initialisation***

Exemple

char *JOUR[] = {"dimanche", "lundi", "mardi", "mercredi", "jeudi", "vendredi", "samedi"};

5. Allocation dynamique de mémoire

- **5.1. Déclaration statique de données**

```
float A, B, C;                  /* réservation de 12 octets */
short D[10][20];                /* réservation de 400 octets */
char E[] = {"Bonjour !"}; /* réservation de 10 octets */
char F[][10] = {"un", "deux", "trois", "quatre"};
                                /* réservation de 40 octets */
```

5.1. Déclaration statique de données

- ***Pointeurs***
  - double *G; /* réservation de p octets */
  - char *H; /* réservation de p octets */
  - float *I[10]; /* réservation de 10*p octets */

(En DOS: p =2 ou p = 4)

5.1. Déclaration statique de données

- ***Chaînes de caractères constantes***

- char *J = "Bonjour !"; /* réservation de p+10 octets */
- float *K[] = {"un", "deux", "trois", "quatre"};
  /* réservation de 4*p+3+5+6+7 octets */

5.2. Allocation dynamique

- **5.3. La fonction malloc et l'opérateur sizeof**

  La fonction **malloc** de la bibliothèque <*stdlib*> nous aide à localiser et à réserver de la mémoire au cours d'un programme.

La fonction malloc

malloc(<N>)

fournit l'adresse d'un bloc en mémoire de <N> octets libres ou la valeur zéro s'il n'y a pas assez de mémoire.

<N> est du type **unsigned int**.(max 65535 octets à la fois)

5.2. Allocation dynamique

L'opérateur unaire sizeof

- **sizeof <var> fournit la grandeur de la variable <var>**
- **sizeof <const> fournit la grandeur de la constante <const>**
- **sizeof (<type>) fournit la grandeur pour un objet du type <type>**

5.2. Allocation dynamique

Exemple

Nous voulons réserver de la mémoire pour X valeurs du type **int**; la valeur de X est lue au clavier:

```
int X; int *PNum;
printf("Introduire le nombre de valeurs :");
 scanf("%d", &X);
PNum = malloc(X*sizeof(int));
```

5.2. Allocation dynamique

exit

S'il n'y a pas assez de mémoire pour effectuer une action avec succès, il est conseillé d'interrompre l'exécution du programme à l'aide de la commande **exit** (de <*stdlib*>) et de renvoyer une valeur différente de zéro comme code d'erreur du programme

5.4. La fonction free

Si nous n'avons plus besoin d'un bloc de mémoire que nous avons réservé à l'aide de **malloc**, alors nous pouvons le libérer à l'aide de la fonction **free** de la bibliothèque *<stdlib>*.

free(<Pointeur>)

libère le bloc de mémoire désigné par le <Pointeur>;

n'a pas d'effet si le pointeur a la valeur zéro.

5.4. La fonction free

Attention !

- La fonction **free** peut aboutir à un désastre si on essaie de libérer de la mémoire qui n'a pas été allouée par **malloc**.
- La fonction **free** ne change pas le contenu du pointeur; il est conseillé d'affecter la valeur zéro au pointeur immédiatement après avoir libéré le bloc de mémoire qui y était attaché.
- Si nous ne libérons pas explicitement la mémoire à l'aide **free**, alors elle est libérée automatiquement à la fin du programme.

Exercices d'application

Série d’exercices