

4. LES POINTEURS

1.Introduction.

La plupart des langages de programmation offrent la possibilité d'accéder aux données dans la mémoire de l'ordinateur à l'aide de *pointeurs*, c.-à-d. à l'aide de variables auxquelles on peut attribuer les *adresses d'autres variables*.

En C, les pointeurs jouent un rôle primordial dans la définition de fonctions: Comme le passage des paramètres en C se fait toujours par la valeur, les pointeurs sont le seul moyen de changer le contenu de variables déclarées dans d'autres fonctions. Ainsi le traitement de tableaux et de chaînes de caractères dans des fonctions serait impossible sans l'utilisation de pointeurs.

2. 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

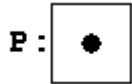
```
int N;  
printf("Entrez un nombre entier : ");  
scanf("%d", &N);
```

Attention !

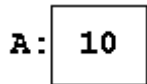
L'opérateur **&** peut seulement être appliqué à des objets qui se trouvent dans la mémoire interne, c.-à-d. à des variables et des tableaux. Il ne peut pas être appliqué à des constantes ou des expressions.

Représentation schématique

Soit P un pointeur non initialisé



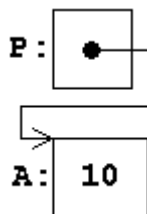
et A une variable (du même type) contenant la valeur 10 :



Alors l'instruction

P = &A;

affecte l'adresse de la variable A à la variable P. Dans notre représentation schématique, nous pouvons illustrer le fait que 'P pointe sur A' par une flèche:



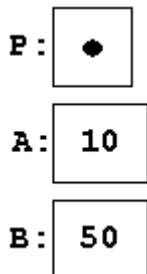
*L'opérateur 'contenu de' : **

***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:

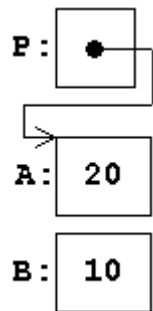


Après les instructions,

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.



3. Déclaration d'un pointeur

type *NomPointeur ;

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

Une déclaration comme **int *PTI;** peut être interprétée comme suit:

*"*PTI est du type **int**"*

ou

*"PTI est un pointeur sur **int**"*

ou

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

Exemple

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

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

Remarque

Lors de la déclaration d'un pointeur en C, ce pointeur est **lié explicitement** à un type de données. Ainsi, la variable PTI déclarée comme pointeur sur **int** ne peut pas recevoir l'adresse d'une variable d'un autre type que **int**.

3.1. Priorités des opérateurs * et &

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

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.

3.2. Valeur NULL

Le pointeur ne pointant sur rien (vide)

La valeur NULL est utilisée pour indiquer qu'un pointeur ne pointe 'nulle part'.



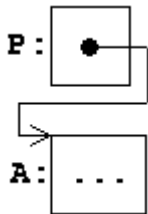
```
int *P;
P = NULL;
```

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)

3.3. Opérations de base: affectation, test

```
. ptr1 = NULL ;  
. ptr1 = ptr2 ;  
. ptr++ ;  
.   
. if ( ptr == NULL )  
. if ( ptr != NULL )
```

4. Pointeurs et Tableaux

Le nom d'un tableau représente l'adresse de son premier élément. En d'autres 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 ...

- 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.

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

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

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;
}
```

Résumons : A désigne le contenu de A

$\&A$ désigne l'adresse de A

`int TAB[] ;`
déclare un *tableau* d'éléments du type **int**

B désigne *l'adresse du premier élément de TAB*.
(Cette adresse est toujours constante)

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

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

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

***(TAB+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

Arithmétique des pointeurs

Comme les pointeurs jouent un rôle si important, le langage C soutient une série d'opérations arithmétiques sur les pointeurs que l'on ne rencontre en général que dans les langages machines. Le confort de ces opérations en C est basé sur le principe suivant:

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

- 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]

- 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]

Domaine des opérations

L'addition, la soustraction, l'incrémentation et la décrémentation sur les pointeurs sont seulement définies à l'intérieur d'un tableau. Si l'adresse formée par le pointeur et l'indice sort du domaine du tableau, alors le résultat n'est pas défini.

Exemples

```
int A[10];
int *P;
P = A+9; /* dernier élément -> légal */
P = A+10; /* dernier élément + 1 -> légal */
P = A+11; /* dernier élément + 2 -> illégal */
P = A-1; /* premier élément - 1 -> illégal */
```

- Soustraction de deux pointeurs

Soient P1 et P2 deux pointeurs qui pointent *dans le même tableau*:

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

Le résultat de la soustraction **P1-P2** est

- négatif, si P1 précède P2
- zéro, si P1 = P2
- positif, si P2 précède P1
- indéfini, si P1 et P2 ne pointent pas dans le même tableau

Plus généralement, la soustraction de deux pointeurs qui pointent dans le même tableau est équivalente à la soustraction des indices correspondants.

- Comparaison de deux pointeurs

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

La comparaison de deux pointeurs qui pointent *dans le même tableau* est équivalente à la comparaison des indices correspondants. (Si les pointeurs ne pointent pas dans le même tableau, alors le résultat est donné par leurs positions relatives dans la mémoire).

5. Pointeurs et chaînes de caractères

De la même façon qu'un pointeur sur **int** peut contenir l'adresse d'un nombre isolé ou d'une composante d'un tableau, un pointeur sur **char** peut pointer sur un caractère isolé ou sur les éléments d'un tableau de caractères. Un pointeur sur **char** peut en plus contenir *l'adresse d'une chaîne de caractères constante* et il peut même être *initialisé* avec une telle adresse.

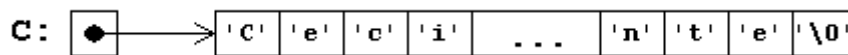
- Pointeurs sur char et chaînes de caractères constantes

Affectation

a) On peut attribuer *l'adresse d'une chaîne de caractères constante* à un pointeur sur **char**:

Exemple

```
char *C;  
C = "Ceci est une chaîne de caractères constante";
```



Nous pouvons lire cette chaîne constante (p.ex: pour l'afficher), mais il n'est pas recommandé de la modifier, parce que le résultat d'un programme qui essaie de modifier une chaîne de caractères constante n'est pas prévisible en ANSI-C.

Initialisation

b) Un pointeur sur **char** peut être initialisé lors de la déclaration si on lui affecte l'adresse d'une chaîne de caractères constante:

```
char *B = "Bonjour !";
```

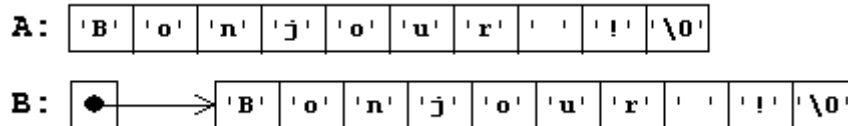
Attention !

Il existe une différence importante entre les deux déclarations:

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

A est un tableau qui a exactement la grandeur pour contenir la chaîne de caractères et la terminaison '\0'. Les caractères de la chaîne peuvent être changés, mais le nom A va toujours pointer sur la même adresse en mémoire.

B est un pointeur qui est initialisé de façon à ce qu'il pointe sur une chaîne de caractères constante stockée quelque part en mémoire. Le pointeur peut être modifié et pointer sur autre chose. La chaîne constante peut être lue, copiée ou affichée, mais pas modifiée.



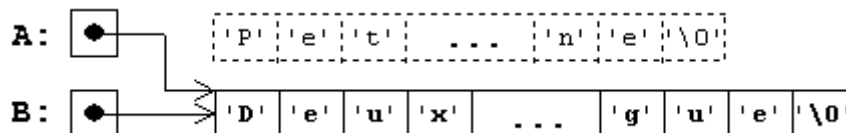
Modification

c) Si nous affectons une nouvelle valeur à un pointeur sur une chaîne de caractères constante, nous risquons de perdre la chaîne constante. D'autre part, un pointeur sur **char** a l'avantage de pouvoir pointer sur des chaînes de n'importe quelle longueur:

Exemple

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

Maintenant A et B pointent sur la même chaîne; la "Petite chaîne" est perdue:

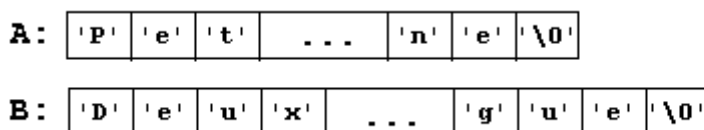


Attention !

Les affectations discutées ci-dessus ne peuvent pas être effectuées avec des tableaux de caractères:

Exemple

```
char A[45] = "Petite chaîne";
char B[45] = "Deuxième chaîne un peu plus longue";
char C[30];
A = B;           /* IMPOSSIBLE -> ERREUR !!! */
C = "Bonjour !"; /* IMPOSSIBLE -> ERREUR !!! */
```



Dans cet exemple, nous essayons de copier l'adresse de B dans A, respectivement l'adresse de la chaîne constante dans C. Ces opérations sont impossibles et illégales parce que ***l'adresse représentée par le nom d'un tableau reste toujours constante.***

Pour changer le contenu d'un tableau, nous devons changer les composantes du tableau l'une après l'autre (p.ex. dans une boucle) ou déléguer cette charge à une fonction de `<stdio.h>` ou `<string.h>`.

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.

Avantages des pointeurs sur char

Comme la fin des chaînes de caractères est marquée par un symbole spécial, nous n'avons pas besoin de connaître la longueur des chaînes de caractères; nous pouvons même laisser de côté les indices d'aide et parcourir les chaînes à l'aide de pointeurs.

Cette façon de procéder est indispensable pour traiter de chaînes de caractères dans des fonctions. En anticipant sur la matière du chapitre 10, nous pouvons ouvrir une petite parenthèse pour illustrer les avantages des pointeurs dans la définition de fonctions traitant des chaînes de caractères:

Pour fournir un tableau comme paramètre à une fonction, il faut passer *l'adresse du tableau* à la fonction. Le nom du tableau est l'adresse du tableau.

6. Allocation dynamique de mémoire

Nous avons vu que l'utilisation de pointeurs nous permet de mémoriser économiquement des données de différentes grandeurs. Si nous générons ces données pendant l'exécution du programme, il nous faut des moyens pour réserver et libérer de la mémoire au fur et à mesure que nous en avons besoin. Nous parlons alors de *l'allocation dynamique* de la mémoire.

Revoyons d'abord de quelle façon la mémoire a été réservée dans les programmes que nous avons écrits jusqu'ici.

6.1 Déclaration statique de données

Chaque variable dans un programme a besoin d'un certain nombre d'octets en mémoire. Jusqu'ici, la réservation de la mémoire s'est déroulée automatiquement par l'emploi des déclarations des données. Dans tous ces cas, le nombre d'octets à réserver était déjà connu pendant la compilation. Nous parlons alors de la *déclaration statique* des variables.

Exemples

```
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 */
```

Pointeurs

Le nombre d'octets à réserver pour un *pointeur* dépend de la machine et du 'modèle' de mémoire choisi, mais il est déjà connu lors de la compilation. Un pointeur est donc aussi déclaré statiquement. Supposons dans la suite qu'un pointeur ait besoin de p octets en mémoire.

Exemples

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

Chaînes de caractères constantes

L'espace pour les *chaînes de caractères constantes* qui sont affectées à des pointeurs ou utilisées pour initialiser des pointeurs sur **char** est aussi réservé automatiquement:

Exemples

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

6.2. Allocation dynamique

Souvent, nous devons travailler avec des données dont nous ne pouvons pas prévoir le nombre et la grandeur lors de la programmation. Ce serait alors un gaspillage de réserver toujours l'espace maximal prévisible. Il nous faut donc un moyen de gérer la mémoire lors de l'exécution du programme.

Exemple

Nous voulons lire 10 phrases au clavier et mémoriser les phrases en utilisant un tableau de pointeurs sur **char**. Nous déclarons ce tableau de pointeurs par:

```
char *TEXTE[10];
```

Pour les 10 pointeurs, nous avons besoin de 10*p octets. Ce nombre est connu dès le départ et les octets sont réservés automatiquement. Il nous est cependant impossible de prévoir à l'avance le nombre d'octets à réserver pour les phrases elles-mêmes qui seront introduites lors de l'exécution du programme ...

Allocation dynamique

La réservation de la mémoire pour les 10 phrases peut donc seulement se faire *pendant l'exécution du programme*. Nous parlons dans ce cas de l'*allocation dynamique* de la mémoire.

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. Elle nous donne accès au tas (*heap*); c.-à-d. à l'espace mémoire libre.

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.

Cast de malloc()


La fonction **malloc()** donne l'adresse d'une zone de mémoire, mais cette adresse retournée ne permet pas de savoir quel est le type des valeurs qui seront stockées dans cette zone. En effet, ce que l'on indique à **malloc()**, c'est simplement le nombre d'octets demandés. Dans ce sens, on dit que l'adresse renvoyée ou donnée par **malloc()** est **générique** : cela se traduit en langage C en disant que l'adresse de la zone donnée par **malloc()** est de type `void*` : pointeur générique. Or cette adresse sera affectée à un pointeur du programme, dont le type sera `long*` ou `char*` ou encore `double*` : au sens strict, ces types sont différents. Pour éviter les messages d'avertissement et garder une bonne compatibilité, on prend systématiquement la précaution de faire précéder l'utilisation de **malloc()** d'un transtypage, en indiquant entre parenthèses le type du pointeur dans lequel sera affecté l'adresse que la commande fournira.

De même, selon le type du pointeur dans lequel on affectera la valeur donnée par la commande.

Exemple :

```
float *ptrF;
```

```
ptrF = (float *)malloc(sizeof(float *));
```



→ même type `float *`

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.

* La fonction **free** ne change pas le contenu du pointeur; il est conseillé d'affecter la valeur NULL au pointeur immédiatement après avoir exécuter.

* Si nous ne libérons pas explicitement la mémoire à l'aide **free**, alors elle est libérée automatiquement à la fin du programme.

La fonction calloc

calloc() donne l'adresse générique (donc de type `void*`) d'une zone permettant de stocker le nombre de variables demandé, chacune de ces variables occupant la taille, en octets, précisée.

```
int nb=10;
```

```
double *zone = (double *)calloc(nb, sizeof(double));
```

7. Pointeurs et structures.

Soit les déclarations des structures suivantes

Structures :

```
typedef struct date {  
    int jour;  
    char mois[20];  
    int annee;  
} date;
```

```
typedef struct {  
    char nom[32];  
    char prenom[32];  
    date date_naissance;  
} eleve;
```

Les champs peuvent donc être de n'importe quel type connu : types de bases, tableaux, **pointeurs** ou autre structure.

Soit la variable :

```
eleve E1; /* un eleve */
```

Déclaration d'un pointeur sur une structure :

```
int *pn; /* un pointeur sur un entier */
```

```
eleve *pE; /* un pointeur sur un eleve */
```

1.4 Accès aux champs d'une structure via un pointeur

Avec `eleve *pE`, on peut accéder aux champs de deux façons :

- `(*pE).nom` : `*pE` représente la structure pointée par `pE`
- `pE->nom` : notation spécialisée équivalente, qui est un raccourci très pratique.
La flèche `->` (2 caractères) indique que l'on suit le pointeur pour arriver au champ.

1.5 Allocation dynamique de structure.

```
eleve *ptr ;
```

```
ptr = (eleve *) malloc ( sizeof(eleve));
```

7. Pointeurs et fonctions.

7.1. Relation entre paramètres d'une fonction et pointeurs

A part les tableaux, les paramètres des fonctions sont des variables locales : modifications faites dans la fonction

Le paramètre d'une fonction est la copie de l'argument : l'argument n'est pas modifié, on dit que l'on fait un **passage par valeur**.

On utilisera un mécanisme permettant de garder les modifications.

7.2. Passage par adresse.

Ce mécanisme est appelé passage par adresse ou passage par pointeur (c'est la même chose, car un pointeur est une adresse).

Plutôt que de fournir à la fonction une valeur, on va fournir **son adresse** ou **un pointeur** sur cette valeur.

Rappel : une constante n'a pas d'adresse, ce mécanisme n'est utilisable qu'avec des variables.

D'ailleurs, peut-on changer la valeur d'une constante numérique ?

Exemple

Passage par valeur

Passage par adress

<pre>void modifier(int X) { X = X + 1; }</pre>	<pre>void modifier (int *X) { *X = *X + 1; }</pre>
---	---

7.3. Passage d'une structure en paramètre

Une fonction peut prendre une variable structure en paramètre.

On peut passer par valeur

```
void Affiche( eleve E );
```

ou par adresse :

```
void Affiche( eleve *pE );
```

On préférera toujours la deuxième solution, qui évite la duplication de la structure sur la pile (opération qui peut être coûteuse,).

Dans la fonction, on utilise alors la notation **->**.

7.4. Fonction retournant un pointeur

Une fonction peut retourner un pointeur de variable de type quelconque

```
int * nbre ( int Tab[], int p );
```

```
eleve * extraire ( eleve E , int p );
```

```
float * nbre ( eleve E , int p );
```

Il convient d'être prudent lors de l'utilisation d'une fonction retournant un pointeur. Il faudra éviter l'erreur qui consiste à retourner l'adresse d'une variable *temporaire*.

Exemple

```
#include <stdio.h>
void main() {
    char *p;
    char *ini_car(void);
    p = ini_car();
    printf("%c\n", *p);
}

char *ini_car(void) {
    char c;
    c = '#';
    return(&c);      <===  ERREUR
}
```

7.5. Pointeur de fonction

Une fonction en C n'est pas une variable. Cependant on peut accéder à son adresse et manipuler des pointeurs de fonction.

Déclaration

Voici trois exemples de pointeurs de fonction :

```
int (*pf1) (void);
```

pf1 pointe sur une fonction qui a `int` pour type de retour et qui n'a pas de paramètres.

```
double (*pf2) (double, double);
```

pf2 : pointe sur une fonction qui renvoie un **double** et qui a deux `double` en paramètres.

```
void (*pf3) ();
```

pf3 pointe sur une fonction qui ne renvoie rien. On ne connaît pas les paramètres de la fonction.

Remarque : Dans la définition du pointeur de fonction, le parenthésage est obligatoire (la priorité de l'opérateur `*` est trop faible).

Prenons par exemple, la fonction suivante :

```
double max (double a, double b) {  
    return (a>b)? a:b;  
}
```

On associe pointeur de fonction et fonction comme cela : (les deux lignes sont équivalentes)

```
pf2 = max    /* forme la plus portable */  
pf2 = &max; /* forme la plus "cohérente" */
```

Maintenant `(*pf2)(10.5,21.0);` ,

`pf2(10.5,21.0)`

et `max(10.5,21.0);` donnent le même résultat.

La forme avec `*` est préconisée : la deuxième forme est plus difficile à repérer en cas d'erreur car on ne voit pas directement qu'il s'agit d'un pointeur, sauf si le nom est explicite.