Chapitre 3 : Listes chaînées

1. Introduction

Nous avons vu dans le premier semestre que le programme est un ensemble de données et un ensemble d'instructions où ces données sont stockées en mémoire sous forme de variables.

Une **variable** est un emplacement en mémoire qui possède une adresse de stockage, un nom, un type et une valeur.

- **Adresse** : Chaque variable stockée en mémoire a une adresse qui indique son emplacement. C'est un nombre naturel qui identifie le premier octet dans lequel se trouve la variable. Habituellement, il est écrit dans le système hexadécimal 16, tel que : 0x5A63
- **Nom :** Identifiant utilisé par le programmeur pour se référer à la valeur stockée et on manipule le nom de la variable au lieu de l'adresse. Ex : poid
- **Type :** Tout dans un ordinateur est composé de 0 et 1. Le type détermine comment les traduire, ainsi que la taille qui doit être réservée en mémoire, c'est-à-dire le nombre de bits et les opérations autorisées. Exemple : int (32 bits)
- **Valeur :** C'est le contenu des bits qui composent sa valeur, et c'est généralement la chose qui change pendant l'exécution du programme, comme : 15

Lors de l'exécution du programme et lorsqu'il rencontre une instruction du type déclaration de variable, par exemple var age :entier (int age ;), le programme demande au système d'exploitation (Windows) de réserver une place en mémoire de la taille x (selon le type). Et après réservation, le système renvoie l'adresse mémoire qu'on peut l'utiliser comme variable.
Pour obtenir la valeur de la variable, il suffit d'écrire son nom, mais pour obtenir son adresse, c'est-à-dire son emplacement en mémoire, dans algorithme on met le symbole **@** avant le nom de la variable, et en C on met le symbole **&** avant le nom de la variable.

Exemple:

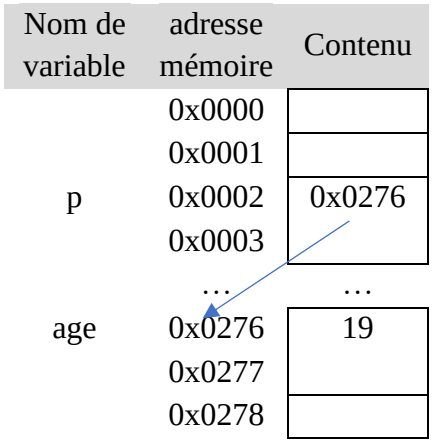
```
écrire("valeur de age =", age," son adresse =",@age);
printf("valeur de age = %d son adresse = %p",age,&age);
```

%p est un format permettant de traiter la valeur &age comme une adresse en mémoire, c'est-à-dire un nombre écrit en hexadécimal 16. On peut utiliser %d pour le voir en décimal. Ici, **age** est la valeur de la variable et &age est son adresse en mémoire où elle peut changer chaque fois que nous exécutons le programme.

2. Les pointeurs

Un **pointeur** est une variable dont la valeur pointe vers une adresse dans la mémoire de l'ordinateur. Cette adresse est soit une variable, soit un programme. Il est utilisé pour passer des paramètres par adresse, réserver de la mémoire de manière dynamique ou définir des types récursifs (listes, piles et files), et il a d'autres utilisations.

Exemple:
La mémoire peut être considérer comme un tableau numéroté de 0 à la capacité de mémoire -1
Dans l'exemple suivant, deux variables ont été réservées, la première est l'âge de type entier situé à l'adresse 0x0276 et contient la valeur 19 ici 0x signifie que le nombre soit écrit dans le système hexadécimal 16 (0x0276 = 630 dans le système décimal). La deuxième variable est p et sa valeur est 0x0276, qui représente l'emplacement où se situe l'âge. On dit donc que p pointe vers age.



La création

Pour créer une variable de type pointeur, dans l'algorithme nous ajoutons le symbole ^ devant le type de variable. Où il prend la forme suivante :

```
var p1, p2 :^type
```

Pour créer une variable de pointeur en C, nous ajoutons * avant le nom de la variable

```
type *p1,*p2;
```

Ici ^ ou * indique que la variable est du type pointeur, c'est à dire une adresse mémoire, tandis que type est le type du contenu de cet emplacement.

Exemple : On déclare six variables x et y de type entier, p1 et p2 de type pointeur vers entier, z de type réel et pz de type pointeur vers réel.

`int x,*p1,y,*p2 ;` `float z,*pz;`	`Var x, y : entier  p1, p2 : ^ entier` `z : réel  pz : ^réel`

Lors de la déclaration d'une variable, elle porte une valeur indéfinie, il est donc recommandé de lui attribuer la valeur **NULL** en majuscules, ce qui signifie que le pointeur ne pointe nulle part (défini à l'intérieur de stdio.h, qui représente le chiffre 0)

```
p1= NULL;
```

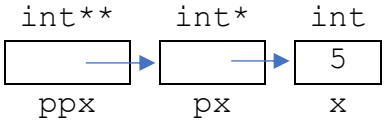
La variable p1 peut prendre l'adresse de la variable x ou la valeur de la variable p2, mais elle ne peut pas prendre l'adresse de la variable z, l'adresse de p2 ou la valeur de pz.

Opérations valides	Opérations non valides	L'explication
`p1=&x ;`	`p1=x;`	p1 est de type pointeur et x est un entier
`p2=p1;`	`p1=&z ;`	p1 est un pointeur d'entier et &z est une adresse de réel

`pz=&z ;`	`pz=p1;`	pz est un pointeur sur un réel et p1 est un pointeur sur un entier
	`p2=&p1 ;`	P2 est un pointeur vers entier, mais **&p1** est l'adresse d'un pointeur sur un entier.
	`p1=&(0x0276);`	Doit etre une variable et pas un nombre.

Il faut faire la différence entre l'adresse stockée dans le pointeur et l'adresse du pointeur lui-même, car le pointeur est une variable qui a une adresse comme le reste des variables, et donc son adresse peut être assignée à un autre pointeur, mais dans ce cas le second type de pointeur doit être l'adresse d'un pointeur du premier type.

Par exemple : x est de type entier (int), et px contient l'adresse de x, donc son type est (int*) et ppx contient l'adresse de px, donc son type est (int**) comme indiqué dans le schéma suivant :



Il est déclaré comme suit :

```
int x,*px,**ppx;
x=5 ;
px=&x ;
ppx=&px ;
```

typedef peut être utilisé pour créer de nouveaux types et la déclaration ci-dessus devient quelque chose comme ça :

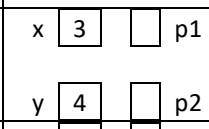
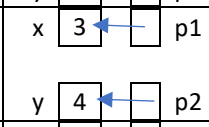
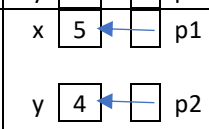
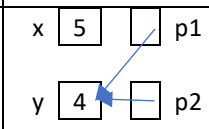
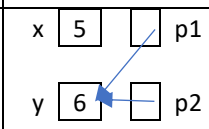
```
typedef int* pint;
typedef int** ppint;
pint px;
ppint ppx;
```

Utilisation :

Il est rare que nous traitions les adresses mémoire comme des nombres directs, mais nous les traitons comme des adresses pour des variables existantes. Pour obtenir l'adresse d'une variable, nous utilisons l'opération @ dans l'algorithme ou & dans le langage de programmation C **avant** le **nom** de variable, et pour récupérer la valeur de la variable (Déréférencement) à partir de son adresse stockée dans un pointer, nous utilisons le symbole ^ **après** le **nom** de la variable dans l'algorithme et *** avant** le **nom** de la variable dans le langage de programmation C.

```
p←@x ⇒ p^ ⇔ x
p=&x ⇒ *p ⇔ x
```

Exemple:

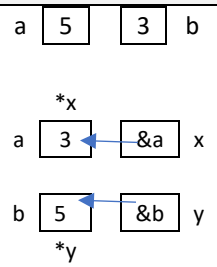
C	l'algorithme	mémoire	L'explication
`int x,*p1,y,*p2 ;`	Var x, y : entier p1, p2 : ^ entier		
`x=3 ;` `y=4 ;`	x←3 y←4		
`p1=&x ;` `p2=&y ;`	p1←@x p2←@y		Ici p1 contient l'adresse de x et p2 contient l'adresse de y
`*p1=5;`	p1^←5		On affecte le numéro 5 à la variable dont l'adresse est en p1, et à ce moment c'est la variable x, comme si la variable x avait un deuxième nom, qui est *p1 peut être remplacé par l'instruction x=5 ;
`p1=p2;`	p1←p2		Nous attribuons la valeur de p2, qui représente l'adresse de y, à p1, de sorte que y, *p1 et *p2 deviennent la même variable à ce moment.
`*p1=6;`	p1^←6		On affecte le chiffre 6 à la variable dont l'adresse est en p1 et à ce moment c'est la variable y peut être remplacé par l'instruction y=6 ; ou *p2=6 ;

Des remarques :

- Pour comprendre les pointeurs, il est toujours recommandé de dessiner les variables, où le pointeur porte une flèche vers la variable qui porte son adresse, et nous symbolisons le pointeur qui porte la valeur NULL, c'est-à-dire qu'il ne pointe pas à n'importe quel endroit avec 
- Un pointeur est toujours de type simple, tandis que la variable dont il contient son adresse peut être de type complexe (tableau ou structure).
- Tenter de récupérer la valeur d'un pointeur non initialisé ou une valeur NULL entraîne l'arrêt du programme.
  - Une valeur (adresse variable) doit être affectée au pointeur avant d'essayer de récupérer la valeur vers laquelle il pointe.
  - Avant de récupérer la valeur sur laquelle pointe le pointeur, vous devez vous assurer qu'elle ne porte pas la valeur NULL.
- Il est maintenant possible de comprendre le passage des paramètres par adresse dans les sous-programmes.

Exemple

C	mémoire	l'explication

`void echanger(int *x, int *y){`
`int t;`
`t=*x;`
`*x=*y;`
`*y=t;`
`}`
`int a=5,b=3;`
`echanger(&a,&b);`
		Ici x et y sont deux pointeurs et lors de l'appel à la fonction on affecte à x l'adresse de la variable a soit x=&a et à y l'adresse de la variable b soit y=&b et à l'intérieur de la fonction echanger pour obtenir la variable dont l'adresse x porte nous utilisons l'opération * où *x à ce moment représente la variable a et *y représente la variable b

3. Opérations sur les pointeurs

Supposons que P et Q soient des pointeurs et que i soit un entier. Le tableau suivant résume les opérations pouvant être effectuées sur les pointeurs :

Opération algorithme	Opération C	Type du 2e opérant	Type du résultat	Exemple	Observation
+	+	int	Pointeur	P + i	Renvoie un pointeur vers le i^{iem} élément après P dans un tableau
	++		Pointeur	P++	Renvoie un pointeur vers l'élément suivant immédiatement P dans un tableau
-	-	int	Pointeur	P – i	Renvoie un pointeur vers le i^{iem} élément avant P dans un tableau
	--		Pointeur	P--	Renvoie un pointeur vers l'élément précédant immédiatement P dans un tableau
-	-	Pointeur du même type	int	P - Q	Renvoie le nombre d'éléments entre P et Q où P et Q doivent pointer vers le même tableau
=	==	Pointeur	Booléen	P == Q	C'est vrai si P et Q ont la même adresse, c'est-à-dire qu'ils pointent vers le même endroit
≠	!=	Pointeur	Booléen	P != Q	C'est vrai si P et Q sont différents
^	*		Type de valeur	*P	Pour récupérer la valeur dont elle contient l'adresse

4. Gestion dynamique de la mémoire

La méthode que nous connaissons jusqu'à présent pour réserver des variables en mémoire s'appelle la réservation statique, où la variable est déclarée au début du programme, et le compilateur réserve la mémoire nécessaire de manière automatique, et la variable est n'est supprimée qu'à la fin de l'exécution du programme (ou du sous-programme dans le cas d'une variable locale) . Mais parfois, nous devons réserver une quantité de mémoire, que ce soit un tableau avec N éléments, par exemple, et N ne peut être connu que lors de l'exécution, nous déclarons donc un pointeur et lorsque N devient disponible, nous réservons le tableau.

Le développeur dispose d'un ensemble de fonctions lui permettant de gérer la mémoire de manière dynamique, c'est-à-dire pendant l'exécution.

En algorithme :

Il existe trois procédures de gestion dynamique de la mémoire :

1. **allouer**() pour réserver un tableau où il prend en paramètre le nom du pointeur (nom du tableau) et le nombre des éléments

   ```
   allouer(nom_tab,nb_elements)
   ```

   Exemple :

   ```
   allouer(t,10)
   ```

2. **réallouer**() pour modifier la taille du tableau, que ce soit en augmentant ou en diminuant, et prend en paramètre le nom du pointeur (le nom du tableau) et le nouveaux nombre des éléments (nouvelle taille), il conserve les valeurs des éléments précédemment réservés et supprime le surplus ou ajoute de nouveaux éléments au tableau

   ```
   réallouer(nom_tab, nouvelle_taille)
   ```

   Exemple :

   ```
   réallouer(t,15)
   ```

3. **désallouer**() pour supprimer le tableau réservé avec **allouer()** et prend en paramètre le nom du pointeur (nom de du tableau)

   ```
   désallouer(nom_tab)
   ```

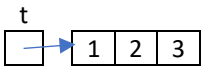
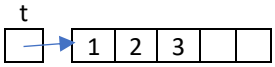
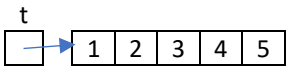
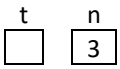
   Exemple :

   ```
   désallouer(t)
   ```

Après avoir créé un tableau t par allouer(), ses éléments sont accessibles par les crochets [ ] ou par l'opération de récupération ^, où l'on sait que le pointeur t contient l'adresse du premier élément t[0] soit @t[0]= t et t^=t[0] et pour obtenir L'adresse du deuxième élément t[1] ajoute 1 à t c'est-à-dire @t[1] ⇔t+1 et (t+1)^ ⇔t[1] donc le l'adresse de t[i] est t+i. c'est-à-dire @t[i]⇔(t+i) et (t+i)^ ⇔t[i].

Exemple:

algorithme	mémoire	l'explication
var t : ^réel n :entier	t n	Un pointeur t et une variable n représentant le nombre de ses éléments sont déclarés
début ecrire("entrer le nombre des éléments") lire(n)	t n 3	Soit n prendre 3
allouer(t ,n)	t	allouer() réserve un tableau de trois éléments et met son adresse à t

t[0] ←1 t[1] ←2 t[2] ←3 t^ ←1 (t+1)^ ←2 (t+2)^ ←3		Nous remplissons le tableau où nous pouvons utiliser les crochets [ ] ou utiliser ^ où t[i] ⇔ (t+i)^
reallouer(t,n+2)		L'appel de reallouer() redimensionne le tableau à 5
t[3] ←4 t[4] ←5 (t+3)^ ←4 (t+4)^ ←5		Nous remplissons les deux éléments ajoutés
désallouer(t)		On appelle désallouer() pour supprimer le tableau

En C

La gestion de la mémoire en C est un peu différente de celle des algorithmes, et avant de pouvoir en apprendre davantage, nous devons apprendre sizeof et le changement de type.

4.1. L'opération « sizeof »

Une variable occupe plus ou moins de place en mémoire selon son type. Comme une variable de type char prend un octet, alors qu'une variable de type int nécessite deux ou quatre octets, selon la version C. Pour connaître la taille requise pour un type, on utilise sizeof(), qui prend le nom du variable ou le nom du type pour renvoyer le nombre d'octets dont il a besoin en mémoire.

```
int sizeof(type);
```

Exemple:

```
float t[20] ;
printf("char   : %d octets\n", sizeof(char));
printf("int    : %d octets\n", sizeof(int));
printf("double : %d octets\n", sizeof(double));
printf("la taille de t: %d octets\n", sizeof(t));
printf("la taille de t: %d octets\n", 20*sizeof(float));
```

qui affiche sur l'écran

```
char   : 1 octets
int    : 4 octets
double : 8 octets
la taille de t: 80 octets
la taille de t: 80 octets
```

La taille d'un tableau peut être trouvée en multipliant la taille d'un seul élément par le nombre des éléments.

4.2. Changement de type : transtypage/casting

Parfois, nous devons convertir une valeur spécifique d'un type à un autre, et pour forcer le compilateur à changer le type d'une valeur spécifique, nous utilisons la formule suivante :

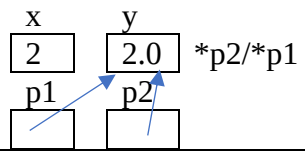
```
(type) expression
```

Où l'expression est convertie en type

Exemple 1

`int A=8,B=3 ;`	
`float R=A/B ;`	Les operants A et B étant de type entier, l'opération / effectue une division entiere R=8/3
`printf("no casting R=%f \n",R);`	affiche no casting R=2.000000
`R=(float)A/B ;`	Nous convertissons la valeur de A (pas la variable A) en un nombre réel, puis nous effectuons le processus de division, de sorte que l'opération devient R = 8,0/3
`printf("with casting R=%f \n",R);`	affiche with casting R=2.666666

Exemple 2

`int x,*p1 ;`	Un entier et un pointeur d'entier
`float y=2,*p2 ;`	Un nombre réel et un pointeur vers un nombre réel
`x=(int)y ;`	il Convertit la valeur de y en un entier et le met dans x, donc x prend la valeur 2
`p2=&y ;`	p2 prend l'adresse de y
`p1=(int*)p2 ;`	Conversion de l'adresse d'un float à l'adresse d'un int, mais l'adresse de du variable reste dans les deux variables, qui est l'adresse de y 
`printf("x=%d \n",x);`	Affiche x=2
`printf("*p2=%f \n",*p2);`	Affiche *p2=2.000000 la même que y
`printf("*p1=%d \n",*p1);`	Affiche *p1=1073741824 Parce que traduire les bits d'un nombre réel en entier ne donne pas le même nombre

4.3. Gestion de la mémoire en C

La gestion dynamique de la mémoire en C se fait à l'aide de quatre fonctions définies dans la librairie stdlib :

- malloc()) ‹**m**emory **alloc**ation Cela signifie réserver de la mémoire) Il demande au système d'exploitation de réserver la quantité de mémoire requise.

```
void * malloc(int taille);
```

Il prend en paramètre la taille mémoire requise (le nombre d'octets) et retourne un pointeur vers la mémoire qui a été réservée, ou retourne NULL en cas d'échec du processus car la taille requise n'est pas disponible.

Exemple:

```
float *t;

t=(float *)malloc(10*sizeof(float));
```

t=	(float *)	malloc(	10*	sizeof(	float	));
Nom du tableau	Convertir en type du pointeur	Pour réserver tableau	Nombre de éléments	La taille de chaque élément	Type de chaque élément	

- free(), pour retourner la mémoire précédemment réservée par le malloc du système d'exploitation afin qu'elle puisse être utilisée par d'autres programmes.

```
void free( void * pointeur );
```

Prend comme parametre un pointeur précédemment réservée. Il est recommandé d'affecter NULL au pointeur après avoir appelé free pour s'assurer que le pointeur ne pointe nulle part et pour éviter toute erreur.

Exemple :

```
free(t);
```

- realloc(), pour changer la taille de la mémoire réservée, soit en augmentant soit en diminuant.

```
void * realloc(void * pointeur, int nouvelle_taille);
```

Où la fonction appelle malloc pour réserver une nouvelle place de la taille de la nouvelle_taille, puis copie toutes les valeurs du tableau "pointeur" vers la nouvelle location (ou supprime les éléments supplémentaires si la nouvelle_taille est inférieure à l'ancienne taille), puis supprime l'ancien tableau réservée en appelant free, et si l'opération réussit, elle renvoie un pointeur vers le nouvel emplacement sinon renvoie NULL.

Exemple:

```
t=(float*)realloc(t, 20*sizeof(float));
```

- calloc(), comme malloc, sauf qu'il met des zéros dans la mémoire réservée.

```
void * calloc(int nb_element, int taille_element);
```

Il prend nb_element, qui représente le nombre d'éléments du tableau, et taille_element, qui représente la taille d'une cellule, et renvoie un pointeur vers l'espace réservé.

Exemple :

```
t=(float*)calloc(10,sizeof(float));
```

Observation :

- Dans la leçon sur les fonctions, nous avons vu que void signifie que la fonction ne renvoie rien, mais void* signifie que la fonction renvoie un pointeur de type indéfini.
- Le type void* doit être convertit vers le type du pointeur qui contiendra l'adresse en plaçant le type pointeur entre parenthèses avant les noms de fonction malloc, calloc et realloc, mais cette conversion n'est pas nécessaire en C++.
- Pour utiliser ces fonctions, vous devez appeler la bibliothèque stdlib ou alloc à l'aide de l'instruction suivante :

```
#include <stdlib.h>
#include <alloc.h>
```

l'opération `sizeof` n'est pas une fonction, les parenthèses peuvent donc être omises.

Lorsque nous réservons de la mémoire, nous suivons ces étapes :

1. Nous réservons de la mémoire avec malloc.
2. On s'assure que le processus de réservation s'est terminé avec succès en utilisant **`if`** `(pointeur ! =` **`NULL`**`)`
3. Lorsque nous avons fini d'utiliser l'espace réservé, nous retournons la mémoire au système via free

Exemple

C	l'explication
`#include <stdio.h>` `#include <stdlib.h>`	inclusion de la bibliothèque stdlib
`int main(void) {` `  char *str;`	Déclarer un pointeur de type char
`str = (char *) malloc(4*sizeof(char));`	Réserver une table pour 4 caractères
`str[0]='A'; str[1]='S'; str[2]='D';` `str[3]='\0';`	Nous remplissons le tableau avec la chaîne "ASD" en utilisant [ ] et le symbole '\0' pour indiquer la fin de la chaîne.
`*str='A'; *(str+1)='S'; *(str+2)='D';` `*(str+3)='\0';`	Nous remplissons le tableau avec la chaîne littérale "ASD" en utilisant l'opération de récupération * où *(str+i) ⇔ str[i]
`printf("String is %s\n  Address is %p\n",` `str, str);`	Pour afficher la chaîne et son adresse, où nous notons que & n'est pas utilisé car str est une adresse
`str = (char *) realloc(str,` `5*sizeof(char));`	Modification de la capacité du tableau de 4 à 5
`str[3]='2'; str[4]='\0';`	Nous remplissons les deux derniers caractères pour que la chaîne devienne "ASD2"
`*(str+3)='2'; *(str+4)='\0';`	
`printf("String is %s\n  New address is` `%p\n", str, str);`	Affiche la chaîne "ASD2" et sa nouvelle adresse

`free(str);`
`return 0;`
`}`
	Retourner la mémoire réservée

4.4. Pointeurs et matrices en C

Les matrices en C sont un tableau dont chaque élément est un tableau. Nous voulons créer une matrice M[3][4] avec 3 lignes et 4 colonnes.

Supposons que nous ayons 3 tableaux M0, M1, M2

```
float M0[4],M1[4],M2[4] ;
```

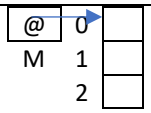
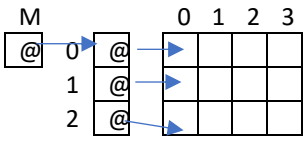
Ces tableaux peuvent être créés à l'aide de pointeurs

```
float *M0,*M1,*M2 ;
M0=(float *)malloc(4*sizeof(float));
M1=(float *)malloc(4*sizeof(float));
M2=(float *)malloc(4*sizeof(float));
```

Notons que M0, M1 et M2 sont tous du même type (float *), ils peuvent donc être remplacés par un tableau M de type (float *).

```
float * M[3] ;
for(int i=0;i<3;i++)
  M[i]=(float *)malloc(4*sizeof(float));
```

Maintenant, les pointeurs peuvent être utilisés pour créer la table M

C	mémoire	l'explication
`float **M ;`		Un pointeur M est déclaré de type float **
`M=(float**) malloc( 3*sizeof(float*));`		
		Le tableau M est créé, qui contient 3 éléments qui représentent le nombre de lignes, le type de chaque element est float *
`for(int i=0;i<3;i++)` `  M[i]=(float*) malloc(4*sizeof(float));`		
		Nous créons 3 tableaux, dont chacun représente une ligne dans la matrice. 4 est le nombre de colonnes et le type de chaque colonne est float. *(M+i) peut être utilisé à la place de M[i]

Tout élément de la matrice est accessible en utilisant [] ou en utilisant l'opérateur de récupération * où

M[i][j] ⇔ *(M[i]+j)

M[i][j] ⇔ *(*(M+i)+j)

en utilisant typedef

```
typedef float ** matrix;
typedef float * table;
matrix M ;
M=(matrix)malloc(3* sizeof(table));
for(int i=0;i<3;i++)
  M[i]=(table) malloc(4*sizeof(float));
```

Remarque : un tableau statique en C est une adresse mémoire constante qui ne peut pas être modifiée.

Exemple:

```
int *p,t[10];
```

`p=t;` Correct car t est l'adresse du premier élément

`t=p;` Non acceptée car t est une constante qui ne peut pas être modifiée.