

Programmation en C/C++

Série 8

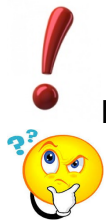
Les pointeurs

Passage de paramètres par adresse

Gestion de la mémoire

@ G.CANESI Le Cnam

Document pour un usage personnel uniquement



Important !

Difficile !

Objectifs

Maîtriser les adresses des variables, utiliser les pointeurs.

Conseil

Ce chapitre est difficile et important ; il faut prendre le temps de le comprendre puis de l'assimiler. De plus, les notions abordées dans ce chapitre seront reprises par la suite.

Introduction



Chaque variable est rangée à une adresse précise dans la mémoire vive de l'ordinateur. Une variable est donc caractérisée par son nom, son adresse et son contenu. Une adresse est généralement un entier long.

Un pointeur est une variable qui contient une adresse.

Quel est l'intérêt des pointeurs, à quoi peuvent-ils bien servir ? Les pointeurs vont permettre de travailler directement à une adresse mémoire donnée : on va pouvoir modifier ou lire directement le contenu à cette adresse. Les pointeurs vont permettre la modification du contenu des variables lors de l'appel de fonctions. Avec les fonctions, il n'est possible de renvoyer qu'une seule valeur ; grâce aux pointeurs on pourra contourner ce problème : il sera possible de modifier plusieurs valeurs auxquelles pourra ensuite accéder le programme appelant. Les pointeurs vont permettre également de gérer plus finement la mémoire. Par exemple, au lieu d'allouer de la mémoire pour un tableau de 100 entiers, même s'il n'est pas plein, on va pouvoir créer des structures dynamiques comme les listes chaînées qui vont permettre

d'allouer de la mémoire en fonction des besoins, supprimer ou ajouter des éléments, ce qui n'était pas possible avec les tableaux. Les listes chaînées font faire l'objet d'un chapitre important et difficile qui s'appuiera complètement sur celui-ci mais également sur ceux traitant des fonctions et de la récursivité.

Définition d'un pointeur



Un pointeur est une variable qui contient une adresse.

Exemples

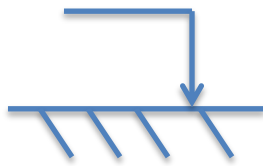
```
int *p;
```

On a défini un pointeur *p* qui aura en mémoire l'adresse d'un entier.

```
int *P1= NULL;
```

On a déclaré un pointeur *P1* qui aura en mémoire l'adresse d'un entier ; pour le moment le pointeur est "vide" : il ne stocke pas d'adresse : on dira qu'il ne pointe sur rien.

Pointeur NULL

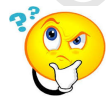


Affichage d'une adresse mémoire



Pour afficher l'adresse d'une variable ou le contenu d'un pointeur (il contient une adresse !) on utilisera :

- Le format entier long (%ld) pour un affichage en base 10 (cela génère un warning) ou
- Le format hexadécimal (%p) pour un affichage en base 16 (hexadécimal).



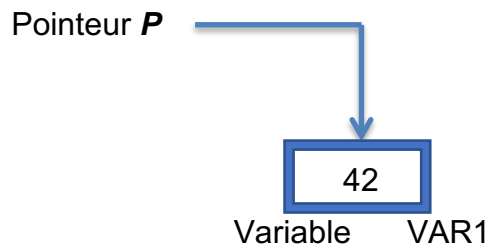
Faire "pointer" un pointeur sur une variable

Soit une variable *VAR1* stockant un entier. Créons un point *P* qui pointe sur cette variable :

```
int VAR1;
```

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

```
P = &VAR1;
```



Explications

Pour que *P* pointe sur la variable *VAR1* il faut que *P* ait en mémoire l'adresse de *VAR1* !

int VAR1;

On a déclaré une variable *VAR1* qui contiendra un entier. En fait, on a réservé un espace mémoire suffisant pour stocker un entier, à une adresse donnée. La variable *VAR1* connaît l'entier rangé à cette adresse.

*int *P = NULL;*

On déclare un pointeur nommé *P*. En fait, on a réservé un espace mémoire suffisant pour stocker une adresse où est rangé un entier. Attention ! Il faut toujours préciser vers quel type de variable pointe *P* (ici une variable de type entier).

P = &VAR1;

L'affectation se lit, comme toujours, de la droite vers la gauche : on recopie l'adresse de *VAR1* dans *P*. Autrement dit : à l'adresse de la variable *VAR1*, on fait pointer *P*. On dira que *P* pointe sur *VAR1*. *P* a en mémoire l'adresse où est stockée la variable *Var1*.

Conséquences

Toute modification du contenu de *VAR1* est vue par le pointeur *P*. Réciproquement, toute modification via le pointeur *P* affectera le contenu de *VAR1*.

Adresse d'une variable



Pour afficher l'adresse d'une variable on fait précéder son nom du symbole `&`.

Pour afficher une adresse on utilisera le format entier long (`%ld`) ou hexadécimal (`%p`).

Exemple

Soit une variable *VAR1* contenant un entier (*int VAR1*).

Pour afficher le contenu de *VAR1* : on écrit `printf("%d", VAR1)`

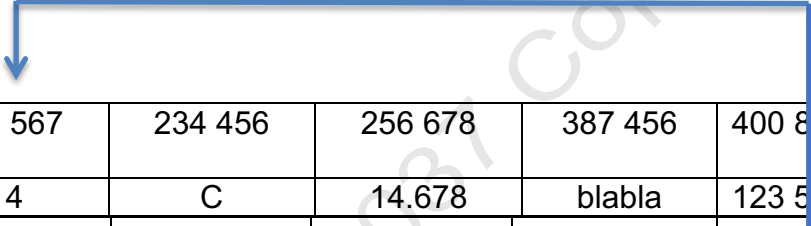
Pour afficher l'adresse de VAR1 : on écrit `printf("%p",&VAR1)` (%p pour afficher l'adresse en hexadécimal ou %ld pour afficher l'adresse en entier long mais cela génère un warning).



Remarque sur l'instruction `scanf()`

Lorsqu'on a utilisé précédemment l'instruction `scanf()`, on a utilisé les adresses des variables, sans le savoir. En effet, une ligne du type `scanf("%d", &Var)` va écrire l'entier saisi au clavier à l'adresse où est stockée la variable nommée `Var` (&Var).

Pointeur **P**



| | | | | | |
|---------------------|-------------|-------------|--------------|---------------|--|
| Adresse mémoire | 123 567 | 234 456 | 256 678 | 387 456 | 400 800 |
| Valeur | 14 | C | 14.678 | blabla | 123 567 |
| Nom de la variable | VAR1 | A | Ent1 | STR1 | P |
| Type de la variable | <i>int</i> | <i>char</i> | <i>float</i> | <i>char *</i> | <i>Int *</i> <i>Pointeur sur entier</i> |

Le pointeur `P` contient l'adresse 123567 où est rangée la variable `A`. Le contenu stocké à cette adresse est l'entier 14. On dira que `P` pointe sur `VAR1`.

Contenu stocké à une adresse donnée par un pointeur



Pour afficher le contenu stocké à une adresse mémorisée par un pointeur : on fait précéder le pointeur du symbole `*`.

Exemple

Soit un pointeur sur un entier `P1` (`int *P1`).

Pour afficher l'adresse pointée : on écrit `printf("%p", P1);`

Pour afficher le contenu stocké à cette adresse : on écrit `printf("%d", *P1);`



Exemple d'erreur 1

`int Alpha = 10;`

```
float *P;  
P = &Alpha;
```

On définit une variable *Alpha* qui est initialisée à 10 ; on crée un pointeur *P* sur des *float*. L'erreur est de faire pointer *P* sur la variable *Alpha* : il y a une erreur ici car *P* est un pointeur vers des *float* et non vers des entiers. On dit que "**les pointeurs sont typés**" en langage C.



Exemple d'erreur 2

```
int Alpha = 10;  
int *P;  
Alpha = *P;
```

Erreur car le pointeur *P* n'a pas été initialisé (il pointe sur rien) : on cherche à placer le contenu pointé par *P* (**P*) dans la variable *Alpha* or *P* ne pointe sur rien.



Exemple d'erreur 3

```
int Alpha = 10;  
int *P = NULL;  
Alpha = *P;
```

Même erreur car le pointeur *P* n'a pas été initialisé (*NULL*) : on cherche à placer le contenu pointé par *P* (**P*) dans la variable *Alpha* or *P* ne pointe sur rien.



Passage de paramètres par adresse aux fonctions

Exemple 1

Jusqu'à présent, nous avons passé des paramètres par valeur, aux fonctions : Reprenons le programme précédent :

```
// Fonction Somme  
int Somme (int VAR1, int VAR2)  
{  
    return (VAR1+VAR2);  
}
```

```
// programme principal  
int main()  
{  
    int a,b;  
    printf("Valeur de a ?");  
    scanf("%d", &a);  
    printf("Valeur de b ?");
```

```
scanf("%d", &b);
printf("La somme : %d", Somme(a,b));
return 0;
}
```

Au niveau du programme principal, on appelle la fonction *Somme* en lui "envoyant le contenu des variables *a* et *b*". Au niveau de cette fonction, le contenu de *a* est recopié dans *VAR1* et le contenu de *b* est recopié dans *VAR2*. La fonction *Somme* n'accède pas aux variables *a* et *b* ; elle ne peut donc modifier leur contenu.

Exemple 2

Voyons maintenant le programme suivant qui permute les contenus de deux variables :

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

// Fonction Permut
void Permut (int* VAR1, int* VAR2)
{
    int tampon;

    tampon=*VAR1;
    *VAR1=*VAR2;
    *VAR2=tampon;
}

// programme principal
int main()
{
    int A,B;
    printf("Valeur de a ?");
    scanf("%d", &A);
    printf("Valeur de b ?");
    scanf("%d", &B);

    Permut(&A,&B);

    printf("A vaut : %d",A);
    printf("B vaut : %d",B);

    return 0;
}
```



Explications

Le programme principal appelle la fonction *Permut* et lui envoie, en paramètres, les adresses des variables *A* et *B*. Ces 2 adresses sont recopiées respectivement dans

les pointeurs *VAR1* et *VAR2*. Ces deux pointeurs accèdent donc directement aux contenus des variables *A* et *B*, via ces adresses.

```
tampon=*VAR1;
```

On recopie dans *tampon* le contenu présent à l'adresse pointée par *VAR1*.

```
*VAR1=*VAR2;
```

On recopie, à l'adresse pointée par *VAR1*, le contenu présent à l'adresse pointée par *VAR2*.

```
*VAR2=tampon;
```

On recopie, à l'adresse pointée par *VAR2*, le contenu de *tampon*.

En revenant dans le programme principal, on affiche les contenus de *A* et *B* qui ont donc été modifiés.

Remarque

Le prototype de *Permut* est : `void Permut (int *, int *)`



Opérations sur des pointeurs

Exemple avec un pointeur sur entier :

```
int A = 5
```

```
int *P = Null;
```

```
int *Q = Null;
```

```
P = &A;
```

```
Q = P + 1;
```

- **P* correspond au contenu pointé par *P*, ici le contenu de *A*
- *P+1* correspond à l'adresse suivante : adresse en mémoire dans *P* décalée de 4 octets (décalage d'un entier), le pointeur *P* ne change donc pas ici. (On décale de 4 octets car un entier est codé sur 4 octets).
- *++P* on décale le pointeur *P* de 4 octets (car c'est un pointeur sur entier). Attention ! Danger ! Ici, l'adresse stockée par *P* a changé : *P* ne pointe plus au même endroit !
- *Q = P+1* ; *Q* a en mémoire l'adresse stockée par le pointeur *P* décalée de 4 octets. *Q* pointera donc 4 octets après *P*.
- *Q - P* ; affichera le nombre d'octets séparant les adresses pointées par les pointeurs *P* et *Q* ; ici 4.
- `printf ("%d" ;*P)` affiche le contenu pointé par *P*.

- `printf ("%d" ;*(P+1))` affiche le contenu pointé 4 octets après l'adresse en mémoire dans `P`.
- `printf ("%d" ;*(++P))` On décale de 4 octets l'adresse en mémoire dans `P` puis on affiche le nouveau contenu pointé par `P`.



Pointeurs sur des tableaux

Lorsque nous avons travaillé sur les fonctions, lorsque nous passons en paramètre un tableau nous écrivions : `int T [5]` cela signifiait que nous déclarions un tableau de 5 entiers. En fait, `T` est un pointeur sur le premier élément du tableau `T`. Donc `T` a en mémoire l'adresse de `T [0]` c'est-à-dire `&T [0]`, l'adresse où le début du tableau est stocké. C'est d'ailleurs pour cela que nous étions obligés d'envoyer en deuxième paramètre le nombre d'éléments du tableau aux fonctions manipulant des tableaux : On envoyait donc l'endroit où débutait le tableau (adresse) et le nombre de "cases" qu'il comportait.

Exemple 1

```
int T [5] = {10, 11, 12, 13, 14};
int *P;
P = T;
```

- `printf ("%d" ;*P)` affiche le contenu de `T [0]` c'est-à-dire 10. Car `P` pointe sur le premier élément du tableau `T`.
- `printf ("%d" ;*(P+3));` affiche le contenu de `T [3]` (décalage de 3 entiers), le quatrième élément du tableau `T` c'est-à-dire 13 ; le pointeur `P` n'a pas été modifié, il pointe toujours à l'adresse de `T [0]`.
- `T [n]` et `*(P+n)` où `n` est un entier, sont équivalents : On parle de l'élément `n+1` du tableau.
- `printf ("%d" ;*(++P));` affiche le contenu de `T [1]`, le deuxième élément du tableau `T` c'est-à-dire 11 ; Attention ! Ici, le pointeur `P` a été modifié : il pointe désormais à l'adresse de `T [1]`.

Exemple 2

On cherche, dans un premier temps, à afficher un tableau de 5 réels (1.1 , 1.2, 1.3 , 1.4, 1.5). Nous n'utiliserons pas directement les pointeurs.

Solution

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



```

int main() {
int i;
float T [5]={1.1,1.2,1.3,1.4,1.5};
for(i=0 ; i<5 ; i++) {
    printf("%f\t",T[i]); }

return 0;
}

```

Exemple 3

On cherche à réaliser la même chose mais en utilisant un pointeur *p* sur ce tableau...

Solution

```

#include<stdio.h>
#include<stdlib.h>
int main() {
int i;
float *p;

float T[5]={1.1,1.2,1.3,1.4,1.5};

p=T;

for(i=0 ; i<5 ; i++) {
    printf("%f\t",p[i]); }

return 0;
}

```

Remarque

p est un pointeur sur un réel (*float*).

p = *T* signifie que *p* pointe sur le tableau donc **sur le premier élément du tableau**.

On aurait pu écrire *p* = &*T*[0];

Autre solution...

```

#include<stdio.h>
#include<stdlib.h>

int main() {
int i;
float *p;

float T[5]={1.1,1.2,1.3,1.4,1.5};
p=T;
for(i=0 ; i<5 ; i++) {
    printf("%f\t",*p);
    p++;
}
}

```

```
return 0;
}
```



Pointeurs sur chaînes de caractères

En langage C, une chaîne de caractère (succession de lettres, chiffres ou symboles) est en fait un tableau de caractères dont le dernier élément est '\0' (fin de chaîne).

En fait, une chaîne de caractères est un tableau de caractères... Donc, pour les manipuler, on peut les traiter soit comme un tableau soit en utilisant les pointeurs puisque qu'à tableau correspond un pointeur sur le premier élément de ce dernier.

Les chaînes de caractères seront traitées dans un chapitre spécifique.

Exemple 1

Nous allons chercher à afficher le contenu de la chaîne nommée *ch* et à laquelle on a affecté 'Salut !', soient 7 caractères. Nous n'utiliserons pas les pointeurs sauf pour déclarer la chaîne (*char * ch*). Nous traiterons la chaîne comme un tableau de caractères.

Solution

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

```
int main() {
int i;
```

```
//déclaration de la chaîne ch
char *ch ;
```

```
//on affecte les 7 caractères suivants à ch :
ch = "Salut !";
```

```
for(i=0 ; ch[i]!='\0' ; i++) {
    printf("%c",ch[i]);
}
```

```
return 0;
}
```

Remarque

On affiche le tableau, caractère après caractère (%c).

Exemple 2

On cherche à réaliser les mêmes choses avec les notations des pointeurs plutôt que celles des tableaux...

Solution

```
#include<stdio.h>
#include<stdlib.h>
int main() {
    int i;
    char *ch ;
```

```
//on affecte les 7 caractères suivants à ch :
ch = "Salut !";
```

```
for(i=0 ; i<7 ; i++) {
    printf("%c",*ch);
    ch++;
}
```

```
return 0;
}
```

Remarque

À chaque tour de boucle, on affiche le contenu vers lequel pointe *ch* et ensuite on incrémente *ch* (on passe au caractère suivant).



Pointeurs sur fichiers

Un pointeur peut stocker l'adresse où est stocké un fichier, en mémoire de masse. On devra faire appel à un pointeur pour lire ou écrire dans un fichier texte.

Exemple

```
FILE* fic = NULL;
fic = fopen("Fichier1.txt","a");
```

- *FILE* fic = NULL;* On crée un pointeur sur fichier nommé *fic* ; ce pointeur aura donc en mémoire l'adresse d'un fichier (*FILE*) ; l'adresse est vide pour le moment : *fic* ne pointe sur rien.
- *fic = fopen("Fichier1.txt","a");* On associe le pointeur *fic* au fichier nommé *Fichier1.txt* ; on ouvre le fichier en mode "écriture à la fin du fichier". La fonction *fopen* retourne dans *fic* l'adresse du début de fichier, une fois ce dernier ouvert et *NULL* s'il ne peut pas être ouvert.

Les fichiers et pointeurs sur fichiers seront traités dans un chapitre spécifique.



Pointeurs sur pointeurs (hors programme)

Un pointeur est une variable qui possède en mémoire une adresse mémoire...

On peut donc créer un pointeur qui possède en mémoire l'adresse où est rangé un autre pointeur. Ce pointeur pointera donc sur un pointeur. On utilisera la double étoile. Exemple : Soit *P1* un pointeur sur un pointeur sur entier : *int **P1*

| pointeur de pointeur | |
|-----------------------------------|--|
| <code>float x, *pf, **ppf;</code> | <ul style="list-style-type: none"> - x est un float et pf est un pointeur sur un float - **ppf est un float - *ppf est un pointeur sur un float - ppf est un pointeur sur un pointeur sur un float |
| <code>int (**chose)[10];</code> | <ul style="list-style-type: none"> - (*chose)[10] est un int - (*chose) est un tableau de 10 int - *chose est un pointeur sur un tableau de 10 int - chose est un pointeur de pointeur sur un tableau de 10 int <p>Attention à bien mettre les parenthèses ici. Sans les parenthèses l'interprétation est différente comme nous pouvons le voir dans la suite.</p> |
| <code>int *pt[10];</code> | <ul style="list-style-type: none"> - *pt[10] est un int - pt[10] est un pointeur sur un int - pt est un tableau de 10 pointeurs sur un int |
| <code>int **ppt[10];</code> | <ul style="list-style-type: none"> - **ppt[10] est un int - *ppt[10] est un pointeur sur un int - ppt[10] est un pointeur sur un pointeur sur un int - ppt est un tableau de 10 pointeurs sur un pointeur sur un int |
| <code>int *f(float x);</code> | <ul style="list-style-type: none"> - *f(float x) est un int - f(float x) est un pointeur sur un int - f est une fonction recevant un float et retournant un pointeur sur un int |

Intérêt des pointeurs

Les pointeurs permettent de réaliser des choses jusqu'à présent impossibles :

- Accès direct et donc plus rapide aux contenus à partir des adresses,
- Modification des contenus en utilisant des fonctions,
- Modification des contenus sans utiliser des variables globales,
- Possibilité d'avoir plusieurs résultats à partir d'une fonction,
- Créer de nouvelles structures (listes chaînées, arbres...),
- Supprimer ou ajouter des éléments dans des listes (interdit avec les tableaux).



En résumé !

Pour une variable A

Supposons que la variable A soit rangée à l'adresse mémoire 150200

Supposons que le pointeur P soit rangé à l'adresse mémoire 250600

- A : on parle du contenu de la variable A (ici 12)
- $\&A$: on parle de l'adresse de la variable A (ici 150200)

Pour un pointeur P

- $\text{int} * P$ on crée un pointeur P sur un entier
- P : on parle de l'adresse où pointe P (ici 150200) (le contenu de P)
- $*P$: on parle du contenu présent là où pointe P (ici 12 : c'est aussi le contenu de la variable A)
- $\&P$: on parle de l'adresse du pointeur P (ici 250600)

Nous reviendrons sur les pointeurs lors des chapitres suivants, en particulier ceux sur les chaînes de caractères et les listes chaînées.

Gestion de la mémoire

Au lieu d'allouer de la mémoire automatiquement lors de la déclaration de variables on peut allouer de la mémoire dynamiquement.

En C, il existe deux mots clés liés à la gestion dynamique de la mémoire :

- **malloc()** qui alloue de la mémoire,
- **free()** qui libère la mémoire occupée préalablement allouée avec **malloc()**.



Exercice de cours 1

Qu'affiche le programme suivant :

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

int main()
{
    // Déclaration d'un pointeur P sur Entier
    int* P = NULL;
    printf("Adresse de P avant malloc : %p \n", &P);
    printf("Adresse pointée par P avant malloc : %p \n\n", P);

    // on ne peut pas écrire cette ligne là pour le moment : printf("Contenu pointé par P
    //avant malloc: %d \n\n", *P);

    // Allocation de mémoire
    P = malloc(sizeof(int));
    printf("Adresse de P après malloc : %p \n", &P);
    printf("Adresse pointée par P après malloc: %p \n", P);
    printf("Contenu pointé par P après malloc: %d \n\n", *P);
```

```

if (P != NULL)
{
printf("Quel est votre prix ? ");
scanf("%d", P);

printf("Avant libération de la mémoire : \n\n");

printf("Adresse de P : %p \n", &P);
printf("Adresse pointée par P après saisie : %p \n", P);
printf("Contenu pointé par P après saisie : %d \n\n", *P);
}

// Libération de mémoire
free(P);
printf("Après libération de la mémoire : \n\n");
printf("Adresse de P : %p \n", &P);
printf("Adresse pointée par P après libération : %p \n", P);
printf("Contenu pointé par P après libération : %d \n", *P);

return 0;
}

```

```

Adresse de P avant malloc : 0x7ffcc61beec8
Adresse pointée par P avant malloc : (nil)

Adresse de P après malloc : 0x7ffcc61beec8
Adresse pointée par P après malloc: 0x15d3010
Contenu pointé par P après malloc: 0

Quel est votre prix ? 12
Avant libération de la mémoire :

Adresse de P : 0x7ffcc61beec8
Adresse pointée par P après saisie : 0x15d3010
Contenu pointé par P après saisie : 12

Après libération de la mémoire :

Adresse de P : 0x7ffcc61beec8
Adresse pointée par P après libération : 0x15d3010
Contenu pointé par P après libération : 0

```

Affichages générés par le programme précédent

Remarques

- On constate qu'après la déclaration du pointeur une adresse a bien été créée pour celui-ci mais qu'il n'a aucune valeur en mémoire (nil ; NULL) : il ne pointe sur rien.
- Après le *malloc()* le pointeur pointe sur une nouvelle zone mémoire mais rien n'a été enregistré à cet endroit et à cet instant.
- On peut écrire : *scanf("%d", P);* car cela signifie que l'entier saisi au clavier sera rangé là où pointe *P*. Pas besoin de "&" car *P* doit avoir une adresse en mémoire : c'est un pointeur !
- Dès que l'on n'a plus besoin du pointeur *P* on libère la mémoire en écrivant *free(P)*



Exemple de cours 2

Qu'affiche le programme suivant :

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

int main()
{
    int Nb = 0;
    int i ;
    int* Prix = NULL;

    printf("Nombre de prix à saisir ? ");
    scanf("%d", &Nb);

    if (Nb > 0)
    {
        Prix = malloc (Nb * sizeof(int));
        if (Prix != NULL)
        {
            for (i = 0 ; i < Nb ; i++)
            {
                printf("Prix numéro %d ? \n ", i + 1);
                scanf("%d", &Prix[i]);
            }

            printf("Prix saisis :\n");

            for (i = 0 ; i < Nb ; i++)
                printf("%d \n", Prix[i]);

            free(Prix);
        }
    }
    return 0;
```

}

Corrigé

- En fait le pointeur *Prix* joue le rôle d'un tableau de *Nb* valeurs entières.
- Lorsque l'on crée un tableau on crée un pointeur qui pointe sur son premier élément.
- Du coup *Prix = malloc(Nb * sizeof(int));* alloue de la mémoire pour *Nb* entiers et fait pointer *Prix* sur le premier d'entre eux. *Prix* est donc bien un tableau et l'expression *Prix[i]* n'est donc pas interdite !

```
Nombre de prix à saisir ? 4
Prix numéro 1 ?
123
Prix numéro 2 ?
134
Prix numéro 3 ?
144
Prix numéro 4 ?
155
Prix saisis :
123
134
144
155
```

Affichages générés par le programme précédent

EXERCICES

Exercice 1

Énoncé

Créer une variable *a* qui stocke l'entier 44. Créer un pointeur *p* qui pointe sur la variable *a*. Écrire le programme qui affiche successivement :

- Le contenu de *a*,
- L'adresse de *a*,
- Le contenu du pointeur *p*,
- L'adresse du pointeur *p*,
- Le contenu présent à l'adresse que contient le pointeur *p*.

Corrigé

```
contenu de a : 44
adresse de a : 140734744370104
contenu du pointeur p : 140734744370104
adresse du pointeur p : 1550982064
le contenu present a l'adresse ou pointe le pointeur p est : 44
```

Affichage généré à l'exécution

Programme source

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

int main () {
    int a =44;

    // pointeur p qui a en memoire l'adresse de a
    int *p= &a;

    printf("Contenu de a :%d \n",a);
    printf ("Adresse de a : %ld \n", &a);
    printf("Contenu du pointeur p (adresse où il pointe) :%ld \n",p);
    printf ("Adresse du pointeur p : %ld \n", &p);
    printf ("Contenu present a l'adresse que contient le pointeur p : %d \n", *p);

    return 0;
}
```

Rq

Le contenu du pointeur est bien une adresse.
Le pointeur est stocké lui-même à une adresse.

Exercice 2

Énoncé

Créer une variable *a* qui stocke l'entier 44 puis dans l'ordre :

- Créer une variable *b* de type entier.
- Créer un pointeur *p* qui pointe sur la variable *a*.
- Affecter à la variable *b* le contenu présent à l'adresse pointée par *p*.
- Afficher le contenu de *b*, pour finir. Qu'affiche le programme ?

Corrigé

```
#include<stdio.h>
#include<stdlib.h>
int main () {
    int a =44; int b;

    //pointeur p qui a en memoire l'adresse de a
    int *p= &a;

    // le contenu stocké à l'adresse pointé par p est stockée dans b
    b=*p;
    printf (" le contenu de la variable b vaut maintenant : %d\n ",b);
    return 0;
}
```

Le programme affiche 44.

Exercice 3

Énoncé

Qu'affiche le programme suivant ? Pourquoi ?

```
#include<stdio.h>
#include<stdlib.h>
int main () {
    int a =44;
    int *P= NULL;

    P=&a;
    a=45;

    printf (" le contenu de a vaut maintenant : %d\n ",*P);
    return 0;
}
```

Corrigé

On affecte l'adresse de *a* au pointeur *P* (*P*=&*a*) donc *P* pointe sur *a*. On affecte ensuite la valeur 45 à la variable *a*. Donc, si on affiche le contenu à l'adresse où pointe *P*, il s'affichera 45.

Exercice 4

Énoncé

Qu'affiche le programme suivant ? Pourquoi ?

```
#include<stdio.h>
#include<stdlib.h>
int main () {
    int a =44; int b;
    int *P= NULL;

    P=&a;
    a=45;
    *P=46;

    printf (" le contenu de a vaut maintenant : %d\n ",a);
    return 0;
}
```

Corrigé

On affecte l'adresse de *a* au pointeur *P* ($P=&a$) donc *P* pointe sur *a*. On affecte ensuite la valeur 45 à la variable *a* puis on affecte la valeur 46 à l'adresse où pointe *P*. Donc, le contenu de *a* est modifié puisqu'il s'agit du contenu à la même adresse que *P* ! Si on affiche le contenu de la variable *a*, il s'affichera 46.

Exercice 5

Énoncé

On reprendra l'exemple du cours où un programme faisait appel à une fonction Somme en lui passant 2 nombres entiers en paramètres. Cette fois, on enverra les adresses des deux nombres préalablement saisis au niveau du *main()* ; la fonction retournera le résultat de la somme de ces 2 nombres.

Contrainte : utilise une prototype pour la fonction.

Corrigé

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

int Somme(int *, int *);

int main() {
    int a, b;

    printf("Saisir a : ");
    scanf("%d", &a);
    printf("Saisir b : ");
    scanf("%d", &b);

    printf("%d + %d = %d\n", a, b, Somme(&a, &b));
}
```

```
}
```

```
int Somme(int *varA, int *varB) {  
    return (*varA + *varB);  
}
```



Exercice 6

Énoncé

On reprendra ici l'exemple de cours sans le regarder :

Écrire un programme principal qui demande de saisir 2 entiers dans des variables et appelle ensuite une fonction qui va permuter les contenus de ces 2 variables. On n'utilisera pas de variable globale mais des pointeurs.

Corrigé

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

```
// Fonction Permut  
void Permut (int* VAR1, int* VAR2)  
{  
    int tampon;
```

```
    tampon=*VAR1;  
    *VAR1=*VAR2;  
    *VAR2=tampon;  
}
```

```
// programme principal  
int main()  
{  
    int A,B;  
    printf("Valeur de a ?");  
    scanf("%d", &A);  
    printf("Valeur de b ?");  
    scanf("%d", &B);  
    Permut(&A,&B);  
    printf("A vaut : %d\n ",A);  
    printf("B vaut : %d\n ",B);  
  
    return 0;  
}
```



Exercice 7

Énoncé

Qu'affiche le programme suivant si on saisit successivement 1 et 5 ? Pourquoi ?

Faire un schéma pour comprendre les contenus des différents pointeurs ou variables.

Bien réfléchir, la réponse n'est pas celle que l'on croit !!

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

```
// Fonction Permut
void Permut (int* VAR1, int* VAR2)
{
    int* Tampon;
```

```
    Tampon=VAR1;
    VAR1=VAR2;
    VAR2=tampon;
}
```

```
// programme principal
int main()
{
    int A,B;
    printf("Valeur de A ?");
    scanf("%d", &A);
    printf("Valeur de B ?");
    scanf("%d", &B);
    Permut(&A,&B);
    printf("A vaut : %d\n ",A);
    printf("B vaut : %d\n ",B);

    return 0;
}
```

Corrigé

```
Valeur de a ?1
Valeur de b ?5
A vaut : 1
B vaut : 5
```

Il s'affichera encore 1 et 5 !!

On a tout simplement et uniquement permuté les adresses en mémoire des pointeurs VAR1 et VAR2.

Le contenu pointé par VAR1 vaut certes 5 et le contenu pointé par VAR2 vaut 1 mais on affiche les contenus de A et B : les contenus aux adresses des variables A et B n'ont pas été modifiés !!



Exercice 8

Énoncé

Écrire le programme qui affecte la valeur 10 à une variable x puis envoie l'adresse de x à une fonction nommée "maFonction" qui va modifier le contenu de x à la valeur 5. Le programme devra afficher les données comme suit :

```
L'adresse de x est : 140734726032312
Contenu de x avant l'appel de la fonction : 10
Valeur contenue a l'adresse ou pointe P2 avant modif dans la fonction : 10
Valeur contenue apres modif a l'adresse ou pointe P2 dans la fonction : 5
Valeur de x dans le main et apres l'appel de fonction : 5
L'adresse de x apres appel de la fonction : 140734726032312
```

Corrigé

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

// prototype de la fonction
void maFonction(int *);

int main() {

    int x=10;

    printf("L'adresse de x est : %ld\n", &x);
    printf ("Contenu de x avant l'appel de la fonction : %d\n", x);

    // Envoi de l'adresse de x à maFonction
    maFonction(&x);

    // Affichage de la valeur de x modifiée et qui vaut maintenant 5 !
    printf("Valeur de x dans le main et apres l'appel de fonction : %d\n", x);
    printf("L'adresse de x apres appel de la fonction : %ld\n", &x);
    return 0;
}

void maFonction(int *P_2) {
    printf("Valeur contenue a l'adresse ou pointe P2 avant modif dans la fonction : %d\n",
    *P_2);

    // Affectation de la valeur 5 à l'adresse pointée par P_2
    *P_2 = 5;

    printf("Valeur contenue apres modif a l'adresse ou pointe P2 dans la fonction : %d\n",
    *P_2);
}
```

Exercice 9

Énoncé

Créer un tableau de 5 nombres réels. Écrire une fonction *Affiche* qui affichera les éléments du tableau grâce à un pointeur que l'on déplace d'élément en élément.

Corrigé

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

```
void Affiche (float T [] , int Taille_Tab) {
    int i;
    float *P = NULL; // on déclare un pointeur sur des réels
```

```
    for(i=0; i<Taille_Tab; i++) {
        P= &T [i]; // on fait pointer P sur le contenu de T [i]

        // on affiche le contenu présent où pointe P :
        printf("Element %d : : %.2f \n", (i+1), *P);
    }
```

```
    return;
}
```

```
int main() {
    int i;
    float T [5];

    for(i=0 ; i<5 ; i++) {
        printf("Saisir le reel %d : \n ",(i+1));
        scanf("%f",&T[i]); }
    Affiche (T, 5);
    return 0;
}
```



Exercice 10

Énoncé

Créer un tableau de 5 nombres réels. Écrire une fonction qui permet de calculer la valeur la plus petite et la valeur la plus grande de ce tableau.

Contrainte : utiliser 2 pointeurs.

Corrigé

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

```
void Max_Min (float T [] , int Taille_Tab, float *Min, float *Max) {
```

```

int i;

for(i=0; i<Taille_Tab; i++) {
    if (T [i]>*Max)
        *Max = T [i];
    if (T [i]<*Min)
        *Min = T [i];
}
return;
}

int main() {
int i;
float Min; float Max;
float T [5];

for(i=0;i<5;i++) {
    printf("Saisir le reel %d : \n ",(i+1));
    scanf("%f",&T[i]);
}
Min = Max = T[0];

Max_Min (T, 5, &Min, &Max);

printf("Valeur mini : %f\n",Min);
printf("Valeur max : %f\n",Max);

return 0;
}

```

Remarque

On envoie les adresses de *Min* et *Max* à la fonction qui recherche les 2 valeurs extrêmes du tableau ; il s'agit d'un passage de paramètres par adresse. La fonction *Max_Min* peut donc modifier les contenus présents à ces 2 adresses. Au niveau du programme principal on récupérera donc les contenus présents à ces deux adresses. La fonction *Max_Min* reçoit 4 arguments : le tableau, le nombre d'éléments du tableau et les adresses de *Min* et *Max*. On n'aurait pas pu traiter cet exercice avec un passage de paramètres par valeur : une seule valeur aurait pu être retournée !

Nous reviendrons sur les pointeurs et les utiliserons lors des séries suivantes.