Chapitre 11

Les Pointeurs

Une des particularités extrêmement puissante du langage C est qu'il propose la notion de "pointeur". La notion de pointeur est souvent assez mal comprise par les débutants en programmation, et, à cause de cela, quelque peu redoutée.
Cela dit, utilisés judicieusement, les pointeurs rajoutent au langage C un grand nombre de fonctionnalités dont on ne peut pas se passer à partir du moment où on sait comment les utiliser, et qu'on commence à développer des programmes de taille non négligeable.
Au cours de ce chapitre, nous commencerons par voir ce que sont les pointeurs. Ensuite, nous verrons comment en créer et comment les utiliser pour quelques cas simples, et nous finirons par quelques exemples montrant leur utilité.

I:\ Qu'est-ce qu'un pointeur ?

Lorsque l'on jette un coup d'oeil dans un livre traitant de programmation en C au chapitre abordant le sujet des pointeurs (ce que je vous conseille de faire, naturellement), on constate qu'un pointeur est souvent représenté par une case et une flèche, qui indique une autre case, un peu comme ceci :

Si un jour, vous vous promenez dans la rue, que vous passez devant deux boulangeries côtes à côtes, et qu'un passant vous demande laquelle des deux fait le meilleur pain, vous allez en indiquer une des deux du doigt. Vous serez alors un pointeur, qui pointe vers l'une des deux boulangeries.

Pour parler en termes de langage informatique, sachez que chaque case mémoire peut être désigné par son adresse, c'est-à-dire, par son emplacement.
Un pointeur, tout simplement, est une case mémoire, qui contient l'adresse d'une autre case mémoire. Naturellement, l'adresse contenue par le pointeur peut changer, et pointer, selon le moment, sur une autre case mémoire.
De la sorte, en consultant une seule case mémoire, le pointeur, on peut, selon les circonstances, obtenir l'adresse d'autres cases mémoires nous intéressant.
En langage C, ces "cases mémoire" sont généralement appelées "variables". Un pointeur est donc une variable d'un type particulier, destinée à contenir non pas une valeur, mais l'adresse d'une autre variable.

Par exemple, si on a un pointeur nommé A, une variable entière, nommée X, qui contient la valeur X, et que A pointe sur X, on aura tendance à représenter cela de la sorte :

Dans la suite de ce chapitre, nous verrons comment ceci se représente en langage C, et quelle utilité cela peut avoir.

II:\ Déclaration d'un pointeur

Après cette introduction, nous allons voir comment, en langage C, déclarer des pointeurs.

Tout d'abord, il est à noter qu'il n'existe pas, en langage C, de "type pointeur", au sens où il y a un type int, un type float, ou d'autres types de ce genre.
Par contre, lorsque l'on crée une variable jouant le rôle de pointeur, il nous faut indiquer au compilateur vers quel type de données elle pourra pointer.
Ainsi, de la même façon qu'un float et un short sont différents, un pointeur déclaré comme pouvant pointer sur un short ne pourra pas pointer sur un float ; la réciproque étant, bien entendue, vraie ; et cela s'applique à tout type de variable.

Pour déclarer un pointeur sur une donnée de type donné, on procède comme si on voulait déclarer une variable de ce type là, mais en précédant le nom de celle-ci d'une étoile, comme ceci :

```
TYPE *nom_du_pointeur;
```

Par exemple, pour déclarer un pointeur, nommé 'a', qui puisse pointer sur un short, et un pointeur nommé 'b', qui puisse pointer sur un float, on écrira :

```
short *a;
float *b;
```

Le C permettant de mettre des espaces un peu où on le souhaite, il est naturellement possible d'écrire ceci :

```
short* a;
```

Cela dit, je vous conseille de préférer la première écriture que je vous ai proposé, à savoir, d'accoler l'étoile au nom de la variable pointeur.
En effet, lorsque vous serez amenés à déclarer plusieurs variables de type pointeur, il vous faudra vous souvenir que l'étoile indiquant que la variable que vous déclarez est un pointeur ne porte que sur la variable qu'elle précède.
Ainsi, si vous utilisez une des deux syntaxes suivantes :

```
short *a, b, c;   // seul a sera un pointeur !
short* a, b, c;   // Même chose !
```

Seul 'a' sera un pointeur sur une donnée de type short. b et c, en revanche seront des variables entières de type short, comme nous en avons utilisé dans les chapitres précédents.
Si vous voulez déclarer plusieurs pointeurs, il vous faut, j'insiste, précéder chacun de leurs noms par une étoile, comme ceci :

```
short *a, *b, *c;
```

Je vous conseille donc de prendre l'habitude, lorsque vous déclarez une variable de type pointeur, de toujours placer l'étoile immédiatement avant son nom, afin de ne pas croire que celle-ci affecte plus d'une variable, et, en même temps, afin de rendre votre code plus lisible.

III:\ Utilisation de base des pointeurs

Maintenant que nous savons comment déclarer des pointeurs, voyons comment les utiliser. Nous commencerons par voir comment faire pour obtenir l'adresse d'une variable, puis nous étudierons la méthode à suivre pour accéder à la donnée pointée par un pointeur.

A: Adresse d'une variable

Le C fournit un opérateur qui permet de récupérer l'adresse d'une variable, de façon à ce qu'on puisse la stocker dans un pointeur, qui pointera alors sur la variable donnée.
Il s'agit de l'opérateur unaire &, qui se se place avant le nom de la variable dont on souhaite obtenir l'adresse.

Par exemple, pour obtenir l'adresse de la variable a, on utilisera ceci :

```
&a
```

Maintenant que l'on sait comment obtenir l'adresse d'une variable, nous allons pouvoir mémoriser cette adresse dans un pointeur déclaré comme pouvant pointer sur ce type de variable, afin qu'il pointe effectivement vers quelque chose.
Pour cela, tout naturellement, nous utilisons l'opérateur d'affectation =, que nous avons déjà eu l'occasion d'étudier précédemment.
Par exemple, déclarons une variable de type float, et un pointeur sur float, affectons une valeur à notre première variable, et faisons pointer le pointeur sur celle-ci ; cela se fera de la manière suivante :

```
float b;
float *p2;

b = 3.14156;
p2 = &b;
```

Nous avons déjà expliqué que, tant qu'elle n'a pas été initialisée (c'est-à-dire, tant que nous ne lui avons pas encore affecté de valeur), une variable contient une valeur indéterminée, ce qui revient à dire qu'elle contient absolument n'importe quoi.
Il en est exactement de même pour les pointeurs, qui, finalement, ne sont rien de plus que des variables un peu particulières : tant qu'on ne leur a rien affecté, ils pointent sur une zone mémoire indéterminée. Prenez garde, essayer d'utiliser un pointeur qui n'a pas encore été initialisé et qui ne pointe donc vers "rien" est une source de bugs que l'on retrouve fréquemment, et qui, dès que le programme grossit un peu, est difficile à localiser.

Lorsque l'on utilise l'opérateur & en tant qu'opérateur unaire sur une variable, on dit qu'on référence cette variable, & étant alors l'opérateur de référence.
Il est à noter que cet opérateur ne peut s'appliquer qu'à des entités présentes en mémoire, c'est-à-dire, aux variables, et aux éléments de tableaux, que nous étudierons au chapitre suivant. Il ne peut, en particulier, pas s'appliquer à une expression, une constante, ou une variable déclarée comme register.

B: Valeur pointée

A présent, puisque nous savons comment obtenir un pointeur pointant sur une donnée, nous allons voir comment faire pour accéder à celle-ci, via le pointeur.

1: Valeur pointée, en C

Pour cela aussi, le C nous fournit un opérateur : l'étoile, que nous avions déjà utilisé pour déclarer un pointeur.
Pour obtenir l'adresse d'une variable, nous utilisions l'opérateur unaire & placé devant la nom de la variable dont nous souhaitions connaître l'adresse. Pour accéder à la valeur pointée par un pointeur, nous utiliserons l'opérateur * de la même façon, devant le nom du pointeur.

En utilisant la syntaxe *pointeur, nous avons accès à la donnée, aussi bien en lecture qu'en écriture, exactement comme si nous avions travaillé directement sur la variable pointée.
Par exemple, supposons que nous avons déclaré une variable de type entier, et un pointeur sur entier, de la façon suivante :

```
short a;
short *p;

a = 10;
p = &a;   // p pointe maintenant sur la variable a.
```

Nous pourrons mémoriser une valeur à l'adresse de la variable pointée par p, c'est-à-dire la variable a, de la façon suivante :

```
*p = 150;
```

Si nous essayons d'afficher la valeur contenue dans la variable a, nous constaterons que celle-ci est à présent 150, preuve que le pointeur p pointait bien vers a, et que nous pouvons y accéder de cette manière.
Et, naturellement, nous aurons accès à cette valeur en lecture, comme ceci par exemple :

```
printf("*p=%d\n", *p);
```

Lorsque l'on applique l'opérateur * à un pointeur, on dit qu'on déréférence ce pointeur, l'opérateur * étant alors l'opérateur de déréférence, aussi parfois appelé opérateur d'indirection.

Comme on pourrait s'en douter, lorsque l'on déclare un pointeur comme pouvant pointer sur des données d'un certain type, la valeur obtenue par déréférencement de ce pointeur est du type pointé.
Par exemple, avec un pointeur p déclaré comme pointant sur des données de type short, *p sera de type short.

2: Exemple illustré

Nous allons maintenant, en quelques schémas, illustrer ce que nous venons de présenter.
A chaque fois, nous ferons suivre la portion de code C par le schéma, et un bref commentaire décrivant ce que nous avons fait.

Tout d'abord, nous déclarons une variable de type short, nommée a, et un pointeur pouvant pointer sur des short :

```
short a;
short *p;
```

Ensuite, nous initialisons la variable a, en lui affectant la valeur 10 :

```
a = 10;
```

Puis nous initialisons le pointeur p, en lui affectant l'adresse de la variable a :

```
p = &a;
```

Et enfin, nous affectons la valeur 150 à la zone mémoire pointée par le pointeur p, c'est-à-dire, étant donné ce que nous avons effectué juste au-dessus, à la variable a :

```
*p = 150;
```

C: Pointeur NULL

Le standard C défini la constante NULL, qui correspond à l'adresse 0.
Par convention, l'adresse NULL étant considérée comme "impossible", cette valeur est donc considérée comme permettant de coder une erreur, ou le fait que le pointeur ne pointe sur rien.
Notamment, lorsque nous utilisons des fonctions retournant un pointeur, il est fréquent que celles-ci soient conçues de manière à retourner NULL en cas d'erreur.

Il est à noter que l'écriture vers un pointeur NULL est interdite, et causera un plantage de votre programme. Lorsqu'une fonction est susceptible de retourner NULL, ce qui est généralement indiqué dans sa documentation, il vous faut donc impérativement vérifier, avant d'utiliser le pointeur retourné, que celui-ci n'est pas NULL.

Lorsque l'on utilise la valeur NULL dans un contexte où on attendrait vrai ou faux, celle-ci est évaluée comme fausse.
Cela permet notamment des écritures comme celle-ci :

```
#include <tigcclib.h>

short *p;

// Travail avec p ; en particulier, on le fait pointer sur quelque chose.

if(!p)
{
     printf("p est NULL");
}
```

Au lieu de :

```
short *p;

// Travail avec p ; en particulier, on le fait pointer sur quelque chose.

if(p == NULL)
{
     printf("p est NULL");
}
```

Certes, cela ne fait que quelques caractères de moins dans le premier cas... Mais les programmeurs C ayant tendance à apprécier les formes concises, ils auront généralement tendance à éviter la seconde écriture.
En effet, pourquoi tester si "(p est NULL) est vrai ?", alors qu'il suffit de tester si "p est faux ?" ?

D: Exemple complet

Pour finir avec ces deux opérateurs, voici un petit exemple complet les mettant en oeuvre :

```
#include <tigcclib.h>

void _main(void)
{
    short a;
    short *p;

    clrscr();

    a = 150;
    p = &a;

    printf("a=%d ; *p=%d\n", a, *p);

    a += 10;
    printf("a=%d ; *p=%d\n", a, *p);

    *p += 40;
    printf("a=%d ; *p=%d\n", a, *p);

    ngetchx();
}
```

Exemple Complet

Ce petit programme affecte la valeur 150 à une variable entière, et déclare un pointeur sur celle-ci. Ensuite, nous ajoutons 10 à notre variable, puis 40, en y accédant, la seconde fois, par l'intermédiaire du pointeur.
Entre chaque opération, nous affichons la valeur de la variable, et la valeur pointée par le pointeur, ce qui nous permet de constater que nos manipulations ont le même effet, que nous accédions à la variable directement, ou par un pointeur pointant sur elle.

E: Résumé des priorités d'opérateurs

Exactement comme nous l'avons fait à chaque fois que nous avons vu de nouveaux opérateurs, nous allons, puisque nous venons d'en étudier deux nouveaux, dresser un tableau résumant leurs priorités respectives. Tout comme pour les tableaux des chapitres précédents, la ligne la plus haute du tableau regroupe les opérateurs les plus prioritaire, et les niveaux de priorité diminuent au fur et à mesure que l'on descend dans le tableau. Lorsque plusieurs opérateurs sont sur la même ligne, cela signifie qu'ils ont le même niveau de priorité.

Opérateurs, du plus prioritaire au moins prioritaire

```
()
! ~ ++ -- + - * & sizeof
* / %
+ -
<< >>
< <= > >=
== !=
&
^
|
&&
||
?:
= += -= *= /= %= &= ^= |= <<= >>=
```

Notez que les opérateurs + et - unaires présentés sur la seconde ligne sont plus prioritaires que leurs formes binaires présentes sur la quatrième ligne. Il en va de même pour les opérateurs & et * unaires que nous venons d'étudier, présents en seconde ligne, par rapport à leurs formes binaires vues précédemment, qui sont en troisième ligne.

IV:\ Quelques exemples d'utilisation

Pour terminer ce chapitre, nous allons dire quelques mots sur l'arithmétique de pointeurs, que nous utiliserons dès le chapitre prochain, et nous verrons un cas où nous n'avons pas d'autre solution que l'emploi de pointeurs.

A: Arithmétique de pointeurs

On appelle "arithmétique de pointeurs" le fait d'appliquer des opérateurs mathématiques sur des pointeurs.
Un pointeur contenant, en guise de valeur, une adresse, faire de l'arithmétique de pointeurs revient à effectuer des calculs portant sur des adresses mémoire.
Nous commencerons par constater que nous pouvons ajouter des entiers à des pointeurs, et nous finirons en parlant brièvement de la soustraction de pointeurs.

1: Ajouter un entier à un pointeur

Lorsque l'on dispose d'un pointeur sur un emplacement mémoire, il est possible, en lui ajoutant une valeur entière, de le faire pointer sur un des emplacements mémoire qui suit (ou qui précéde, si la valeur que l'on ajoute est négative).
Cela n'est pas extrêmement utile à savoir pour l'instant, mais nous utiliserons cette propriété des pointeurs dès le prochain chapitre, où nous verrons comment demander au compilateur de réserver des emplacements mémoire successifs.

Il est à noter que, lorsqu'on ajoute 1 à un pointeur, l'adresse sur laquelle il pointe n'est pas incrémenté de 1, mais du nombre d'octets correspondant au type pointé, c'est à dire, à 1*sizeof (TYPE).
Ainsi, si on incrémente de 1 un pointeur déclaré comme pointant sur des short, l'adresse ne sera pas incrémentée de 1, mais de 2, puisque les short sont codés, sur nos machines, sur deux octets.
Cela explique pourquoi, lorsque l'on veut travailler directement avec la mémoire, octets par octets, on utilise généralement des pointeurs déclarés comme pouvant pointer sur des char, qui sont codés sur un octet.

2: Soustraction de pointeurs

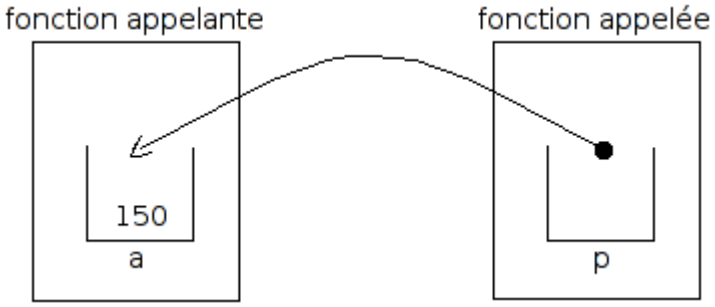
Rapidement, pour mémoire, notons qu'il est possible de soustraire des pointeurs, afin, par exemple, de savoir combien d'emplacements mémoire les séparent, mais c'est la seule opération que l'on peut effectuer sur deux pointeurs.
En particulier, il n'est pas possible d'en additionner, multiplier, diviser... Et, de toute façon, cela ne serait guère utile, après tout.

B: Pointeurs en paramètres de fonctions

Nous avons eu l'occasion de voir, lorsque nous travaillions sur les fonctions, au chapitre précédent, que leurs paramètres étaient passés par valeurs, et que cela empêchait les fonctions de modifier des variables appartenant à la portion de code appelante.
Nous avions dit que les pointeurs nous permettraient de trouver une solution à ce problème. C'est ce dont nous allons à présent discuter.

1: Un peu de théorie

Nous ne pouvons pas changer le fait que le passage de paramètres se fait par copie ; c'est fixé par le standard C, et il ne peut en être autrement. Cela dit, si plutôt que de passer en paramètre à la fonction une copie de la variable que nous voulons qu'elle puisse modifier, nous lui passons un pointeur sur celle-ci, elle aura accès, par le biais de ce pointeur, à la variable déclarée dans la fonction appelante. Nous pourrions représenter cette idée par un schéma tel celui-ci :



Dans notre fonction appelante, nous avons déclaré une variable nommée a, et nous avons passé à la fonction appelé l'adresse de cette variable. De la sorte, la fonction appelée a accès à l'emplacement mémoire correspondant à la variable a.

2: Un exemple

Etant donné que les pointeurs font parti des éléments du langage C qu'il est quasiment impossible de comprendre sans la pratique, nous allons reprendre l'exemple que nous avions utilisé en fin de chapitre précédent, à savoir, l'écriture d'une fonction dont le rôle est d'échanger les valeurs des deux variables passées en paramètres.
Nous avions vu que, puisque le C passe les paramètres par copie, cela n'était pas possible, avec les connaissances dont nous disposions avant de passer à ce chapitre...
Nous allons donc maintenant utiliser ce que nous venons d'apprendre, pour résoudre ce problème.
Voici le code-source d'un exemple complet, afin que vous puissiez voir par vous-même comment cela fonctionne :

```
#include <tigcclib.h>

void echange(short *a, short *b);

void _main(void)
{
    short a = 10;
    short b = 20;

    clrscr();

    printf("AVANT:   a=%d ;   b=%d\n", a, b);

    echange(&a, &b);

    printf("APRES:   a=%d ;   b=%d\n", a, b);

    ngetchx();
}


void echange(short *pa, short *pb)
{
    short temp;

    printf("DEBUT: *pa=%d ; *pb=%d\n", *pa, *pb);

    temp = *pa;
    *pa = *pb;
    *pb = temp;

    printf("FIN  : *pa=%d ; *pb=%d\n", *pa, *pb);
} // Fin echange
```

Exemple Complet

Que fait ce programme ?
Tout d'abord, dans la fonction _main, nous déclarons deux variables et leur affectons des valeurs.
Ensuite, nous passons les adresses de ces deux variables à la fonction echange.
Cette fonction travaille ensuite sur les pointeurs, et échange les valeurs pointées.
Et nous finissons par revenir à la fonction _main, où nous pourrons constater que les valeurs de a et de b ont bien été échangées.

Et voici le résultat qui sera affiché à l'écran :

```
AVANT:   a=10 ;   b=20
DEBUT: *pa=10 ; *pb=20
FIN  : *pa=20 ; *pb=10
APRES:   a=20 ;   b=10
```

Il est possible que, une fois arrivé à la fin de ce chapitre, les pointeurs vous paraissent encore un peu énigmatiques... Je dirais presque que c'est normal : d'après ce que j'ai pu observer, et d'après mon expérience personnelle, on ne maîtrise la notion de pointeur, et on ne sait véritablement les utiliser, qu'au bout d'un certain temps, une fois qu'on s'y est habitué... En somme, le savoir-faire vient avec la pratique, une fois encore.
Au cours des chapitres suivant, nous serons amenés à utiliser les pointeurs plus d'une fois. En particulier, nous les utiliserons pour les tableaux, et, donc, les chaînes de caractères, et pour tout ce qui se rapporte à l'allocation dynamique de mémoire.