

Pointeurs

Temps d'étude conseillé : 3h.

Voici enfin le cours le plus important de toute votre année ! Les pointeurs sont au cœur du langage C, et vous en entendrez parler pendant de nombreuses années. Pour autant, il s'agit d'une notion pas si complexe, et vous avez déjà quelques bases pour les comprendre.

1 Introduction

La définition suivante est à retenir toute votre vie.

Définition 1 (Pointeur). Un *pointeur* est une variable qui contient une adresse.

En l'état, un pointeur ne semble pas particulièrement utile. Au lieu de stocker une valeur, un pointeur stocke une adresse, comme par exemple l'adresse d'une autre variable. En effet, comme vous l'avez vu dans le chapitre 2, toute variable déclarée a un emplacement mémoire réservé. Cet emplacement mémoire est repéré à l'aide d'adresse. Un pointeur stocke donc ce genre d'adresse.¹

En pratique, il est obligatoire de connaître le type de données que l'on peut trouver à une adresse donnée. En effet, si on vous donne une adresse, autrement dit un début de zone de lecture, sans aucune autre information, comment savez-vous le nombre d'octet à lire pour interpréter correctement la donnée ? C'est pour cette raison qu'un pointeur est associé à un type de donnée particulier, et on ne peut pas le modifier, tout comme le type affecté à une variable.

Une fois qu'un pointeur est déclaré, il est possible de réaliser deux actions :

- affecter l'adresse d'une variable au pointeur ;
- aller voir le contenu de l'adresse pointée par le pointeur : on parle de *déréférencement*.

Pour effectuer ces deux actions, les symboles `&` (adresse de...) et `*` (déréférencement de...) sont disponibles comme l'indique l'extrait de code suivant.

```
// Déclaration
type *pointeur;

// Affectation d'une adresse
pointeur = &variable;

// Dééréférencement
variable = *pointeur;
```

1. Notez qu'un pointeur est une variable qui stocke une adresse.

Exemple 1. Considérons l'extrait de code ci-après et commentons son effet sur la mémoire.

Adresse	Nom de var.	Valeur de var.
0x0160	a	10 8
0x0161		
0x0162		
0x0163		
0x0164	b	5 10
0x0165		
0x0166		
0x0167		
0x0168	p_a	NULL 0x0164
0x0169		
0x016A		
0x016B		
0x016C	p_b	0x0164
0x016D		
0x016E		
0x016F		
0x0170		
0x0171		
0x0172		

```


1  int a = 10
2  int b = 5;
3  int *p_a = NULL;
4  int *p_b = &b;
5  p_a = p_b;
6  *p_a = a;
7  a = *p_b - 2;

```

Les lignes 1 et 2 réservent chacune 4 octets sur la mémoire pour les variables `a` et `b` et initialisent ces deux plages mémoire à la valeur 10 et 5. La ligne 3 permet de déclarer un pointeur nommé `p_a` qui contient l'adresse NULL. Il s'agit de l'adresse 0x00 qui est une adresse particulière, nous verrons son utilité plus tard. La ligne 4 permet aussi de déclarer un pointeur : on vient y stocker l'adresse de la variable `b`. Notez que le type du pointeur est bien `int*` et que `b` est de type `int` : tout est bon !

Les choses sérieuses commencent à la ligne 5 : on affecte au pointeur `p_a` la valeur du pointeur `p_b`. Car oui, un pointeur étant une variable, on peut parler de *valeur de pointeur* (même si cette valeur est une adresse). L'adresse pointée par `p_b` étant l'adresse 0x0164, la valeur du pointeur `p_a` change et passe de NULL à 0x0164.

La ligne 6 montre un premier exemple de déréférencement. L'expression `*p_a`, placée à **gauche du symbole '='**, indique que l'on va **modifier** ce qu'il y a à l'adresse pointée par `p_a`. Autrement dit, le contenu de la variable `a` (ici 10) est copié à l'adresse pointée par `p_a` (ici 0x0164).

Sur la dernière ligne, on retrouve à nouveau un déréférencement, cette fois ci à **droite sur symbole '='**. Cela signifie qu'il faut aller **récupérer** le contenu de l'adresse pointée par `p_b`. Le pointeur `p_b` ayant pour valeur l'adresse 0x0164, son contenu est 10. La ligne 7 indique qu'il faut ensuite lui soustraire la valeur 2 et copier ce résultat dans la variable `a`. La variable `a` passe donc de la valeur 10 à la valeur 8. 

TD Exercices 1 et 2.

2 Intérêt des pointeurs

Sans le savoir, vous avez déjà manipulé les pointeurs. Pour récupérer une valeur saisie par l'utilisateur, vous avez utilisé la fonction `scanf()`. Pour faire fonctionner correctement cette fonction, votre enseignant vous a précisé qu'il fallait utiliser le symbole `&` devant le nom de la variable à affecter. Maintenant, vous savez pourquoi. Mais expliquons-le quand même.

Dans le chapitre sur les fonctions, vous avez appris que lorsqu'on passait une variable en paramètre d'une fonction, seule sa valeur était envoyée, pas la variable elle-même. Un tel appel de fonction utilisait le principe de *passage par valeur*. Si l'on reprend l'exemple

de `scanf()`, utiliser un passage par valeur n'aurait pas sens : on veut que `scanf()` affecte une valeur à notre paramètre, on ne veut pas donner une valeur particulière à `scanf()`.

L'intérêt de mettre le symbole `&` devant le nom de la variable est de donner à `scanf()` l'adresse de la variable à modifier. Ainsi, un déréférencement est effectué dans `scanf()` afin de modifier la valeur de la variable présente dans le `main()`. L'intérêt des pointeurs est donc ici de pouvoir modifier une variable déclarée en dehors de la fonction ; chose qui aurait été impossible sans dû à cause du principe de portée des variables.

Définition 2. Lors d'un *passage par valeur*, seule la valeur du paramètre est passé à la fonction. Lors d'un *passage par l'adresse*, l'adresse de la variable passée en paramètres est envoyée à la fonction.

Exemple 2.

```
1 void permute ( int *i, int *j )
2 {
3     int tmp = *i;
4     *i = *j;
5     *j = tmp;
6 }
7
8 int main()
9 {
10    int a = 0, b = 10;
11    permute (&a, &b);
12    printf ("a=%d, b=%d\n", a, b);
13    return 0;
14 }
```

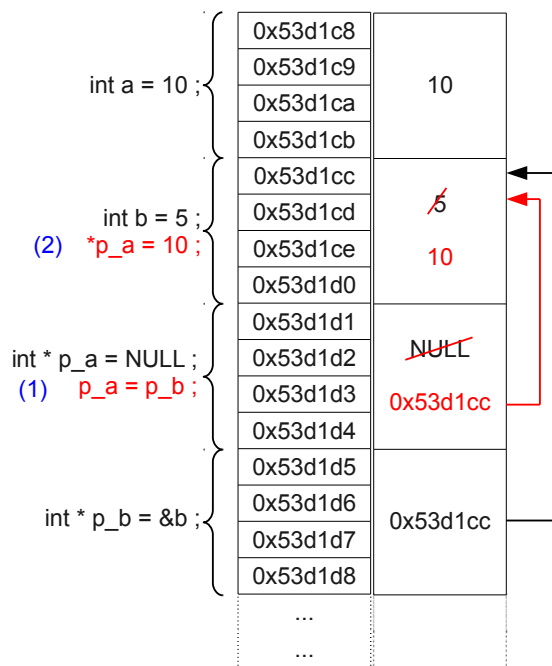
L'exemple typique du passage par adresse est celui de l'échange de deux variables. On déclare dans l'exemple ci-dessus deux variables `a` et `b` dans le `main()`. En donnant l'adresse des variables `a` et `b` à la fonction `permute()`, il est possible de modifier leur valeur **depuis** la fonction `permute` en utilisant un déréférencement. ▲

TD Exercices 3 et 4.

3 Résumé

Concluons ce chapitre avec un exemple résumant les différentes notions liées aux pointeurs. Un pointeur est donc une variable manipulant une adresse mémoire et **uniquement une adresse mémoire**, et est associé à un type de donnée. Il se définit de la façon suivante :

```
type * nom_pointeur;
```



```

int a = 10, b = 5;
int *p_a = NULL, *p_b = &b;
p_a = p_b;           // (1)
*p_a = a;            // (2)

```

Le schéma ci-contre nous montre l'action des pointeurs sur la mémoire. Deux entiers et deux pointeurs sur entier sont utilisés. Le premier pointeur `p_a` est initialisé à `NULL` tandis que le pointeur `p_b` désigne l'adresse de la variable `b` (en noir dans le tableau ci-contre).

L'instruction (1) modifie l'adresse sur laquelle pointe `p_a`. Ce pointeur `p_a` pointe maintenant sur la même zone mémoire que `p_b`.

L'action (2) change quant à elle le contenu de la zone mémoire pointée par `p_a` en lui affectant la valeur stockée dans la variable `a`.

TD Exercices 5 à 7.