Pointeurs en C

Hicham Janati

1 Introduction

Un pointeur est une variable contenant l'adresse d'une autre variable. Ils nous permettent de manipuler les variables en passant par leur adresse (on verra plus tard pourquoi c'est plus puissant).

RAPPEL : Supposons deux variables a et b déclarées. On peut accéder à l'adresse de a avec le caractère "&". Voici un schéma montrant un pointeur p "pointant" sur la variable a :

Adresse	...	&a	...	&p	...	...
Contenu	...	a	...	&a	...	...

Pour déclarer un pointeur, il faut préciser le type sur lequel il va pointer suivi d'un astérisque. Pour afficher un pointeur, on utilise le format "%p" :

1. **Reprendre ce code sur votre machine :**

```c
#include<stdio.h>
int main(){
   float a = 5;
   float* p;
   p = &a;

   // On affiche la valeur de a et son adresse:
   printf("La variable a = %f\n",a);
   printf("L'adresse de a = %p\n",&a);

       // La valeur de p et son adresse
   printf("La valeur de p = %p\n", p);
   printf("L'adresse de p = %p\n", &p);

   return 0;
}
```

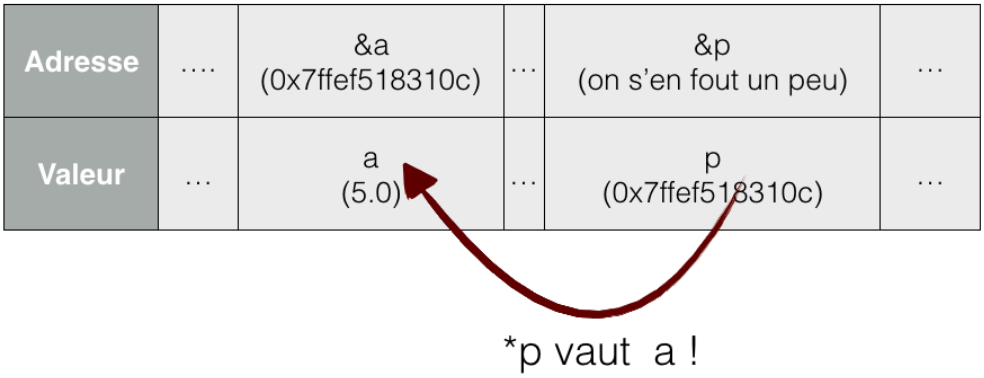
```
sh-4.2$ main
La variable a = 5.000000
L'adresse de a = 0x7ffef518310c
La valeur de p = 0x7ffef518310c
L'adresse de p = 0x7ffef5183100
```

On remarque bien que la valeur de p est bien égale à l'adresse de a ! Cool mais ça sert à quoi au juste ?

Eh ben en fait on peut *accéder à la VALEUR de a* en passant par p : il suffit d'ajouter un * avant le pointeur :

```c
   printf("La valeur de  *p = %f\n", *p);
```

```
La valeur de  *p = 5.000000
```



À retenir

Si p est un pointeur sur a :

- p vaut l'adresse de a : &a.
- *p vaut la valeur de a : a : Changer *p est équivalent à changer a !
- pour afficher une adresse (un pointeur) on utilise le format %p.
- pour déclarer un pointeur, il faut préciser le type sur lequel il va pointer suivi d'un *.

2. **Créez un pointeur sur un entier et changez sa valeur en passant par le pointeur.**

3. **Reprenez cette fonction et devinez pourquoi la ligne 8 n'a aucun effet !**

```
1  #include<stdio.h>
2  void Changer(int a){
3          a = 0;
4      }
5  int main(){
6      int a = 10;
7      printf("La variable a = %d\n",a);
8      Changer(a);
9      printf("La variable a = %d\n",a);
10     return 0;
11 }
```

```
1      La variable a = 10
2      La variable a = 10
```

En passant la variable a à la fonction *Changer*, la fonction n'opère pas *directement* sur la variable a en mémoire mais crée une copie de a ! à cette copie, elle affecte la valeur 0. Mais comme la copie est une variable *locale* à la fonction, elle est détruite à la sortie de la fonction. La variable a reste inchangée.

4. **Pour y remédier, au lieu de passer à la fonction la variable, on lui donne un pointeur sur a : elle aura ainsi un accès direct à sa place en mémoire et pourra changer sa valeur. Remplacez la fonction Changer par la suivante, quel changement doit être effectué à main pour utiliser cette nouvelle fonction ?**

```
1  #include<stdio.h>
2  void Changer(int* a){
3          *a = 0;
4      }
```

5. **Écrire une fonction qui prend deux arguments et permutent leurs valeurs *en mémoire*.**

Mystère de scanf élucidé

Vous êtes maintenant en mesure d'expliquer pourquoi avec printf on donne la variable en valeur alors qu'on passe l'adresse de la variable à scanf :
Printf n'a besoin que de la valeur d'une variable pour l'afficher à l'écran. **Scanf** a par contre besoin de savoir où se trouve la variable en mémoire pour pouvoir changer sa valeur et la remplacer par la valeur saisie au clavier.

Conclusion

Il existe deux façons de passer un argument à une fonction :

1. Par valeur : La fonction prend la **valeur** de la variable et crée une **variable locale (copie)**. Elle ne peut donc pas modifier la variable en mémoire. (Fonctions qu'on a vues précédemment)
2. Par adresse (pointeur) : La fonction prend **l'adresse** de la variable en argument et a donc un accès direct à sa position en mémoire et peut la changer.

Mais lorsqu'on appliquait des fonctions sur un tableau pour faire le tri par exemple, elle changeait bien ses valeurs en mémoire non ?

Tout à fait ! C'est parce qu'en réalité, le nom que vous donnez à votre tableau n'est autre qu'un pointeur sur la première case ! Autrement dit, c'est l'adresse de votre tableau. Ceci est l'objet de la partie suivante.

2 Pointeurs et tableaux

Au chapitre des tableaux, nous avons vu que lorsque l'on déclare un Tableau T de taille 5 par exemple, le système d'exploitation (Linux, Windows ..) réserve 5 cases **suivies** en mémoire (de même type) pour notre programme. Vérifions la remarque mentionnée ci-dessus. Je vous rappelle que %p est le format spécial des pointeurs.

```c
#include<stdio.h>
int main(){
   float T[] = {0,5,10,15,20};
   printf(" T = %p \n", T);
   printf(" L'adresse de la premiere case du tableau : %p \n", &T[0]);
   return 0;
}
```

```
 T = 0x7fffdb3df120
L'adresse de la premiere case du tableau : 0x7fffdb3df120
```

Elles sont bien égales ! Et maintenant ?

J'ai une nouvelle pour vous : les pointeurs sont stables par addition ! C'est à dire que si T est l'adresse de la première case, T+1 sera l'adresse de la deuxième etc ... On peut donc "nous balader" dans la mémoire en incrémentant un pointeur. Essayons pour la 4ème case :

```c
printf("methode 1: &T[3]  =   %p \n", &T[3]   );
printf("methode 2:  T+3   =  %p  \n", T+3     );
```

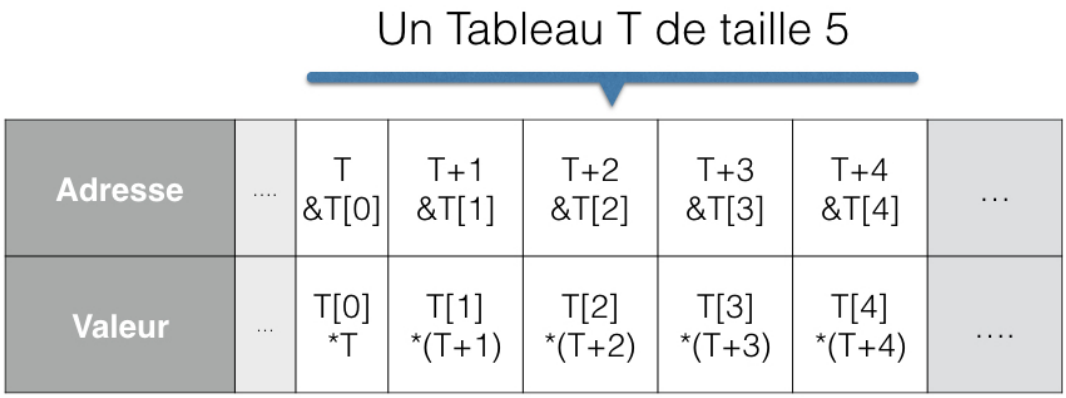
```
methode 1: &T[3]  =  0x7ffd6e9584bc
methode 2:  T+3   =  0x7ffd6e9584bc
```

1. **Complétez le code suivant pour afficher l'adresse chaque case du tableau T de deux manières différentes :**

```
1  int i;
2  for(i=0;i<5;i++){
3          printf("methode 1: &T[i]  =    \n",     );
4          printf("methode 2:        =    \n",     );
5  }
```

2. **Comme *p est la valeur stockée à l'adresse p, eh ben pour afficher la (i+1)ème valeur d'un tableau T, on peut logiquement écrire *(T+i). À ne pas confondre avec *T+i ! *T+i est tout simplement égale à la valeur T[0]+i alors que *(T+i) est T[i]. (Je dis (i+1) ème car, rappelez-vous, on commence l'indexation à 0). Écrire un programme qui affiche les valeurs du tableau de deux manières différentes.**

```
1  int i;
2  for(i=0;i<5;i++){
3          printf("methode 1:   =    \n",     );
4          printf("methode 2:   =    \n",     );
5  }
```



3. **Écrire un programme qui permet à l'utilisateur de remplir un tableau en utilisant la deuxième méthode.**

4. **Écrire un programme qui affiche le tableau en utilisant la deuxième méthode.**

Conclusion

1. Lorsqu'on déclare un tableau *NomDuTableau*, alors *NomDuTableau* est l'adresse de sa première case.
2. Il existe deux équivalences à retenir :

$$\begin{aligned} \text{L'adresse de la (i+1)eme case :} \quad & \&T[i] \quad \Leftrightarrow \quad T+i \\ \text{La valeur de la (i+1)eme case :} \quad & T[i] \quad \Leftrightarrow \quad {}^*(T+i) \end{aligned}$$

3 Les différents types et les pointeurs

Si un pointeur est une variable qui contient une adresse qui .. est un nombre (en base hexadécimale mais ça reste un nombre), pourquoi doit-on préciser son type ?

La mémoire est structurée en plusieurs cases unitaires (vous le savez déjà, normalement). Ce que l'on a omis jusque là est le fait qu'une variable **peut en occuper plusieurs !**.

On peut savoir combien de cases occupe une variable avec la fonction *sizeof*. Ce nombre ne dépend que du type de la variable :

```
1  char c;
2  int i;
3  float x;
4  double y;
5  printf("La taille d'un char: %d", sizeof(c));
6  printf("La taille d'un int: %d", sizeof(i));
7  printf("La taille d'un float: %d", sizeof(x));
8  printf("La taille d'un double: %d", sizeof(y));
```

```
1 La taille d'un char: 1
2 La taille d'un int: 4
3 La taille d'un float: 4
4 La taille d'un double: 8
```

Ainsi, un char occupe une case en mémoire, un float en occupe quatre etc ... Vous pouvez donc en déduire l'utilité de la précision du type à la déclaration d'un pointeur : **À la déclaration d'un pointeur, le système d'exploitation doit savoir combien de cases en mémoire il doit réserver pour la variable.**

Pour vous en convaincre, créez un tableau de char (une chaîne de caractères) et affichez les adresses de chaque case du tableau. Puis faites de même pour un tableau d'entiers :

```
1  int i;
2  char chaine[] = "Les pointeurs, c'est facile !"
3  for(i=0;i<5;i++){
4          printf("Adresse chaine + i = %p \n",  chaine +i   );
5  }
```

```
1 Adresse chaine + 0 = 0x7fff348381b0
2 Adresse chaine + 1 = 0x7fff348381b1
3 Adresse chaine + 2 = 0x7fff348381b2
4 Adresse chaine + 3 = 0x7fff348381b3
5 Adresse chaine + 4 = 0x7fff348381b4
```

C'est inhumain certes, mais on voit bien que les adresses augmentent de 1 en hexadécimal.

```
1  int i;
2  int T[] = {3,4,5,6,3};
3  for(i=0;i<5;i++){
4          printf("Adresse T + %d = %p \n",i,  T +i   );
5  }
```

```
1 Adresse T + 0 = 0x7fffba5fc6e0
2 Adresse T + 1 = 0x7fffba5fc6e4
3 Adresse T + 2 = 0x7fffba5fc6e8
4 Adresse T + 3 = 0x7fffba5fc6ec
5 Adresse T + 4 = 0x7fffba5fc6f0
```

Et de 4 pour des entiers ! Les adresses évoluent donc avec un pas égal à la taille du type.

Pour les sceptiques d'entre vous à qui le 'c' et le '0' à la fin des dernières lignes donnent une migraine, voici une explication :

En base hexadécimale, on compte avec 16 caractères au lieu des 10 caractères numériques entre 0 et 9 mais on ajoute 6 caractères supplémentaires : a,b,c,d,e,f ! Et ce de la manière suivante :

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. Puis :
10,11,12,13, 14,15,16,17, 18,19,1a,1b, 1c,1d,1e,1f

Ainsi, au dernier caractère, lorsque vous ajoutez 4 à 8, vous tombez sur c, puis sur 0 etc ... [vous passez de e à f car en ajoutant 4 à c, vous mettez 0 aux unités et retenez 1, e+1 .. c'est f]

ATTENTION : On se soucie assez rarement de l'adresse en base hexadécimale. Car lorsqu'on incrémente un pointeur de 1, le compilateur avancera d'une, de quatre ou de huit cases selon le type de la variable. Il suffit de retenir qu'en incrémentant un pointeur, on passe à la variable suivante.

Remarque

Maintenant vous savez que vous pouvez manipuler un tableau en utilisant le pointeur sur sa première case et sa taille. Il est donc équivalent de définir des fonctions opérant sur les tableaux de la manière suivante :

```
1  void SommeTableau(int* T, int N){
2      int i,s = 0;
3      for(i=0;i<N;i++){
4          s = s + *(T+i);
5          }
6      return s;
7  }
8
9  void SommeTableau(int T[], int N){
10     int i,s = 0;
11     for(i=0;i<N;i++){
12         s = s + T[i];
13         }
14     return s;
15 }
```

4 Exercices

En utilisant les pointeurs :

1. Écrire une fonction qui prend deux variables passées en adresses et **renvoie** le nombre le plus grand.
2. Écrire une fonction qui inverse un tableau.
3. Écrire une fonction qui inverse une chaîne de caractères.
4. Écrire une fonction qui prend deux tableaux de même taille et permutent leurs valeurs.
5. Vous avez appris qu'une fonction à qui on passe une variable en valeur en fait une copie locale et ne peut donc pas accéder à son adresse en mémoire. Créez un programme qui permet de le vérifier.