Pointeurs et fonctions en C : un résumé

Alexandra Bac

Biomédical 1ère année

1 Introduction

Les liens entre pointeurs et fonctions font partie des notions les plus complexes du C. Afin, suite aux TDs, de clarifier ces notions, ce document reprend ces notions.

En C, comme cela a été dit de nombreuses fois :

En C, les arguments d'une fonction sont passés par valeur.

Par exemple :

```
(1)  int mafonction{int x}
(2)  {
(3)    ....
(4)  }
(5)
(6)  int main()
(7)  {
(8)     int a = 2 ;
(9)     mafonction(a) ;
(10) }
```

A la ligne 9, ce n'est donc pas la variable `a` qui est passée à la fonction, mais sa valeur (soit 2) qui est stockée dans la variable locale `x` pour la durée de la fonction. Il n'y a pas de lien entre `a` et `x`.

Le but du lien entre fonctions et pointeurs est de pouvoir **retourner des résultats** de fonctions et de pouvoir **modifier des valeurs de variables**.

2 Modifier des valeurs de variables dans une fonction

Pour pouvoir, dans une fonction, modifier le contenu d'une variable passée en argument, pas de choix en C :

Pour modifier une variable dans une fonction, il faut **passer un pointeur sur cette variable comme argument.**

Exemple 1. On veut écrire une fonction ajoutant à une variable entière `x` une variable `y`. On veut donc modifier la variable `x` : il faut passer un pointeur sur cette dernière :

```
(1)  void ajouter(int *p, int b)
(2)  {
(3)     *p = *p + b ;
```

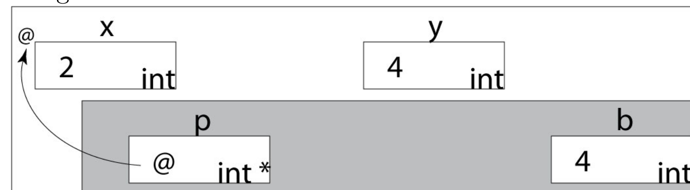
```
(4)     return ;
(5)  }
(6)  int main ()
(7)  {
(8)     int x = 2, y=4 ;
(9)     ajouter(&x, y) ;
(10)    return 0 ;
(11) }
```

Dans le `main`, on a les variables suivantes :

Dans la fonction `ajouter` :

- deux variables locales sont crées : `p` (pouvant contenir l'adresse d'un entier) et `b` (pouvant contenir un entier).
- dans `p`, on stocke l'adresse de `x`
- dans `b`, on stocke la valeur de `y`

On obtient donc la configuration suivant de variables :



La partie grisée de la mémoire est celle des variables locales : ce sont les variables propres à la fonction. Cette partie est vidée (et les variables détruites) à la fin de la fonction ... c'est tout le problème !

3 Retourner un résultat

Pour retourner un résultat, deux cas de figures :

- **Cas des types simples (`int, float, char`)**
  On veut renvoyer un entier, un flottant ou un caractère. Ce sont des types simples. Il suffit de :

  renvoyer une **valeur** grâce à `return`

  Exemple :

```
(1)  int somme (int x, int y)
(2)  {
(3)    return x+y ;
(4)  }
(5)
(6)  float myf (float x)
(7)  {
(8)    float y ;
(9)    y = y*x + 2*(y-1) ;
(10)   return y ;
(11) }
```

  Dans la première fonction, la fonction somme renvoie un entier (la somme `x+y`), dans la seconde, un flottant.
  Dans les deux cas, on renvoie des valeurs de type élémentaire.

- **Cas des types complexes (tableaux, structures)**
  Il est (comme vous l'avez peut-être testé en TP) possible de faire un `return` sur une structure, mais dans ce cas, la structure est copiée. C'est la copie qui est retournée alors que la variable est détruite ...
  L'intérêt des structures est de pouvoir créer de grosses structures (listes, arbres ...) Dans ce cas, ces copies deviennent très coûteuses.
  Tout cela vous paraît compliqué ? C'est pourquoi on peut adopter la politique suivante (pas obligatoire, mais la plus efficace).

  On ne fait jamais de `return` sur un tableau ou structure :

– Stratégie 1 : on modifie une variable passée en argument
– Stratégie 2 : on renvoie un pointeur sur le résultat (+ `malloc`)

A titre d'exemple, nous allons suivre 2 fonctions :
- `fct_tableau` prenant en argument un entier `n` et retournant un tableau contenant les entiers de 1 à `n`,
- `fct_structure` prenant en argument un entier et un flottant et renvoyant une structure contenant ces deux nombres.

3.1 Retour de résultat, stratégie 1 : via les arguments

Fonction `fct_tableau`. Dans cette stratégie, le tableau est passé comme argument supplémentaire à la fonction :

```
void fct_tableau (int n, int t[])
{
   int i ;
   for (i=0; i<n; i=i+1)
      t[i] = i ;
   return ;
}
int main ()
{
  int n ;
  int *tab ;
  printf("Entrez n : ") ;
  scanf("%d", &n) ;
  tab = malloc(n * sizeof(int)) ;
  fct_tableau(n, tab) ;
  return 0 ;
}
```

C'est donc dans le main que la déclaration ou allocation du tableau est faite. Notez bien que comme les tableaux C sont des pointeurs, on passe bien un *pointeur* à la fonction `fct_tableau` et celle-ci peut donc bien modifier les cases du tableau.

Fonction `fct_structure`. Dans cette stratégie, la structure est passée comme argument supplémentaire à la fonction. Mais pour qu'elle puisse être modifiée, c'est en fait un pointeur sur la structure qu'il faut passer :

```
struct couple {
     int c1 ;
     float c2 ;
} ;
```

```
void fct_structure (int n, float x, struct couple *p)
{
   (*p).c1 = n ;
   (*p).c2 = x ;
   return ;
}
int main ()
{
   int i = 2 ;
   float pi = 3.14 ;
   struct couple c ;

   fct_structure(i, pi, &c) ;
   return 0 ;
}
```

Idem, la structure est donc déclarée (et allouée) dans le main, la fonction se contente de modifier les valeurs stockées dans les champs.

3.2 Retour de résultat, stratégie 2 : via `return`

Fonction `fct_tableau`. Dans cette stratégie, le tableau est retourné comme résultat de la fonction. Il doit donc être alloué dans la fonction au moyen de `malloc` :

```
int [] fct_tableau (int n)
{
   int i ;
   int *t ;

   t = malloc(n*sizeof(int)) ;
   for (i=0; i<n; i=i+1)
      t[i] = i ;
   return t ;
}
int main ()
{
  int n ;
  int *tab ;
  printf("Entrez n : ") ;
  scanf("%d", &n) ;

  tab = fct_tableau(n) ;
  return 0 ;
}
```

Cette fois, c'est dans la fonction que l'allocation du tableau se fait. On renvoie juste l'adresse où ce tableau a été mis comme résultat de la fonction.

Fonction `fct_structure`. Dans cette stratégie, un pointeur sur la structure est retourné comme résultat de la fonction (elle doit donc être allouée dans la fonction avec un `malloc`) :

```
struct couple {
```

```
    int c1 ;
    float c2 ;
} ;

struct couple * fct_structure (int n, float x)
{
   struct couple *p ;
   p = malloc(sizeof(struct couple)) ;
   (*p).c1 = n ;
   (*p).c2 = x ;
   return p ;
}
int main ()
{
   int i = 2 ;
   float pi = 3.14 ;
   struct couple * c ;

   *c = fct_structure(i, pi) ;
   return 0 ;
}
```

Dans ce cas, l'allocation se fait dans la fonction (plus dans la `main`) et la fonction renvoie un pointeur.

3.3 Conclusion

Choisir une stratégie ou l'autre n'est qu'une question de préférence, de contexte. Dans tous les cas, structures et tableaux doivent être alloués (on doit leur réserver de la place en mémoire). Dans la première stratégie, c'est fait dans le `main`, dans la seconde, c'est fait dans la fonction ...

Vous n'avez qu'à choisir !