TP 5 - Un point sur les pointeurs (de C)

8 octobre 2014

Au menu de la séance : des *pointeurs*.

quelques figures par Jean Goubault

Definition 1. *Un pointeur est en programmation une variable contenant une adresse mémoire.*

On peut donc stocker des adresses mémoires dans des pointeurs : on *pointe* de cette façon la case mémoire qui se situe à cette adresse. De cette façon, on accède à une couche plus basse de l'ordinateur en se donnant un accès direct à la mémoire.

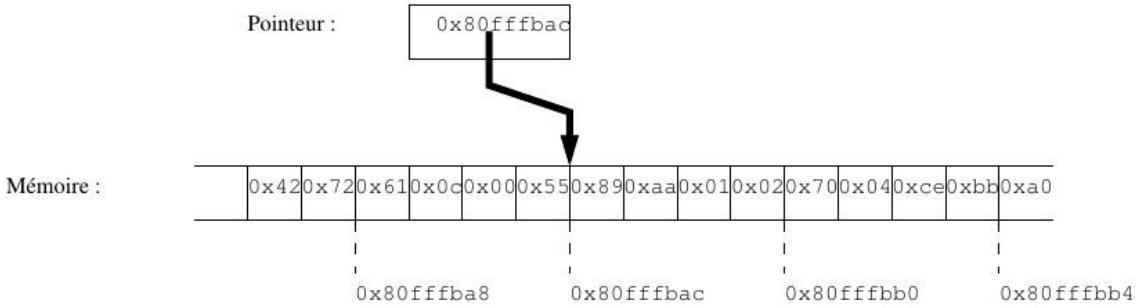

FIGURE 1 – Un pointeur (lui même quelque part dans la mémoire)

On va maintenant comprendre et apprendre à utiliser ces pointeurs en C. Vous aurez pour chaque notion : une explication, un bout de code C et une représentation mémoire de l'état final.

Déclaration d'un pointeur

En écrivant en C : `int *monPointeur;` on déclare un pointeur vers un entier. De façon générale, `type *nom` déclare un pointeur `nom` qui doit pointer vers une case contenant une valeur de type `type`. Il faut comprendre que pour le moment, `monPpointeur` n'a pas encore été instancié par une adresse : il peut contenir n'importe quoi et si on essayer d'aller voir la case pointée, on s'expose à une corruption mémoire (la fameuse `segmentation fault`). En effet, aucune case supplémentaire dans la mémoire n'a été allouée.

```
int *monPointeur;
```

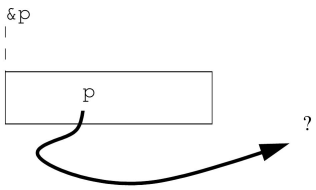

FIGURE 2 – Un pointeur indéfini (p raccourci pour monPointeur)

Assignation d'un pointeur, accès aux cases pointées

Il y a deux façons d'assigner un pointeur ; c'est-à-dire changer l'adresse qu'il contient pour une adresse définie (cad. adresse d'une case mémoire existante) :

1. On peut le faire en récupérant une adresse d'une valeur déjà définie ;
2. On peut *allouer dynamiquement* de la mémoire et récupérer puis assigner l'adresse de la case (ou des cases) qui nous a été allouée.

Je reviendrai sur la seconde solution plus tard. On se concentre pour le moment sur la première façon. Par exemple si `x` est une variable de tyê `int` qui a déjà été définie alors on peut accéder à son adresse en utilisant l'opérateur `&` : comme ceci `&x`. On peut maintenant écrire `monPointeur = &x;` ce qui instancie `monPointeur` par *l'adresse* de x. En d'autres termes, `monPointeur` *pointe* maintenant vers la case mémoire de x.
⇝ L'opérateur `&` devant une variable renvoie donc son adresse (au lieu de renvoyer sa valeur).

L'opérateur "dual" est `*`. Il permet de récupérer la valeur contenue dans la case pointée par le pointeur. Par exemple, `*monPointeur` renvoie le contenu de la case dont l'adresse est `monPointeur` (ici c'est la valeur de `x`). On a aussi accès en écriture. Par exemple si on écrit ensuite `*pointeur = 7` alors on modifie le contenu de la case mémoire vers laquelle pointe `pointeur`. Hors cette case, c'est x. Donc x vaut maintenant 7 !
⇝ En d'autres termes, l'opérateur $*$ devant un pointeur permet d'accéder (en lecture et écriture) à la valeur stockée dans l'adresse contenue par le pointeur.

```
int x = 4;
int *monPointeur;
pointeur = &x;
```

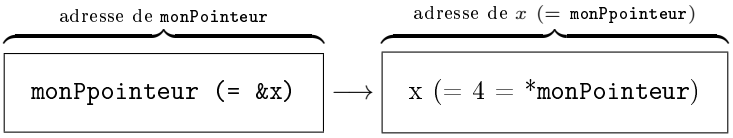

FIGURE 3 – Assignation de `monPointeur` par l'adresse de `x` (dans les boîtes, on représente la valeur stockée par la case mémoire ; au-dessus, l'adresse de la case mémoire ; la flèche nous aide juste à visualiser le lien).

Pointeurs et tableaux

La semaine dernière, on a déja manipulé des pointeurs sans le savoir (ou pas). Par exemple, quand on écrivait `int tab[5]`, on déclarait un tableau de taille 5. Plus précisemment, cette instruction alloue 5 cases mémoire côte à côte et déclare `tab` comme étant un pointeur vers un entier. `tab` est également instancié par l'adresse de la première case mémoire du tableau.

L'accès aux cases du tableau se fait donc très simplement en additionnant le pointeur `tab` avec le décalage voulu. L'adresse de la seconde case du tableau est donc `tab+2` qui revient au même que `&tab[2]`. Attention, le compilateur sait que `tab` est un tableau *d'entiers* donc il comprend `tab+2` comme `tab+(2*sizeof(int))` (cad. qu'il sait qu'il faut se décaler de 8 octets (=`sizeof(int)`) pour passer à la case suivante).

```
int tab[5];
tab[0] = 2;
*(tab+1) = 5;
*(&tab[2]) = 3;
    ....
```

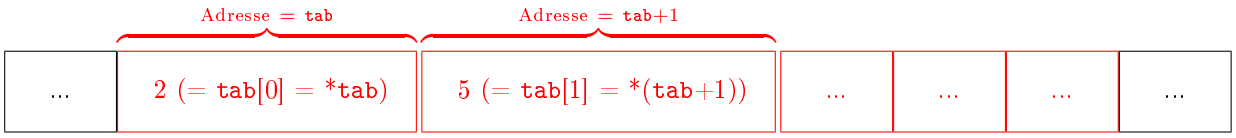

FIGURE 4 – Représentation mémoire de `tab`

Allocation dynamique

Pour allouer dynamiquement des cases mémoires, il faut expliciter la taille que l'on veut allouer (cad. le nombre de cases mémoires). Par exemple pour un tableau d'entiers de taille n la taille du tableau est `sizeof(int) * n` (taille d'un entier fois n). La fonction `malloc` prend en argument la taille à allouer et renvoie un pointeur vers la première case. Exemple : pour déclarer et allouer de l'espace pour un tableau d'entiers `tab` de taille n, vous pouvez utiliser : `int *tableau = malloc(sizeof(int) * n)`. Attention, lorsque vous ne voulez plus utiliser cette espace vous devez manuellement demander sa suppression avec `free`. Exemple : `free(tab)` désalloue la mémoire. Si vous oubliez de le faire, vous risquez une *fuite mémoire* qui peut faire planter votre programme.

Et pour finir : des schémas de structures de donnés classiques

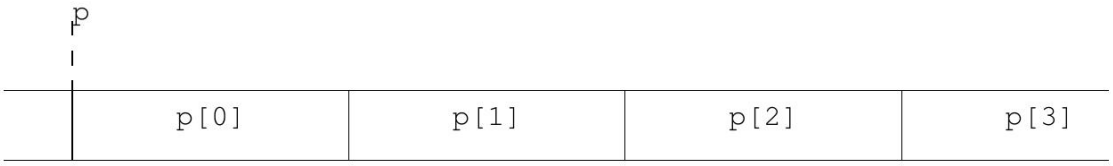

FIGURE 5 – Une tableau (en pointillé : l'adresse)

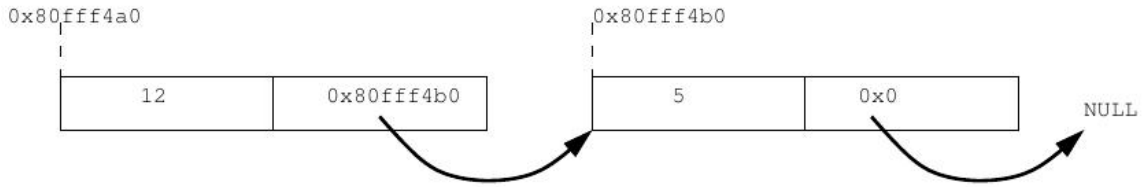

FIGURE 6 – Une liste chaînée

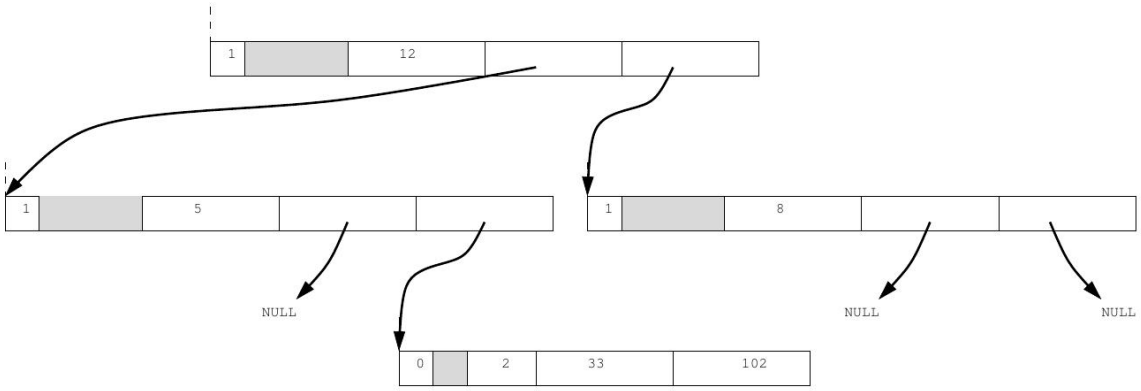

FIGURE 7 – Un arbre binaire