TP6 – Pointeurs et allocation dynamique

Projet de programmation M1

02 et 09 Novembre 2015

Séance casse-tête. Sortez vos crayons. C est un langage de programmation qui laisse une grande responsabilité au programmeur : la gestion de la mémoire. On va essayer de comprendre la gestion de la mémoire en C. Pour cela, on va s'entraîner un peu à manipuler *les pointeurs* et voir le concept de *l'allocation dynamique*. Nous mettrons cela en pratique en implémentant les tableaux à taille variable.

1 La mémoire est un tableau

Lorsque vous exécutez un programme, le système d'exploitation (Linux, Windows, OSX ...) réserve une partie de la mémoire vive (la RAM, une mémoire plus petite que le disque dur mais plus rapide) dans lequel le programme va pouvoir opérer. Une partie de cette mémoire est utilisée pour stocker les instructions que suit votre programme et pour certains détails administratifs que nous ignorerons pour ce cours. Le reste de la mémoire est utilisé pour stocker les variables que vous définissez. C'est cette partie là qui nous intéresse. On va représenter cela comme un gros tableau. Les indices de ce tableau sont appelés *les adresses* (*address* en Afrikaner). Chaque case de ce tableau a une taille de 8 bits, ce qu'on appelle un *octet* (*byte* en Écossais). La plupart de ces cases sont libres, d'autres contiennent les données de votre programme. Lorsque vous définissez une nouvelle variable x, votre programme regarde d'abord la taille de cette variable. Si c'est un `int` par exemple, la variable sera de taille 32 bits, soit 4 octets. Pour un `char`, on a besoin de 1 octet seulement. Supposons pour la suite que x est un `int`. Le programme cherche alors 4 cases libres consécutives dans la mémoire et les réserve, c'est-à-dire que toutes les autres variables de votre programme n'empièteront pas dessus. La première case réservée pour x s'appelle *l'adresse* de x. Les adresses sont en général un nombre stocké sur 64 bits (ce qui permet de stocker plus de 18446744073 milliards d'adresses, ce qui suffit vu la taille des mémoires vives actuelles). On peut manipuler ces adresses avec les pointeurs, c'est le sujet de la partie suivante.

2 Les pointeurs

On commence par expliquer les pointeurs vers `int`. Le type "pointeur vers `int`" est un type à part entière noté `(int *)`. Une variable p de type `(int *)` sera appelé un pointeur vers `int`. Les valeurs qu'elle prendra seront des adresses de case mémoire contenant des `int`.

On peut construire un type pointeur pour n'importe quel autre type en C. On peut donc avoir des pointeurs vers `char` avec `(char *)`, des pointeurs vers des pointeurs vers `int` avec `(int **)` etc.

Question : De combien de bit a-t-on besoin pour stocker une variable de type `(int *)` ? `(char *)` ? `(char **)` ? `(int **********)` ?

Soit x une variable quelconque. Elle se trouve quelque part dans la mémoire, elle a donc une adresse. On peut récupérer cette adresse avec &. `&x` vaut donc l'adresse de la variable `x`. Soit `p` une variable qui est un pointeur. Elle contient une adresse. La valeur stockée à cette adresse peut être récupérée avec *. Par exemple, ce programme affichera 5 :

```c
int x = 5;
int *p = &x;
printf("%d", *p);
```

`&p` a un sens : c'est l'adresse de la variable `p`. Ça ne sert pas à grand chose ici mais ça peut être utile si on a des pointeurs de pointeurs (voir question 3 de l'exercice).

Attention à ne pas confondre l'étoile qu'on utilise pour définir un pointeur vers un certain type et l'étoile qu'on utilise pour récupérer la valeur stockée à une adresse. Attention aussi, si `p` est de type `char *`, alors `*p` est un `char` donc C ira lire seulement une case de la mémoire, celle de l'adresse contenue dans `p`. Si `p` est de type `int *` en revanche, `*p` est un `int`, donc C ira lire 4 cases de la mémoire, même si c'est la même adresse qu'au début ! Voir la question 5 de l'exercice pour un exemple.

Les tableaux sont des pointeurs On peut faire de l'arithmétique avec les pointeurs. Si `p` est un pointeur vers `int` contenant une certaine adresse, alors la valeur de `p+1` est l'adresse du `int` suivant. Attention, vu que `p` est de type `(int *)`, la valeur de `p+1` est celle 4 cases mémoires plus loin que `p`. Si `p` est de type `(char *)`, la valeur de `p+1` est celle de la case suivante. Lorsque vous écrivez `int t[5]={1,2,3,4,5}` que se passe-t-il en fait ? C réserve $5*4 = 20$ cases mémoires consécutives, soit de quoi stocker 5 variables de type `int` et il met dans la variable `t` l'adresse de la première case mémoire réservé. Donc écrire `t[0]` est complètement équivalent à `*t`. De même, `*(t+3)` est équivalent à `t[3]`. De façon plus générale, `t[i]` est strictement équivalent à `*(t+i)`.

Exercice 1. Qu'écrivent ces programmes à l'écran et pourquoi ?

1.
```c
int x;
*(&x) = 7;
printf("%d", x);
```

2.
```c
int x, *y, **z;
z = &y;
y = &x;
**z = 12;
printf("%d", x);
```

3.
```c
int t[] = {1,2,3};
*(t+2) = 5;
printf("%d %d %d",t[0],t[1],t[2]);
```

4.
```c
int t[] = {1,2,3,4,5};
int *x = t+2;
int i = 0;
for(i=0;i<2;i++)
  printf("%d\n", x[i]);
```

5.
```c
int x=280;
int *y=&x;
char *c=&x; // ca affiche un warning mais ca marche
printf("%d\n", *y);
printf("%d\n", *c);
```

Pointeurs et fonctions Il est courant d'utiliser les pointeurs pour créer des fonctions qui modifient la valeur de leurs arguments. Au lieu de passer une variable en argument, on passe son adresse. La fonction peut alors modifier la valeur contenu à cette adresse qui est en fait la valeur de la variable.

Exercice 2. Que fait le programme suivant :

```c
void f(int x, int y, int *res) {
  *res = x+y;
  x=0;
  y=0;
}
int main() {
  int x=5,y=5;
  f(x,y,&x);
  printf("x=%d et y=%d",x,y);
}
```

Expliquez alors pourquoi lorsqu'on modifie une case d'un tableau dans une fonction, cela modifie vraiment le tableau.

3 Allocation dynamique

L'allocation dynamique est un moyen d'obtenir dynamiquement plus de mémoire. On dispose pour cela d'une fonction magique de la librairie `stdlib.h` : `malloc`. Cette fonction prend en argument un nombre k de cases mémoires et renvoie une adresse. Cette adresse renvoyée est une adresse à partir de laquelle `malloc` a reservé pour vous k cases mémoires. Si cette adresse vaut `NULL`, c'est que `malloc` n'a réussi à réserver la mémoire (ça peut arriver s'il n'y a plus de mémoire disponible par exemple).

Supposons par exemple qu'on veuille réserver de la mémoire pour stocker un tableau de taille 25. On pourra écrire :

```
int *t = (int *) malloc(25*sizeof(int));
```

La fonction `sizeof` renvoie le nombre d'octet qu'occupe un certain type. L'adresse renvoyée par `malloc` n'est *a priori* pas typée, c'est pourquoi on la convertit en `(int *)` afin de clarifier la situation, mais ce n'est pas complètement obligatoire. On réserve donc `25*sizeof(int) = 100` octets dans la mémoire qu'on pourra manipuler puisqu'on sait qu'ils se trouvent à l'adresse `t`.

Pourquoi ne pas avoir écrit directement `t[25]` ? C'est parce qu'on ne connaît pas toujours à l'avance la taille dont on aura besoin pour `t`. Par exemple, on pourrait demander à l'utilisateur une suite de valeurs jusqu'à ce qu'il entre 0. À ce moment, on écrit la suite entrée dans l'ordre croissant. On a aucun moyen de savoir à l'avance combien de nombre l'utilisateur entrera. On a besoin pour cela de redimensionner la mémoire au fur et à mesure. C'est ce que nous allons illustrer avec les tableaux à taille dynamique.

Lorsqu'on a fini d'utiliser la mémoire qu'on a réservée avec `malloc`, il convient de la libérer pour que le reste du programme puisse en profiter. On utilise pour cela la fonction `free` qui prend en argument un pointeur vers la zone mémoire à libérer. Attention, si vous essayez de libérer une zone qui n'a pas été réservée avec `malloc`, votre programme plantera.

Tableau à taille dynamique On va créer notre propre structure de données qui sont des tableaux à taille variable. On veut disposer de plusieurs fonctions sur ces tableaux :

- une fonction `length` pour connaître le nombre d'élément dans le tableau
- une fonction `get` pour récupérer le $i$ème élément du tableau
- une fonction `set` pour changer la valeur du $i$ème élément
- une fonction `add` pour ajouter un élément à la fin du tableau

Pour cela, on va commencer par définir une structure :

```
struct tdyn {
  int length;
  int *t;
};
typedef struct tdyn tdyn;
```

L'idée c'est que pour une variable `x` de type `tdyn`, `x.t` est un tableau avec `x.length` cases réservées. Si la longueur est 0, on choisira `x.t = NULL`.

Exercice 3. Écrire les fonctions suivantes :

1. une fonction `int length(tdyn x)` qui renvoie la longueur du tableau dynamique `x`.
2. Une fonction `int get(tdyn x, int i)` qui renvoie la $i$ème valeur du tableau `x` et une fonction `set(tdyn *x, int i, int v)` qui met la valeur `v` à la $i$ème case du tableau
3. une fonction `void create(tdyn *x, int n)` initialise le tableau dynamique à l'adresse x avec n zéros.
4. une fonction `void add(tdyn *x, int v)` qui ajoute une case au tableau contenant la valeur `v`. Il va falloir pour cela réserver `length+1` cases mémoires avec malloc, déplacer les cases du tableau vers cette nouvelle zone, nettoyer la mémoire où se trouvait l'ancienne copie du tableau et mettre à jour les informations dans le tableau dynamique.

Cette implémentation n'est pas très optimale car à chaque fois qu'on ajoute une nouvelle valeur, on recopie toutes les autres ailleurs dans la mémoire. Cela a pour conséquence que pour insérer n valeurs dans un tableau, on a fait $O(n^2)$ copies mémoire. C'est trop. Pour éviter cela, on va enrichir un peu notre structure :

```
struct tdyn {
  int length;
  int reserved;
  int *t;
};
typedef struct tdyn tdyn;
```

L'attribut `length` contient toujours le nombre d'éléments du tableau. L'attribut `reserved` quant à lui, contient le nombre de case réservée par notre dernier `malloc`. Bien sûr, on veut que `length` $\leq$ `reserved` mais l'idée sous-jacente c'est qu'à chaque ajout d'élément, soit on a encore de la place réservée et dans ce cas il suffit de mettre à jour le tableau sans rien recopier. Soit on a `length == reserved` et alors on va réserver plus de mémoire pour prévoir les éventuelles prochaine copies. On va choisir de réserver deux fois plus de mémoire que la longueur actuelle du tableau lorsqu'on doit déplacer des données. Ainsi, pour insérer n éléments, on ne fait que $n + n/2 + n/4 + ... = O(n \log(n))$ copies soit un coup d'insertion en moyenne de $O(\log(n))$.

Exercice 4. Reprendre l'exercice 3 en implémentant cette nouvelle manière de recopier les données.

Pour le 9 Novembre 2015

Lire le sujet du projet, former les binômes, commencer à y réfléchir et préparer des questions.