Programmation Avancée

TP n°4 : Structures sur le tas

Simon Forest

11 février 2021

Exercice 1 : Échauffement

Le but de cet exercice est de vous familiariser avec l'utilisation de `malloc` et `free` dans un exemple très simple.

1. Écrire un programme qui demande à l'utilisateur d'écrire son nom et son âge en entrée et qui réaffiche ensuite ces informations. **Votre programme ne pourra introduire que des variables qui sont des pointeurs, de sorte que vous serez obligés d'utiliser `malloc` pour les allouer. Il faudra aussi les libérer en utilisant `free`.**
2. Introduire la structure `personne`

   ```
   typedef struct personne{
     char* nom;
     int age;
   } personne;
   ```

   et écrire un programme qui demande en entrée un entier n, et qui demande à l'utilisateur de fournir le nom et l'âge de n personnes (chacune de ces informations sera enregistrée dans une structure `personne`), et qui affiche ensuite ces informations à nouveau. **Comme avant, votre programme ne pourra utiliser que des pointeurs. Il faudra en particulier allouer un tableau de `personne` de taille n pour stocker les informations rentrées par l'utilisateur. Il faudra aussi correctement désallouer avec `free` tout ce qui a été alloué avec `malloc`.**

Exercice 2 : Tableaux à taille dynamique

Vous avez vu jusqu'ici comment utiliser des tableaux avec des tailles statiques (qui sont définies à la compilation). Dans cet exercice, vous allez implémenter des tableaux dont la taille peut changer. On va se focaliser sur les tableaux d'entiers mais la structure s'adapte naturellement à n'importe quel type. On va utiliser la structure

```
typedef struct array{
  int* ptr;
  int size;
} array;
```

pour représenter les tableaux.

1. Écrire une fonction `void array_init(array* t, int n)` qui initialise une structure `array` à un tableau de taille `n` (on allouera le tableau pointé par `t->ptr` en utilisant `malloc`).
2. Écrire une fonction `void array_destroy(array* t)` qui libère le tableau stocké dans `t->ptr` (on utilisera la fonction `free` pour cela).

3. Écrire les fonctions

   `int array_get(array* t, int index)`

   et

   `void array_set(array* t, int index, int valeur)`

   qui permettent de lire et d'écrire une valeur dans un tableau à un certain index.
4. Écrire une fonction `void array_insert(array* t, int index, int valeur)` qui permet d'insérer une valeur à un index dans le tableau. La nouvelle valeur devra être insérée entre les cases aux indexes `index-1` et `index`. Ainsi, la fonction `array_insert` devra allouer une nouvelle plage mémoire avec `malloc` et recopier l'ancien tableau dans le nouveau en positionnant correctement l'élément à ajouter (si la taille du tableau est n avant l'insertion, la taille après devra être $n+1$).
5. Écrire une fonction `void array_erase(array* t, int index)` qui supprime une case du tableau (ici aussi, il faudra recopier le tableau dans un nouveau qui aura une case de moins).
6. Écrire une fonction `void array_print(array* t)` qui affiche le contenu du tableau en séparant les nombres affichés par des espaces.
7. Tester le code avec le main suivant :

```c
int main()
{
  array t;
  array_init(&t,10);
  for(int i = 0; i < 10; i++)
  {
    array_set(&t,i,i);
  }
  for(int i = 0; i < 10; i++)
  {
    if(array_get(&t,i) != i)
    {
      fprintf(stderr,"erreur: set ou get non correcte\n");
      return 0;
    }
  }
  for(int i=3; i >= 0; i--)
  {
    array_erase(&t,i*3);
  }
  array_print(&t);
  printf("\n");
  // devrait afficher 1 2 4 5 7 8
  for(int i = 0; i <= 3; i++)
  {
    array_insert(&t,i*3,i*3);
  }
  array_print(&t);
  printf("\n");
  // devrait afficher 0 1 2 3 4 5 6 7 8 9
  return 0;
}
```

Exercice 3 : Listes chaînées

Dans cet exercice, on implémente une autre structure classique qui est celle de *liste chaînée*. Une telle liste est représentée en mémoire par un ensemble de nœuds qui contiennent les éléments de la liste et qui sont reliés entre eux par des pointeurs. Plus précisément, le nœuds i de la liste a un champ `suivant` qui pointe sur le nœud $i+1$ de la liste. Par exemple, la liste $[1, 2, 3]$ est représentée par trois nœuds représentés en memoire comme sur le diagramme

où le dernier nœud pointe sur `NULL` pour indiquer que c'est le dernier.

1. Déclarer une structure `node` qui représente un nœud d'une liste chaînée. Cette structure contiendra un champ `value` représentant un entier stocké, ainsi qu'un pointeur `next` vers un autre `noeud`.
2. Écrire une structure `list` représentant une liste chaînée. Cette structure contiendra un pointeur `head` vers le premier nœud de la liste (c'est-à-dire, une liste dont le pointeur `head` est `NULL`).
3. Écrire une fonction `void list_init(list* l)` qui initialise une liste vide.
4. Écrire une fonction `void list_add_front(list* l, int v)` qui ajoute un nœud en tête de liste contenant la valeur de `v`. Après cet ajout, le pointeur `head` de la liste devra pointer vers le nouveau nœud ajouté à la liste, et le pointeur `next` du nouveau nœud aura l'ancienne valeur de `head`.
5. Respectivement, écrire une fonction `int list_pop_front(list* l)` qui supprime le premier élément de la liste et renvoie sa valeur.
6. Écrire une fonction `int list_length(list* l)` qui calcule la taille d'une liste (le dernier nœud d'une liste est celui dont le champ `next` est nul).
7. Écrire une fonction `void list_free(list* l)` qui libère le contenu d'une liste. Cette fonction devra désallouer les nœuds contenus dans la liste.
8. Écrire une fonction `int list_get(list* l, int index)` qui renvoie la valeur du nœud à la position `index` dans la liste.
9. Écrire une fonction `void list_insert(list* l, int index, int v)` qui ajoute un nœud à l'index donné en argument dans la liste et contenant la valeur `v`.
10. Écrire une fonction `void list_erase(list* l, int index)` qui supprime le nœud à l'index indiqué de la liste.
11. Écrire une fonction `void list_print(list* l)` qui affiche les éléments d'une liste en les séparant par des espaces.
12. Tester le code avec le main suivant :

```c
int main()
{
  list l;
  list_init(&l);
  list_push_front(&l,3);
  list_push_front(&l,2);
  list_push_front(&l,1);
  list_push_front(&l,0);
  list_print(&l);
  printf("\n");
  // doit afficher 0 1 2 3
```

```
  printf("pop: %d\n",list_pop_front(&l));
  // doit afficher pop: 0
  list_push_front(&l,0);
  list_insert(&l,4,9);
  list_print(&l);
  printf("\n");
  // doit afficher 0 1 2 3 9
  for(int i = 4; i <= 8; i++)
  {
    list_insert(&l, i, i);
  }
  list_print(&l);
  printf("\n");
  // doit afficher 0 1 2 3 4 5 6 7 8 9
  printf("taille de la liste: %d\n", list_length(&l));
  // doit afficher taille de la liste: 10
  printf("valeur à l'index 3: %d\n", list_get(&l,3));
  // doit afficher valeur à l'index 3: 3
  list_erase(&l,9);
  list_erase(&l,5);
  list_erase(&l,0);
  list_print(&l);
  printf("\n");
  // doit afficher 1 2 3 4 6 7 8
  list_free(&l);
  return 0;
}
```