Programmation : TP 8

Juliusz Chroboczek

22 novembre 2022

Un *tableau Pascal* est une paire consistant d'un tableau de taille n et de l'entier n. Dans les exercices qui suivent, on représentera un tableau Pascal d'entiers par une structure :

```
struct array {
    int *a;
    int len;
}
```

où `len` est la longueur du tableau, et est à la fois le nombre d'éléments du tableau et la taille de l'allocation du champ `a`.

Exercice 1. Dans cet exercice, ne testez vos fonctions qu'à partir de la question 4.

1. Écrivez une fonction

   ```
   struct array *new_array(int n);
   ```

   qui retourne un tableau Pascal fraîchement alloué de taille `n` et dont tous les éléments valent 0. (Pensez à la gestion des erreurs : si le deuxième `malloc` échoue, il faut libérer les données allouées par le premier.)
2. Écrivez une fonction

   ```
   void destroy_array(struct array *a);
   ```

   qui libère *toute* la mémoire occupée par un tableau (attention, il faudra faire deux appels à `free`, dans le bon ordre).
3. Écrivez une fonction

   ```
   struct array *read_array(int n);
   ```

   qui lit `n` entiers et les stocke dans un tableau Pascal qu'elle retourne.
4. Écrivez une fonction

   ```
   void print_array(struct array *a);
   ```

   qui affiche le contenu du tableau Pascal `a`.

Écrivez une fonction `main` pour tester vos quatre fonctions. Testez à l'aide de `valgrind` qu'il n'y a pas de fuite de mémoire.

5. Écrivez une fonction

   ```
   void print_array_inverse(struct array *a);
   ```

   qui affiche un tableau dans l'ordre inverse de ses éléments. Écrivez un programme qui lit un entier *n*, puis qui lit *n* entiers, puis affiche ces derniers dans l'ordre inverse. Testez à l'aide de `valgrind`.

Un *slice* est une variation sur les tableaux Pascal qui contient non seulement la taille du tableau mais aussi la taille de l'allocation. Dans les exercices qui suivent, on représentera un *slice* par une structure :

```
struct slice {
    int *a;
    int len;
    int cap;
}
```

où `len`, la *longueur*, est le nombre d'éléments du *slice*, et `cap`, la *capacité*, est la taille de l'allocation du champ `a` en unités de la taille d'un élément.

Exercice 2.

1. Écrivez une fonction

   ```
   struct slice *new_slice(int cap);
   ```

   qui retourne un *slice* fraîchement alloué de longueur 0 et de capacité `cap`.
2. Écrivez une fonction

   ```
   void destroy_slice(struct slice *s);
   ```

   qui libère la mémoire occuppée par un *slice*.
3. Écrivez une fonction

   ```
   int snoc(struct slice *s, int v);
   ```

   qui ajoute un élément de valeur `v` à la fin d'un *slice*. Cette fonction retournera 1 si elle réussit, et -1 s'il n'y a plus de place dans le *slice*.
4. Écrivez une fonction

   ```
   struct slice *read_slice();
   ```

   qui alloue un *slice* de capacité 10 puis lit des entiers jusqu'à ce que l'utilisateur rentre une valeur négative et les stocke dans le *slice* qu'elle retournera (sans le -1 de la fin). Si l'utilisateur entre plus de 10 nombres, votre fonction libérera toute la mémoire allouée puis retournera `NULL`.
   Écrivez un programme pour tester vos fonctions. Vérifiez que `valgrind` est satisfait, aussi bien dans le cas où la lecture réussit que dans le cas où on déborde.