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.

5. Écrivez une fonction

   ```
   int extend(struct slice *s);
   ```

   qui augmente l'allocation d'un *slice* sans en changer le contenu. Soit n défini par

   $n = 2 \times$ `s->cap` si `s->cap` > 0;
   $n = 4$ sinon.

   Votre fonction allouera un tableau de taille n, puis y copiera les `s->len` premiers éléments de `s->a`. Elle libérera ensuite `s->a`, puis affectera le tableau nouvellement affecté à `s->a`. Enfin, elle mettra à jour la valeur de `s->cap`.
6. Modifiez votre fonction `snoc` pour qu'elle appelle la fonction `extend` lorsque l'allocation est trop petite. Testez que votre programme supporte maintenant les suites de longueur arbitraire (dans la limite de la mémoire disponible). Que dit `valgrind` ?

Exercice 3. Une *pile* est une suite de valeurs sur laquelle seules trois opérations sont autorisées : *empty*, qui retourne vrai si la pile est vide, *push*, qui ajoute un élément en queue de suite, et *pop*, qui supprime l'élément en queue de suite et retourne ce dernier. On appelle *sommet* de la pile l'élément en queue.

1. Écrivez trois fonctions

   ```
   int empty(struct slice *a);
   int push(struct slice *a, int v);
   int pop(struct slice *a);
   ```

   qui implémentent les opérations sur une pile. La fonction `push` retourne 1 en cas de succès, -1 en cas d'échec. La fonction `pop` appelle `abort` si on l'appelle sur une pile vide.

Un caractère ASCII (mais pas un caractère accentué, chinois ou syriaque) peut être représenté par une valeur de type `char`. On représente une constante par le caractère entouré d'apostrophes « `'a'` », et le descripteur de format correspondant est « `%c` ». Si un caractère `c` est compris entre `'0'` et `'9'`, on peut le convertir en l'entier qu'il représente à l'aide de l'expression « `c - '0'` ».

2. Écrivez un programme alloue une pile vide puis effectue les actions suivantes dans une boucle infinie :
   - Lit un caractère à l'aide de la fonction `getchar` ;
   - si ce caractère représente un entier, il empile l'entier correspondant ;
   - si ce caractère est un point « `'.'` », il vérifie si la pile est vide ; si c'est le cas, il affiche un message d'erreur ; si ce n'est pas le cas, il dépile un élément et l'affiche ;
   - si ce caractère est un espace « `' '` » ou un caractère de fin de ligne « `'\n'` », il ne fait rien ;
   - si ce caractère est un « `'q'` », il libère la pile et termine l'exécution du programme (vous pouvez utiliser l'instruction `break`) ;
   - sinon, il affiche un message d'erreur.

   Testez votre programme, par exemple sur la chaîne « `12 3.4. ..q` », et vérifiez que `valgrind` n'a pas d'objection.

La notation arithmétique usuelle « 2 + 3 » s'appelle aussi la *notation infixe*. Dans la notation *postfixe* ou *polonaise inverse*, une expression s'écrit en écrivant d'abord les opérandes et ensuite l'opérateur « `2 3 +` ». L'avantage principal de la notation polonaise inverse est qu'on peut l'évaluer à l'aide d'une seule pile (il faut deux piles pour évaluer une expression en notation arithmétique).

3. Écrivez chacune des expressions suivantes en notation polonaise inverse :

$$2 + 3 - 4 \qquad (2 + 3) - 4 \qquad 2 + (3 - 4)$$

4. Ajoutez les actions suivantes dans la boucle infinie du programme les actions suivantes dans la boucle infinie :
   - si *c* caractère est un « `'+'` », il vérifie que la pile contient au moins deux éléments, dépile deux éléments, puis empile leur somme ;
   - de même, si *c* caractère est un « `'-'` », il vérifie que la pile contient au moins deux éléments, dépile deux éléments, puis empile leur différence.

   Testez votre programme sur chacune des expressions ci-dessus. Assurez-vous que `valgrind` vous aime encore.
5. Ajoutez la gestion des opérations « `*` », « `/` », « `%` » et « `~` » (opposé unaire) à votre programme.