Enseignants : A. Bonhomme, L. Bougé
Email: {Alice.Bonhomme,Luc.Bouge}@ens-lyon.fr

le 04/12/00

TD Programmation 9 : Allocation dynamique (suite)

1 Allocation d'un tableau à deux dimensions

L'objectif de cet exercice est de travailler sur les différentes déclarations possibles d'un tableau à deux dimensions, et ce que cela implique sur la gestion de la mémoire.

Un tableau à deux dimensions (N lignes et M colonnes) peut être déclaré des trois manières suivantes :

```
#define N 4
#define M 2

/* allocation statique */
int static_tab[N][M];

/* allocation semi-statique */
int * semi_static_tab[N];

/* allocation dynamique */
int ** dynamic_tab;
```

Quelle que soit la déclaration du tableau, les éléments pourront être accèdés par `tab[i][j]`. Toutefois, l'emplacement des éléments du tableau ne seront pas situés au même endroit dans la mémoire suivant les définitions.

1.1 Allocation statique

- Écrire une fonction `init_static_tab` qui initialise tous les éléments du tableau à 0.
- Écrire une fonction `print_static_tab` qui imprime pour chaque case `tab[i][j]` du tableau les informations suivantes : adresse de la case, valeur de `i`, valeur de `j`, valeur de l'élément de la case. (*aide :* utiliser le format `% p` pour afficher une adresse en hexadécimal). Tester cette fonction sur le tableau initialisé.

- On définit une variable `int line[M]`. On souhaite copier les élément de `line` dans une ligne du tableau.
  - Écrire une fonction `copy_line_in_static_tab (int * linesrc, int, line_size, int i)` qui copie la ligne `line` de taille `line_size` dans la ième ligne du tableau, à l'aide d'une boucle `for`.
  - Initialiser `line` avec des valeurs autres que 0, et tester cette fonction en imprimant ensuite le tableau avec `print_static_tab`.
  - Écrire une fonction `copy_line_bis_in_static_tab( int * line, int line_size, int i)` qui copie la ligne `line` à l'aide de la fonction `memcpy` (pour la syntaxe, `man memcpy`). Cette fonction permet de copier tout un espace mémoire source vers un espace mémoire cible.
  - Comme précédemment, tester cette fonction.

1.2 Allocation semi-statique

- Écrire une fonction `alloc_semi_static_tab` qui alloue la partie dynamique du tableau.
- Écrire une fonction `free_semi_static_tab` qui libère ce tableau. (**Attention :** *dans vos programme, il faut toujours libérer les données que vous allouez!!)*
- Écrire une fonction `init_semi_static_tab` qui initialise le tableau à 0, puis la fonction `print_semi_static_tab`. Pourquoi ne peut-on pas utiliser la même fonction que pour `static_tab` en passant le tableau par paramètre? (si vous ne voyez pas, passer à la question suivante, elle vous aidera à comprendre).
- Initialiser le tableau, puis imprimer le. Que remarquez-vous? Comparez par rapport aux résultats obtenus pour `static_tab`.
- Écrire les fonctions `copy_line_in_semi_static_tab (int * linesrc, int, line_size, int i)`, et `copy_line_bis_in_semi_static_tab( int * line, int line_size, int i)`.
- Écrire une fonction `replace_line_in_semi_static_tab (int * line, int i)` qui remplace **physiquement** la ligne `i` du tableau par celle passer en paramètre.
- Tester cette fonction avec une ligne non nulle. Imprimer le tableau ainsi modifié. Que remarquez vous?
- Modifier la ligne `line` et imprimer le tableau. Que remarquez-vous? est-ce normal?

1.3 Allocation dynamique

- Écrire une fonction d'allocation du tableau `alloc_dynamic_tab`.
- Écrire une fonction de libération du tableau `free_dynamic_tab`.

- Reprendre les fonctions écrites pour `semi_static_tab` en passant le tableau par paramètre et pour que ces fonctions soient utilisables avec `dynamic_tab`. Pourquoi ceci est-il possible? Quelle est alors la différence entre ces deux déclarations?

- Tester ces nouvelles fonctions avec les deux tableaux.

1.4 Débordement dans un tableau

Un erreur classique de programmation est "d'aller trop loin dans un tableau", c'est-à-dire d'accèder à une case $i \geq n$ alors que le tableau comporte uniquement n cases.

Nous cherchons à voir ce qui se passe dans ce cas là, en fonction du type d'allocation du tableau. Pour ceci, on va utiliser la fonction `copy_line` avec comme argument, une taille de ligne supérieure à celle du tableau.

On utilise donc une variable `int line[M+1]` et on va copier cette ligne dans le tableau et analyser le résultat.

1.4.1 Allocation statique

Initialiser le tableau à 0, puis la ligne `line` à des valeurs différentes de 0. Copier la ligne dans le tableau, puis imprimer le tableau. Que remarque-t-on? Pourquoi?

1.4.2 Allocation semi_statique et dynamique

Effectuer les mêmes opérations que précedemment, mais avec les tableaux semi_statique et dynamique. Que se passe-t-il?

1.5 Redimensionnement d'un tableau

On souhaite maintenant pouvoir changer la taille d'un tableau, après son allocation. La fonction `realloc` permet de changer la taille d'un espace mémoire alloué précédemment par la fonction `malloc` (`man realloc` pour son utilisation).

1.5.1 Allocation statique

Dans le cadre d'un tableau alloué statiquement, il n'est pas possible de redimensionner un tableau. En effet, celui-ci n'a pas été alloué par un appel à la fonction `malloc`. De plus, l'espace mémoire réservé pour ce tableau, est situé dans la pile, et non dans le tas comme le sont les espace mémoires alloués dynamiquement.

1.5.2 Allocation semi_statique

Dans ce type d'allocation, le redimensionnement est possible mais limité, car comme son nom l'indique, une partie du tableau est allouée statiquement, et une partie dynamiquement. Ainsi on peut augmenter le nombre de colonnes mais pas le nombre de lignes.

- Écrire une fonction `realloc_semi_static_tab(int nb_col)` qui redimensionne le tableau semi_statique pour qu'il contienne `nb_col` colonnes.

- Tester cette fonction en initialisant le tableau réalloué, et en l'imprimant.

- Reprendre la fonction `copy_line_in_static_tab`, en traitant les cas de débordements, c'est-à-dire, en agrandissant suffisamment le tableau pour qu'il n'y ait plus de débordement.

- Tester cette fonction avec des débordements de taille supérieure à 1.

1.5.3 Allocation dynamique

Dans le cadre d'une allocation dynamique, on peut tout redimensionner : colonnes et lignes.

- Écrire une fonction `realloc_dynamic_tab(int nb_col, int nb_line)` qui redimensionne le tableau dynamique pour qu'il contienne `nb_col` colonnes et `nb_line` lignes (Attention, la valeur de la variable `dynamic_tab` va changer!!).

- Tester cette fonction.

1.6 Allocation dynamique et contigüe d'un tableau

On souhaite maintenant allouer dynamiquement un tableau à deux dimensions, mais avec un espace mémoire contigü, ce qui n'est le cas dans aucune des déclaration vues précédemment. L'allocation statique procure bien un espace mémoire contigü, mais il n'est pas alloué dynamiquement. Les deux autres allocations ne procurent pas un espace mémoire contigü.

Pour ceci, on va allouer dynamiquement une zone mémoire de taille `N * M`. Ceci peut être possible uniquement si l'allocation se fait à l'aide d'un seul appel à la fonction `malloc`. Une solution est alors d'aplatir le tableau à deux dimensions en un tableau à une dimension. Les `M` premières cases seront les valeurs de la premières lignes. Ainsi, la `jème` ligne débutera à la case `(j*M)-1`.

- Définir le type de ce tableau, et écrire les fonctions d'allocation et de libération pour ce tableau.

- Écrire les fonctions de manipulation de ce tableau (`init, print, copy`)

- Tester les cas de débordements.

- Écrire une fonction de redimensionnement de ce tableau (Attention : la ré-allocation du tableau est simple, mais les données après la réallocation ne seront plus à leur place dans le tableau! il faudra les remettre à leur place).

- Tester la fonction de redimensionnement en augmentant le nombre de colonnes et le nombre de lignes sur un tableau contenant déjà des données.

- Comparer cette allocation à l'allocation statique. Quelle est la principale différence?

2 Gestion d'exceptions en C

En C, et sous Unix, la gestion des exceptions est faite à l'aide d'échanges de signaux. Les signaux peuvent être émis soit par le système qui détecte un problème (division par 0, accès erroné à la mémoire, etc.), soit par le programme lui-même à l'aide de la fonction `raise`, soit par d'autres processus à l'aide de la commande `kill`. Les signaux peuvent être rattrapés et traités grâce à la fonction `signal` qui rattrape un signal précis et appelle une fonction de traitement. Lorsqu'un signal est rattrapé par la fonction `signal`, il est ensuite traité puis l'exécution reprend au point d'interruption.

Pour l'utilisation de ces fonctions, `man raise, man signal`. La liste des signaux est définie dans le fichier en-tête `<signal.h>`.

- Écrire un petit programme qui rattrape les erreurs arithmétiques, et dans ce cas écrit à l'écran un message d'erreur et quitte le programme. Le signal a attrapé est `SIGFPE`. Tester ce programme avec une division par 0 par exemple.

- Écrire un programme qui rattrape les erreurs d'accès illégal à la mémoire(signal `SIGSEGV` (segmentation violation). Tester ce programme sur un accès à un pointer `NULL`.

- Il est aussi possible de récupérer des signaux émis par des caractères saisis au clavier (comme `C-c`, représenté par le signal `SIGINT`). Écrire un programme qui rattrape une interruption émise par le clavier, imprime un message à l'écran puis quitte le programme.

- Reprendre la gestion du débordement dans la fonction `copy_line` de l'exercice précédent à l'aide de signal. Lorsqu'un débordement est détecté, un signal (SIGSEGV) est émis, via la fonction `raise`. Ce signal est ensuite rattrapé. Le traitement du signal est alors de redimensionner le tableau.