Programmation C
TP 6

Pile d'exécution et récursion.

Toute fonction appelée (par une fonction père) en C se place sur la pile d'exécution avec ses arguments et ses variables locales. Une fois la fonction terminée, seul le résultat (retour de la fonction) perdure sur la pile. Ce même résultat, s'il n'est utilisé par la fonction père (l'appelant) sera perdu. Nous allons, durant ce TP, voir un peu ce qui se passe sur cette pile.

Exercice 1 Hauteur de pile

Télécharger depuis la plate-forme d'enseignement en ligne le code source `stack_height.c` pour se TP. Compiler et exécuter ce code en donnant un petit argument entier au programme.

Lire un peu le code source de ce programme :

`printf("%p", adr);` affiche l'adresse en hexadécimale d'un pointeur. Ce pointeur peut être n'importe quoi : un `int*`, un `char*`, un `float*`, un `void*`, ... Ce doit juste être un pointeur. Cette adresse correspond complètement à un endroit de la mémoire RAM.

Dans le langage C, quand on calcule la différence entre deux adresses pointant vers un même type, on obtient un entier long (C'est pour cela que l'affichage utilisera `"%ld" long decimal number`). Ce nombre est l'écartement entre les deux zones mémoires pointées par les pointeurs. Cet écartement dépend du type. Pour des pointeurs vers des entiers, un écartement de 6 signifie qu'il y a 24 octets d'écart sur la RAM entre les deux zones (car `sizeof(int)` = 4 sur les machines récentes).

Après avoir bien testé ce programme avec différentes valeurs, rajouter une ou plusieurs grosses variables locales et augmenter le nombre de récursions. Faites exploser la pile jusqu'à la `Segfault`.

Pour les plus curieux, la taille de la pile est de 8 Mégaoctets par défaut sur les UNIX courant. Tentez un calcul.

Exercice 2 Lister les permutations

On veut générer, et en fait uniquement afficher ici, toutes les permutations de l'ensemble {1,2, . . . , N}. N sera ici pour nous une constante que l'on définira pour le préprocesseur avec `#define`.

Pour cela, on va remplir un tableau de taille N en plaçant successivement les entiers dans un tableau où les cases sont vides. Une case du tableau sera considérée comme vide lorsqu'elle contient la valeur 0.

On écrit pour cela une fonction récursive qui prend en paramètres :

- un tableau d'entiers (initialement rempli de 0),
- un entier représentant la valeur courante à mettre dans le tableau,
- un entier représentant la valeur maximale à placer.

A chaque appel, on regarde si la valeur courante est supérieure à la valeur maximale à placer, si c'est le cas, on affiche le tableau et on a fini (avec une autre fonction qui affiche un tableau selon sa taille : 1 tache = 1 fonction). Sinon, on regarde case par case si la place est libre et si c'est le cas, on place la valeur et on fait un appel récursif pour placer le reste. Au retour de cet appel récursif, on libère la case (en remettant le 0) et on cherche la case suivante libre.

Écrire dont une fonction `void permutations(int buffer[], int current, int max)` qui affiche les permutations de l'ensemble $\{1, 2, \ldots, max\}$.

Voici l'arbre des appels pour la fonction récursive qui énuméra toutes les permutations de taille 3 :

```
[0,0,0]
 --> [1,0,0]
      --> [1,2,0]
           --> [1,2,3]
      --> [1,0,2]
           --> [1,3,2]
 --> [0,1,0]
      --> [2,1,0]
           --> [2,1,3]
      --> [0,1,2]
           --> [3,1,2]
 --> [0,0,1]
      --> [2,0,1]
           --> [2,3,1]
      --> [0,2,1]
           --> [3,2,1]
```

Exercice 3 **Solver de Sudoku**

Générer les permutations n'était pas drôle ? Le même algorithme permet de faire un solver de sudoku.

On part d'une grille 9 par 9 partiellement remplie. Le 0 caractérisera les cases vides. On programme ensuite une fonction récursive qui affiche la grille si cette dernière est pleine ou sinon recherche une case libre. Une fois une case libre trouvée, soit il est possible d'y mettre une valeur entre 1 et 9, alors on fait un appel récursif pour continuer à remplir la grille, soit toutes les valeurs entre 1 et 9 produiraient une incohérence, alors il faut arrêter l'exécution de la fonction avec `return ;` enlever la dernière valeur mise (qui est donc mauvaise) et continuer de chercher.

Dans une infinie bonté, nous avons mis à votre disposition sur la plate-forme d'enseignement en ligne une ébauche de programme pour lire des grilles de Sudoku de difficultés différentes placées dans un dossier `data`. Lire un peu ce code source pour savoir comment le manipuler.

Écrire une fonction solver de Sudoku dans le module `sudoku`. Tentez de résoudre les 4 premières grilles (`grid1.txt`, `grid2.txt`, `grid3.txt` et `grid4.txt`). Ces dernières grilles n'ont qu'une seule et unique solution.

Et si vous avez bien programmé : déterminer le nombre de solutions possibles à la grille 5 : `grid5.txt`.

Pour la première grille, vous devriez obtenir :

```
nborie@perceval$ ./sudoku data/grid1.txt
-------------------------------------
|   |   |   |   |   |   |   | 4 |   |
-------------------------------------
| 5 |   |   | 1 |   |   | 6 | 3 |   |
-------------------------------------
|   | 7 |   | 8 | 6 |   |   |   | 2 |
-------------------------------------
| 8 | 5 |   |   | 3 |   | 4 |   |   |
-------------------------------------
| 3 | 9 | 7 |   | 8 |   | 2 | 5 | 6 |
-------------------------------------
|   |   | 2 |   | 7 |   |   | 8 | 3 |
-------------------------------------
| 7 |   |   |   | 4 | 9 |   | 6 |   |
-------------------------------------
|   | 4 | 9 |   |   | 7 |   |   | 8 |
-------------------------------------
|   | 3 |   |   |   |   |   |   |   |
-------------------------------------


-------------------------------------
| 1 | 6 | 3 | 7 | 2 | 5 | 8 | 4 | 9 |
-------------------------------------
| 5 | 2 | 8 | 1 | 9 | 4 | 6 | 3 | 7 |
-------------------------------------
| 9 | 7 | 4 | 8 | 6 | 3 | 5 | 1 | 2 |
-------------------------------------
| 8 | 5 | 6 | 9 | 3 | 2 | 4 | 7 | 1 |
-------------------------------------
| 3 | 9 | 7 | 4 | 8 | 1 | 2 | 5 | 6 |
-------------------------------------
| 4 | 1 | 2 | 5 | 7 | 6 | 9 | 8 | 3 |
-------------------------------------
| 7 | 8 | 1 | 2 | 4 | 9 | 3 | 6 | 5 |
-------------------------------------
| 6 | 4 | 9 | 3 | 5 | 7 | 1 | 2 | 8 |
-------------------------------------
| 2 | 3 | 5 | 6 | 1 | 8 | 7 | 9 | 4 |
-------------------------------------
```