TP - 6

09 octobre 2014

On apprend aujourd'hui à se battre contre les `segmentation fault` à l'aide de *gdb* et *ddd*. Ce sera également l'occasion de renforcer votre compréhension des pointeurs.

1 gdb

L'outil *gdb* (*i.e.,* GNU debugger) et un débogueur pour C, C++ et Fortran. Il permet d'exécuter pas à pas votre programme, lire ou changer l'état de la mémoire pendant l'exécution (variables, arguments, registres, etc.) , voir le code assembleur généré (et l'exécuter pas à pas), etc. Tout ceci nous permet de comprendre ce que fait notre programme *en pratique* et de le tester. Vous vous en servirez surement pour comprendre pourquoi vos futurs programmes se plantent (qui a dit `Segmentation fault` ?). L'utilitaire *ddd* fournit une interface graphique à *gdb*. C'est l'outil que nous utiliserons aujourd'hui.

Exercice - 1 *Première approche*

1- Ecrivez une fonction qui affiche les entiers de 1 à 10 (allez en 3 minutes ;)).
2- Compilez votre programme en utilisant l'option `-ggdb` (ex. `$ gcc -ggdb -o exe.out programme.c`). Puis lancez *ddd* sur votre exécutable (ex. `$ ddd exe.out`). Une jolie fenêtre s'affiche, dites boujour à *ddd*. Vous devez voir votre code source au milieu. En-dessous, vous avez accès à *gdb* directement. Dans une petite fenêtre vous avez des boutons permettant de lancer le programme (`Run`), l'exécuter pas à pas (`Next`), etc.
3- Dans les menus, cochez Data/Display Arguments ainsi que Data/Display Local variables. Une grille devrait apparaître (qui affichera les états de la mémoire plus tard) au-dessus. Faites un clic droit au début de la ligne contenant votre `printf` dans la boucle et faites `set breakpoint`. Cliquez maintenant sur `Run`. Vous devez voir une flèche verte se placer sur l'instruction du break point. L'exécution est mise en pause. Vous pouvez exécuter le programme pas à pas en cliquant sur `Next` et admirer votre compteur augmenter.

Exercice - 2 *Retour sur `MinMax`*

L'objectif de cet exercice est de comprendre pourquoi un programme que plusieurs de vos collègues ont écrit ne fonctionne pas. Il s'agit d'une version buggée de `MinMax` (exo 1, qu. 4 du TP5). Pour commencer doucement on va étudier avec *ddd* deux version correctes de cette fonction.
1- Récupérez le fichier `ddd_pointeurs.c` à cette adresse. Compilez-le (avec l'option de debug). Lisez le code puis lancez-le et observez le comportement bizzare de la dernière fonction... On va déboger ça. Lancez *ddd* sur l'exécutable.
2- Faites en sorte d'afficher les variables locales ainsi que les arguments. Placez des break points sur chaque ligne `for` et sur chaque appel de fonction. Puis lancez le programme.
3- Vous devez voir s'afficher la flèche verte ainsi que les variables locales juste avant le premier appel. Essayez de comprendre tout ce qui s'affiche. Faites `Next`. Vous pouvez faire afficher à *ddd* plus de détails sur les variables et arguments en faisant clic-droit puis `Show All` ou `Display(*)`. Par exemple, en le faisant sur les pointeurs `min` et `max` ainsi que sur `tab`, *ddd* vous affiche une flèche vers la valeur pointée par ces pointeurs. Dans la suite, faites le dès que nécessaire. Amusez vous avec `Next` jusqu'à rentrer dans la seconde fonction.

4- Une fois que vous êtes dans `MinMax_p` vous devez voir les doubles pointeurs `min` et `max`. Faites clic-droit/`Display(*)` sur ces deux pointeurs, vous devez obtenir deux flèches vers deux nouvelles cases. Faites la même chose sur ces nouvelles cases. Affichez également le pointeur `tab`. Comprenez vous ce qu'il se passe quand vous faites `Next` ? Quelle est la différence *en pratique* entre `MinMax` et `MinMax_p` ?
5- Placez-vous dans la fonction `MinMax_dummy`. Observez les pointeurs. Comprenez-vous maintenant pourquoi ça ne marche pas ? Comment feriez-vous pour corriger ce code ?

2 Pointeurs (2)

On revient maintenant sur les pointeurs avec quelques exercices. Si vous le voulez, vous pouvez utiliser *ddd* pour comprendre ce qu'il se passe.

Exercice - 3 *Arithmétique des pointeurs*

Le but de cet exercice est de comparer différentes méthodes de stockages de matrices.
1- Ecrivez une fonction qui réalise le produit matriciel. Votre fonction doit avoir le prototype suivant :

```
void prodMatT (int *a, int *b, int *c, int m, int n).
```

C'est à dire que les matrices sont stockées dans un grand tableau.
2- La même chose mais cette fois, votre fonction doit avoir le prototype suivant :

```
void prodMatM (int **a, int **b, int **c, int m, int n).
```

C'est à dire que les matrices sont stockées sous la forme d'un tableau de tableaux.

Exercice - 4 *Pointeurs et strings*

1- Implémentez votre propre fonction `strdup`. Pour savoir ce qu'elle fait, lisez son manuel (`$ man strdup`).
2- (Bonus) Même chose pour `strstr`.

3 Deux problèmes au choix

Exercice - 5 *Automate celullaire (pour produire ça)*

---

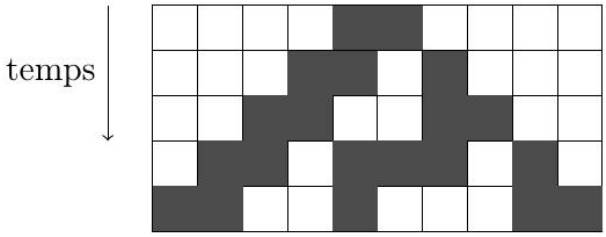
d'après Kevin Perrot et. al
Un automate cellulaire en une dimension consiste en une suite fini de N cellules contenant chacune un état (0 ou 1). L'état de la cellule d'indice i au temps $t+1$ est fonction de l'état des cellules $i-1 mod[N]$, i et $i+1 mod[N]$ au temps t (on considère une structure torique, c'est à dire que le voisin gauche de la cellule la plus à gauche est la cellule la plus à droite et inversement). Chacune des cellules pouvant prendre deux états, il existe $2^3 = 8$ configurations (ou motifs) possibles d'un tel voisinage. Pour que l'automate cellulaire fonctionne, il faut définir quel doit être l'état, à la génération suivante, d'une cellule pour chacun de ces motifs. Il y a $2^8 = 256$ façons différentes de s'y prendre, soit 256 automates cellulaires différents de ce type.

Les automates de cette famille sont dits "élémentaires". On les désigne souvent par un entier entre 0 et 255 dont la représentation binaire est la suite des états pris par l'automate sur les motifs successifs 111, 110, 101, etc.

Par exemple l'automate cellulaire élémentaire 30 (00011110 en binaire) est défini par les règles de transition suivantes :

111	110	101	100	011	010	001	000
0	0	0	1	1	1	1	0

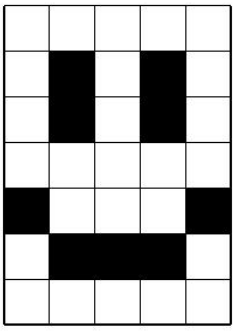
Les 4 premières itérations pour N = 10 et la configuration initiale 0000110000 donnent :



L'objectif de ce problème est de coder un programme qui demande à l'utilisateur un numéro de règle, un nombre de cellules et un nombre d'itération et qui produit une représentation graphique de l'évolution de l'automate (en prenant une configuration initiale aléatoire). Vous utiliserez le format d'images PPM pour produire vos images.

Un fichier PPM commence par la version du format (on utilisera P1) puis le nombre de lignes et de colonnes puis pour chaque ligne de l'image, une suite de 0 et de 1 codant les pixels. Voici un source à gauche et le rendu à droite :

```
P1
5 7
0 0 0 0 0
0 1 0 1 0
0 1 0 1 0
0 0 0 0 0
1 0 0 0 1
0 1 1 1 0
0 0 0 0 0
```



Exercice - 6 *Algorithme de Dijkstra (Bonus)*

Dans ce problème plus libre vous devez implémenter l'algorithme de Dijkstra calculant le plus court chemin dans un graphe. On supposera que le graphe est donné par une matrice d'adjacence. On codera la version une source vers toutes destinations. C'est-à dire que votre fonction doit calculer étant donné une graphe et un point de départ, le plus court chemin entre cette source et n'importe quelle destination.

1- Ecrivez une fonction qui génère un graphe aléatoire (utilisez la fonction `rand` de la librairie `stdlib.h`) sous la forme d'une matrice d'adjacence. Ecrivez une fonction qui affiche (joliment ?) un tel graphe.

2- Ecrivez la fonction implémentant l'algorithme de Dijkstra. Testez.

3- Ecrivez une fonction qui affiche (joliment) le résultat.

Pour le mercredi 15 octobre 12h Envoyez-moi vos solutions de l'exercice 3 (obligatoire) de l'exo 5 et/ou 6 (optionnel) par mail. Format habituel, un fichier par exercice. Je n'ouvrirai plus les archives ni les fameux `TP.c`.

Vos fichiers doivent : compiler sans erreur, être correctement et suffisamment commentés et contenir des tests qui affichent les résultats à l'exécution.

Pour le mercredi 15 octobre 12h Rappel : Vous devez me rendre votre solution à l'exo 3 du TP5. Même nomenclature que d'habitude.