TME1 : Outil de débogue, tableaux et complexité

Les deux exercices de ce premier TME permettent de découvrir deux techniques qui seront utiles pour l'ensemble des TME. Le premier exercice traite de techniques de débogue et de détection de fuite mémoire. Le deuxième exercice est l'occasion d'étudier une façon d'évaluer la vitesse d'un programme et de comparer les vitesses de plusieurs programmes. Les outils fournis pour ces deux exercices ne sont pas obligatoires et peuvent être remplacés par des outils équivalents.
Les fichiers nécessaires à ce TME peuvent être récupérés sur le site web de l'UE.

Exercice 1 – Utilisation d'un outil de débogue

La bonne compilation d'un programme écrit en C n'assure malheureusement pas son bon fonctionnement. Il existe souvent des erreurs qui ne sont révélées qu'au moment de l'exécution et qu'il faut trouver. Parmi ces erreurs, la plus courante est l'« erreur de segmentation » qui survient souvent lors de la manipulation de tableaux, ou de pointeurs. L'exercice suivant vous présente l'utilisation d'un outil de débogue qui vous permettra la plupart du temps de localiser vos erreurs.

Partie 1 : *indice de boucle*
Récupérez le fichier C nommé `gdb_exo1.c` :

```c
#include<stdio.h>
#include<stdlib.h>

const static int len = 10;

int main(void) {
    int *tab;
    unsigned int i;

    tab = (int*)malloc(len*sizeof(int));

    for (i=len-1; i>=0; i--) {
        tab[i] = i;
    }

    free(tab);
    return 0;
}
```

Q 1.1
À la lecture du code, qu'est censé faire le programme d'après vous ? Compilez le avec la commande suivante `gcc -o gdb_exo1 gdb_exo1.c` et lancer le. Que se passe-t-il ?

Q 1.2
Pour trouver l'erreur dans le programme, nous allons utiliser `gdb`, un outil de débogue très pratique. Recompilez votre programme avec la commande suivante : `gcc -ggdb -o gdb_exo1 gdb_exo1.c`. L'option `-ggdb` permet d'inclure dans l'exécutable des informations supplémentaires qui facilite le débogue. Lancez l'outil `gdb` sur l'exécutable de votre programme dans un terminal : `gdb gdb_exo1`.

Pour trouver l'erreur du programme, vous allez l'exécuter pas-à-pas et vous allez "donner la trace" (ou "tracer") de la variable de boucle `i`, c'est-à-dire examiner l'évolution de sa valeur dans le temps. Pour cela, commencer par définir un point d'arrêt au niveau de l'instruction qui se trouve dans la boucle d'exécution (`tab[i] = i;`), à l'aide de l'instruction `break 13` (13 indique le numéro de ligne de l'instruction).
Lancez ensuite l'exécution à l'aide de l'instruction *run*. Le programme devrait s'exécuter jusqu'au point d'arrêt. Lorsque l'exécution du programme est interrompue, on peut examiner l'etat de la mémoire à ce moment là, par exemple en affichant la valeur d'une variable à l'aide de la commande `print`. Pour tracer la variable `i`, exécutez la commande `print i`.
Vous pouvez maintenant exécuter le programme, c'est-à-dire instruction par instruction. La commande `step` permet d'avancer pas à pas dans l'exécution, afin de bien contrôler l'evolution du programme. Elle permet de passer à la ligne successive dans le source.
L'instruction `continue`, elle, relance l'exécution jusqu'au prochain point d'arrêt.
Faites alors itérer la boucle jusqu'au bout et suivez l'évolution de la valeur de `i` à l'aide de l'instruction `print i`.
Après l'itération dans laquelle `i` valait 0, que vaut la valeur de `i` ? Que devrait valoir `i` pour sortir de la boucle ? À quelle case du tableau essaie-t-on d'accéder ? Déduisez-en le sens de l'« erreur de segmentation ».

Q 1.3
Sachant que le mot clé `unsigned` force un type C standard (`char, short, int`) à ne jamais être négatif (c'est-à-dire « non signé »), que proposez-vous pour résoudre cette erreur ?

Voici ci-dessous un résumé des commandes importantes sous `gdb` :

- `quit (q)` : quitter `gdb`
- `run (r)` : lancer l'exécution
- `break,watch (b,w)` : introduire un point d'arrêt, ou bien "surveiller" une variable
- `clear,delete (cl,d)` : effacer un point d'arrêt. `clear` en indiquant un numéro de ligne ou nom de fonction, `delete` en indiquant le numéro du breakpoint (`delete` tout court efface – après confirmation – tous les points d'arrêt).
- `step,next,continue (s,n,c)` : avancer d'un pas (en entrant ou pas dans les sous-fonctions), relancer jusqu'au prochain point d'arrêt.
- `print,backtrace,list (p,bt,l)` : afficher la valeur d'une variable, la pile d'exécution (indiquant à quel endroit l'on se trouve au sein des différents appels de fonctions), afficher l'endroit où l'on se trouve dans le code.

Partie 2 : *Déférencement de pointeur*
Récupérez le fichier C nommé `gdb_exo2.c` :

```c
#include<stdio.h>
#include<stdlib.h>

struct mystruct {
    int value;
    struct mystruct *next;
};

struct mystruct* insert_list (struct mystruct *list, int value) {
    struct mystruct *new;
```

```
    new = malloc(sizeof(struct mystruct));

    new->value = value;
    new->next = list;

    return new;
}

int main(void) {
    struct mystruct *head;

    head = insert_list(NULL, 5);
    head = insert_list(head, 3);
    head = insert_list(head, 6);
    head = insert_list(head, 10);
    head = insert_list(head, 17);

    struct mystruct *scan = head;
    do {
        printf("value=%d\n", scan->value);
        scan = scan->next;
    } while(1);

    return 0;
}
```

Q 1.4
À la lecture du code, qu'est censé faire le programme d'après vous ? Compilez le programme et lancez le. Que se passe-t-il ?

Q 1.5
Recompilez le programme en activant l'option de débogue et lancez l'outil `gdb` sur l'exécutable résultant. Tracez la variable `scan` et exécuter la boucle d'affichage pas-à-pas. Que vaut la variable `scan` au moment où survient l'erreur de segmentation ? Expliquez la cause de l'erreur.

Q 1.6
Proposez une autre condition de boucle (à la place du `while(1)`) pour la boucle d'affichage afin que cette dernière s'arrête correctement.

Partie 3 : *Fuite mémoire*

Récupérez le fichier C nommé `gdb_exo3.c` :

```
#include<stdio.h>
#include<stdlib.h>

typedef struct tableau{
   int* tab;
   int maxTaille;
   int position;
}Tableau;

void ajouterElement(int a,Tableau *t){
   t->tab[t->position]=a;
   t->position++;
}

```

```
15 Tableau* initTableau(int maxTaille){
16     Tableau* t = (Tableau*)malloc(sizeof(Tableau));
17     t->position=0;
18     t->maxTaille=maxTaille;
19     t->tab=(int*)malloc(sizeof(int)*maxTaille);
20     return t;
21 }
22
23 void affichageTableau(Tableau* t){
24     printf("t->position = %d\n",t->position);
25     printf("[ ");
26     for (int i=0;i<(t->position);i++){
27         printf("%d ",t->tab[i]);
28     }
29     printf("]\n");
30 }
31
32 int main(){
33     Tableau* t;
34     t = initTableau(100);
35     ajouterElement(5,t);
36     ajouterElement(18,t);
37     ajouterElement(99999,t);
38     ajouterElement(-452,t);
39     ajouterElement(4587,t);
40     affichageTableau(t);
41     free(t);
42 }
```

Q 1.7
À la lecture du code, qu'est censé faire le programme d'après vous ? Compilez le programme et lancez le. Que se passe-t-il ?

Q 1.8 Quels sont les problèmes éventuels de ce programme ?

Q 1.9 Nous allons utiliser l'outil `valgrind` pour repérer les fuites mémoires éventuelles. Lancer la commande `valgrind --leak-check=yes ./gdb_exo3`. Que constatez-vous ?

Q 1.10 Proposez une correction et vérifiez qu'il n'y a plus de fuite mémoire (en utilisant l'outil `valgrind`).

Exercice 2 – Algorithmique et tableaux

Partie 1 : Tableaux à une dimension

Q 2.1
Créez les fonctions suivantes :

1. une fonction `alloue_tableau` permettant d'allouer la mémoire utilisée par un tableau T d'entiers de taille n. Noter qu'il existe deux versions possibles pour cette fonction :

   `int * alloue_tableau(int n);` et `void alloue_tableau(int **T, int n);`
2. une fonction `remplir_tableau` permettant de remplir le tableau avec des valeurs entières aléatoirement générées entre 0 et V (non-compris).
3. une fonction `afficher_tableau` permettant d'afficher les valeurs du tableau.

Q 2.2
Ecrivez un algorithme permettant de calculer la somme des carrés des différences entre les éléments du tableau pris deux à deux :

$$\sum_{i=1}^{n}\sum_{j=1}^{n}\Big(T(i)-T(j)\Big)^2$$

1. Ecrire un premier algorithme de complexité $O(n^2)$
2. Ecrire un second algorithme de meilleure complexité. Aidez-vous du fait que $\sum_{i=1}^{n}\sum_{j=1}^{n} x_i x_j = \Big(\sum_{i=1}^{n} x_i\Big)^2$

Q 2.3 Comparez les temps de calcul des deux algorithmes. Pour mesurer le temps mis par le CPU pour effectuer une fonction nommée `fct`, on peut utiliser le code suivant où `temps_cpu` contient le temps CPU utilisé en secondes.

```
#include <time.h>
...
clock_t temps_initial; /* Temps initial en micro-secondes */
clock_t temps_final;   /* Temps final en micro-secondes */
double temps_cpu;      /* Temps total en secondes */
...
temps_initial = clock();
  fct();
temps_final = clock ();
temps_cpu =  ((double)(temps_final - temps_initial))/CLOCKS_PER_SEC;
printf("%d %f ",n,temps_cpu);
```

Faites varier la taille n du tableau et analysez les temps de calcul obtenus.

Q 2.4
On veut visualiser par des courbes les séries de nombres obtenues. On peut utiliser pour cela un logiciel extérieur lisant un fichier texte créé par le programme. Par exemple, le logiciel `gnuplot` qui est utilisable en ligne sous linux. Le fichier d'entrée de `gnuplot` est simplement la donnée en lignes de `n` suivi des mesures de temps.
On peut alors lancer `gnuplot` et taper interactivement les commandes. On peut également utiliser une redirection `gnuplopt < commande.txt` avec un fichier du type :

```
plot "01_sortie_vitesse.txt" using 1:2 with lines
replot "02_sortie_vitesse.txt" using 1:3 with lines
set term postscript portrait
set output "01_courbes_vitesse.ps"
set size 0.7, 0.7
replot
```

Ces lignes de commande créent sur le disque le fichier postscript contenant deux courbes sur un même graphique (les lignes précédées par # dans gnuplot sont en commentaires).

Q 2.5 Justifiez les courbes obtenues en fonction de la complexité pire-cas attendue.

Partie 2 : Tableaux à deux dimensions (matrices)

Q 2.6
Créez les fonctions suivantes :

1. une fonction `alloue_matrice` permettant d'allouer la mémoire utilisée par une matrice entière carrée de taille m x m.
2. une fonction `remplir_matrice` permettant de remplir la matrice avec des valeurs entières aléatoirement générées entre 0 et V (non-compris).
3. une fonction `afficher_matrice` permettant d'afficher les valeurs de la matrice.

Q 2.7 Ecrivez un algorithme permettant de vérifier que tous les éléments de la matrice ont des valeurs différentes.

1. Ecrire un premier algorithme de complexité $O(n^2)$ avec n égal au nombre d'éléments de la matrice.
2. Ecrire un second algorithme de meilleure complexité dans le cas où la borne maximale V sur les valeurs contenues dans la matrice est telle qu'un tableau de taille V puisse être alloué en mémoire.

Q 2.8 Comparez les temps de calcul des deux algorithmes et représentez les courbes obtenues en fonction du nombre n d'éléments de la matrice.

Q 2.9 Justifiez les courbes obtenues en fonction de la complexité pire-cas attendue.