TD PG106
valgrind

1 Valgrind : les bases

▸**Exercice 1.** *Accès mémoire : lecture*
Ecrire un programme qui déclare un tableau de dix éléments en allocation automatique et qui affiche les valeurs de ces éléments (sans initialiser le tableau).

Compiler (avec les options `-g -O0`*) et exécuter à l'aide de valgrind :* `valgrind ./a.out`.

▸**Exercice 2.** *Accès mémoire : écriture*
Dans le code précédent, ajouter une boucle pour initialiser les cases du tableaux à 0 et provoquer volontairement un débordement de tableau.

Compiler et exécuter à l'aide de valgrind.

▸**Exercice 3.** *Allocation dynamique*
Reprendre l'exercice précédent mais en allouant le tableau dans le tas. Penser à libérer les ressources.

▸**Exercice 4.** *et gdb ...*
En consultant la page de manuel de valgrind, trouver l'option qui permet de connecter gdb lorsqu'une erreur survient. Tester cette option avec l'exemple précédent.

▸**Exercice 5.** *Fuite mémoire*
Quelle option de valgrind permet d'avoir une analyse mémoire du tas à la fin du programme (détection et caractérisation des fuites mémoires : memory leak *en anglais) ?*

Implémenter les scénarios suivants et tester avec valgrind :

- *Effectué un malloc sans conserver la valeur de retour*
- *Conserver la valeur de retour du malloc dans une variable locale mais ne libérez pas en fin de programme*
- *Conserver la valeur de retour du malloc dans une variable globale mais ne libérez pas en fin de programme*
- *Allouer une matrice de type :* `int **`*, donc un malloc pour le premier tableau puis un malloc pour chaque ligne de la matrice. Libérer uniquement le premier tableau.*
- *Idem, en ne rien libérant du tout.*

2 un peu de debug

Reprendre les exercices du TD *gdb* (`dho.c`, `cafard.c`, `geek.c`, etc) et retrouver les bug en utilisant gdb et valgrind...

3 encore un peu

▸**Exercice 6.** *git toujours...*

Créer une branche `valgrind` *pour l'exercice suivant et placez vous dans cette branche.*

▸**Exercice 7.** *Table de hachage !*
En plus du bug corrigé avec gdb, l'implémentation des tables de hachage comporte des fuites mémoire. Lancer les programmes d'exemple dans valgrind, trouver et corriger ces fuites.

▸**Exercice 8.** *merge :*
Mettre à jour master à partir de la branche créée.

▸**Exercice 9.** *Entrainement : liste chainée*

Télécharger le fichier de liste chainée (page de garde PG106 sur thor) et lancer le dans valgrind. Examiner les sorties de valgrind et identifier les erreurs.

---

4 clang-tidy

clang-tidy est un outil d'analyse statique du code permettant de mettre en avant des erreurs, bugs potentiels etc. Malheureusement, il n'est pas directement disponible sur les machines de l'école, mais vous le trouverez dans `~fmorandat/bin/clang-tidy`.

Ci-dessous un exemple d'utilisation de *clang-tidy* :

```
clang-tidy hash.c -- -I/usr/lib/gcc/x86_64-redhat-linux/4.8.5/include/
```

Le programme détecte deux erreurs potentielles :

```
2 warnings generated.
/net/ens/allali/hash.c:60:10: warning: The left operand of '==' is a garbage value [clang-analyz
    if (c==0)
         ^
/net/ens/allali/hash.c:119:27: note: Calling 'HashAdd'
  printf("add toto %d \n",HashAdd(hash,"toto"));
                          ^
/net/ens/allali/hash.c:47:3: note: 'c' declared without an initial value
  int c;
  ^
/net/ens/allali/hash.c:52:3: note: Taking false branch
  if (hash->entry[value%hash->size]==NULL){
  ^
/net/ens/allali/hash.c:56:12: note: Assuming 'p' is equal to NULL
    while((p!=NULL) && ((c=hash->cmp(donnee,p->donnee))>0)){
           ^
/net/ens/allali/hash.c:56:21: note: Left side of '&&' is false
    while((p!=NULL) && ((c=hash->cmp(donnee,p->donnee))>0)){
                    ^
/net/ens/allali/hash.c:60:10: note: The left operand of '==' is a garbage value
    if (c==0)
         ^
/net/ens/allali/hash.c:61:9: warning: Potential leak of memory pointed to by 'cell' [clang-analy
        return 1; // element is already in the table
               ^
/net/ens/allali/hash.c:119:27: note: Calling 'HashAdd'
  printf("add toto %d \n",HashAdd(hash,"toto"));
                          ^
/net/ens/allali/hash.c:48:21: note: Calling 'hash_newcell'
  struct Cell *cell=hash_newcell(donnee);
                    ^
/net/ens/allali/hash.c:20:18: note: Memory is allocated
  struct Cell *c=malloc(sizeof(*c));
                 ^
```

```
/net/ens/allali/hash.c:48:21: note: Returned allocated memory
  struct Cell *cell=hash_newcell(donnee);
                   ^
/net/ens/allali/hash.c:52:3: note: Taking false branch
  if (hash->entry[value%hash->size]==NULL){
  ^
/net/ens/allali/hash.c:56:12: note: Assuming 'p' is not equal to NULL
    while((p!=NULL) && ((c=hash->cmp(donnee,p->donnee))>0)){
           ^
/net/ens/allali/hash.c:56:11: note: Left side of '&&' is true
    while((p!=NULL) && ((c=hash->cmp(donnee,p->donnee))>0)){
          ^
/net/ens/allali/hash.c:56:25: note: Assuming the condition is false
    while((p!=NULL) && ((c=hash->cmp(donnee,p->donnee))>0)){
                         ^
/net/ens/allali/hash.c:56:5: note: Loop condition is false. Execution continues on line 60
    while((p!=NULL) && ((c=hash->cmp(donnee,p->donnee))>0)){
    ^
/net/ens/allali/hash.c:60:9: note: Assuming 'c' is equal to 0
    if (c==0)
        ^
/net/ens/allali/hash.c:60:5: note: Taking true branch
    if (c==0)
    ^
/net/ens/allali/hash.c:61:9: note: Potential leak of memory pointed to by 'cell'
        return 1; // element is already in the table
               ^
```

Comme vous pouvez le lire, chaque erreur est suivie d'une explication du contexte pouvant mener à l'erreur.