TD PG106
valgrind & clang-tidy

1 Valgrind : les bases

▸**Exercice 1.** *Une nouvelle branche :*

Créer une branche orpheline nommée `exo_valgrind` *(voir le TD précédent pour les commandes) : nous allons faire les exercices suivants dans cette branche.*

▸**Exercice 2.** *Accès mémoire : lecture*
Ecrire un programme de test 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 3.** *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 4.** *Allocation dynamique*
Reprendre l'exercice précédent mais en allouant le tableau dans le tas. Penser à libérer les ressources.

▸**Exercice 5.** *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 6.** *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.*

▸**Exercice 7.** *commit*
Enregistrer dans la branche `exo_valgrind` *votre travail !*

2 un peu de debug

Basculer dans la branche `exo_gdb` et reprendre les exercices `geek.c` et `hack.c` du TD *gdb* et retrouver les bug en utilisant gdb et valgrind...

3 Retour aux sources

▸**Exercice 8.** *git toujours...*
Revenir dans la branche `master` *et créer une nouvelle branche* `valgrind` *(pas orpheline) et placez vous dans cette branche.*

▸**Exercice 9.** *Table de hachage !*
En plus du bug corrigé avec gdb, l'implémentation des tables de hachage comporte de nombreux autres problèmes ! Lancer les programmes d'exemple dans valgrind, trouver et corriger ces fuites.

▸**Exercice 10.** *merge :*
Mettre à jour master à partir de la branche créée.

4 clang-tidy

`clang-tidy` est un outil d'analyse statique du code permettant de mettre en avant des erreurs, bugs potentiels etc.

Ci-dessous un exemple d'utilisation de `clang-tidy` (essayer le sur votre environnement) :

```
clang-tidy prog.c -- -Ihash
```

Les options après les deux - correspondent aux options de compilation (celles que vous avez mises dans la variable `CFLAGS` de votre `Makefile`).

`clang-tidy` est une plateforme d'analyse disposant de nombreux tests, essayer par exemple la commande :

```
clang-tidy prog.c -checks=* -- -Ihash
```

Vous verrez alors appaître beaucoup (trop) de "signalements". l'option `-checks=` permet d'activer ou des désactiver certaines vérifications.

La commande `clang-tidy -checks=* -list-checks` permet de lister tout les tests disponibles (et la commande `clang-tidy -list-checks` ceux qui sont actifs). Il est ainsi possible d'activer ou de désactiver certains tests de la façon suivante :

```
clang-tidy prog.c -checks=clang-analyzer-*,google-* -- -Ihash
ou encore
clang-tidy prog.c -checks=-clang-analyzer-*,google-* -- -Ihash
```

La première commande active les tests `clang-analyzer` et les tests `google` (convention de codage), la deuxième commande désactive les tests `clang-analyzer` et active les tests `google`.

Vous trouverez la liste des *groupes* de tests sur le site internet clang-tidy.

Dans un premier temps, on peut se contenter de n'activer que les tests de `clang-analyzer` (mais ils sont actifs par défaut)et également vouloir vérifier certaines règles concernant le nommage. La commande suivante fait cela :

```
clang-tidy  --checks=readability-identifier* prog.c -- -Ihash
```

Tester les fichiers `list.c` et `hash.c` avec les checks par défaut.

Pour finir, on peut conserver les options d'usage dans un fichier `.clang-tidy` qui permet de paramètrer finement certains tests. On peut générer une base de ce fichier avec la commande :
`.clang-tidy --checks=readability-* --dump-config > .clang-tidy`.

Ci-dessous un exemple :

```
---
Checks:          'clang-diagnostic-*,clang-analyzer-*,readability-*,-readability-magic-numbers'
WarningsAsErrors: ''
```

```
HeaderFilterRegex: ''
AnalyzeTemporaryDtors: false
FormatStyle:     none
User:            allali
CheckOptions:
  - key: readability-identifier-naming.FunctionCase
    value:  'lower_case'
  - key: readability-identifier-length.IgnoredVariableNames
    value:  '[ijp]'
...
```

La ligne `Checks:` permet d'activer ou de désactiver des tests. `CheckOptions:` permet de paramètrer finement certains tests. Par exemple, on peut indiquer la convention de nommage par type d'objet en suivant les indications disponibles sur https://clang.llvm.org/extra/clang-tidy/checks/readability/identifier-naming.html.... On utilise ensuite la commande

`clang-tidy --config-file=.clang-tidy prog.c -- -Ihash`

pour utiliser notre configuration. Lorsqu'un message d'erreur est affiché, le test correspondant est indiqué : libre à nous de modifier les paramètres de ce test (`CheckOptions`) ou bien de désactiver spécifiquement le test (via `Checks:`).

Le paramètrage complet est long, dans un premier temps on peut utiliser `clang-tidy` sans option (cela permet de relever des bugs potentiels), on gardera en mémoire que l'on pourra utiliser `clang-tidy` en complément de `clang-format` pour vérifier plus en détails le bon respect de notre convention de codage.