

TP

Déboguer en C et en Java

L'objectif de ce TP est de se familiariser avec l'utilisation d'outils de débogage.

1 Déroulement du TP

- Le TP est à faire par groupe d'**au plus deux personnes**.
- **Rendu** : Il n'y a pas de rendu pour ce TP : il n'est pas noté. Prenez votre temps pour bien vous familiariser avec les outils.

2 Débogage de programmes C

Gdb permet de déboguer des programmes en C. Nous allons commencer par un tutoriel sur Gdb qui donnera une vue globale de ce qu'on peut généralement faire avec un débogueur.


- Récupérez le programme `gdb-tutorial.c` sur moodle.
- Les instructions du tutoriel sont incluses directement dans le fichier sous forme de commentaires. Il suffit de suivre ces instructions.

3 Débogage de programmes Java sous Eclipse

3.1 Déboguer sous Eclipse

Voici quelques commentaires sur l'utilisation du débogueur sous Eclipse




Exécuter sous débogueur

- Le débogueur est représenté par l'icône .
- Pour exécuter un programme sous débogueur :
 - Clic droit sur le fichier à exécuter → "Debug As" → "Debug Configurations"
 - Spécifiez les arguments de votre programme et lancez le **Debug**
 - Pour les exécutions suivantes, vous pouvez simplement utiliser le bouton **Debug** de la barre d'outils.
- Lancer un programme avec le débogueur fait changer Eclipse de perspective pour passer dans la perspective *débogueur*. Des boutons en haut à droite vous permettent à tout moment de revenir dans la perspective *classique*.
- La perspective *débogueur* comporte plusieurs vues :
 - La vue *Debug* indique le(s) thread(s) en cours et les lignes sur lesquelles ils sont suspendus.
 - La vue *Display* présente le code en cours de débogage
 - La vue *Outline* permet d'explorer les packages/classes
 - La vue *Variables/Breakpoints* est composée de deux onglets :
 - La vue *Variables* permet de visualiser les variables, leurs valeurs et éventuellement leurs données membres.
 - La vue *Breakpoints* affiche les différents points d'arrêts (voir ci-après)




Points d'arrêt

- Pour l'instant le débogage du programme se limite à son exécution. Celle-ci ne s'arrêtera que lorsque le programme sera terminé ou si une exception est levée et n'est pas capturée. Pour observer plus finement l'exécution, nous allons ajouter des **points d'arrêt** (breakpoints).
 - Pour placer un point d'arrêt, il suffit d'effectuer un clic droit sur la marge de gauche, sur la ligne sur laquelle on veut placer le point d'arrêt, et de choisir *Toggle Breakpoint*. Un rond bleu apparaît au niveau de la ligne concernée.
 - Un point d'arrêt peut être défini avant l'exécution ou pendant l'exécution.
 - On peut aussi ajouter des conditions d'activation aux points d'arrêt (clic droit sur le point d'arrêt)
 - Une expression booléenne (*Suspend when true*)
 - Changement de valeur (*Suspend when value changes*)

Contrôle de l'exécution Plusieurs actions peuvent être prises pour suspendre, continuer ou terminer l'exécution d'un programme.

-  (F8) : Continue l'exécution depuis le point où elle a été suspendue jusqu'au prochain arrêt.
-  : Suspend l'exécution du programme là où il en est.
-  (Ctrl+F2) : Termine le programme

De plus, certaines actions permettent de contrôler l'exécution du programme une fois suspendu.

-  (F5) : Exécute la ligne courante. Si la ligne courante est un appel de méthode, le débogueur se place à la première ligne de la méthode. Le débogueur ne peut explorer une méthode que si son code source est accessible.
-  (F6) : Exécute la ligne courante et se place à la ligne qui suit. Si la ligne courante est un appel de méthode, celui-ci est exécuté.
-  (F7) : Retourne de l'appel d'une méthode, cad que toutes les instructions jusqu'à le fin de la méthode en cours d'exploration sont exécutées.

Observer les variables

- Dans la vue *Display*, placez le pointeur de la souris sur une variable permet d'en observer le contenu.
- Le contenu des variables est aussi détaillé dans la vue *Variables*.
 - Notez que par défaut, les variables statiques n'apparaissent pas. Il faut pour cela aller dans le *View Menu* (en haut à droite de la vue) et sélectionner *Show Static Variables*.

3.2 Un premier programme

Récupérez l'archive `debuggingtest.tar.gz` sur Moodle. Extrayez cette archive, et importez le projet dans Eclipse (File → Import → General/Existing Project into Workspace).

Description de l'application L'application à laquelle nous nous intéressons est une version adaptée du problème classique *wordcount* (comptage de mots) utilisé pour évaluer les performances d'un framework *Big Data*. L'application *wordcount* compte le nombre d'occurrence des mots dans un texte.

Notre version adaptée du problème consiste à compter le nombre d'occurrence de taille de mots (en nombre de caractères). Nous voulons donc être capable de dire par exemple, qu'un fichier contient *K* mots de 5 caractères.

A vous de jouer Dans un premier temps, nous nous intéressons au programme `SimpleCode` du package `firstTest`

- Le programme prend un paramètre qui est le chemin vers le fichier à analyser.
- Exécutez le programme en utilisant le fichier `aText.txt` comme fichier d'entrée.
- Vous pouvez observer que `SimpleCode` inclut plusieurs bugs. Votre défi sera de corriger ces bugs avec les contraintes suivantes :
 - Vous devez utiliser le débogueur d'Eclipse. Interdiction d'ajouter des messages de debug.
 - Vous avez le droit d'ajouter/supprimer/modifier seulement deux caractères dans le code.

- Notez que si un message "!!!!!! I IGNORE EXCEPTIONS !!!!!!" apparaît, c'est qu'il reste des erreurs dans votre code. Une exception a été levée (d'où le message affiché) mais nous avons empêché les exceptions d'interrompre le programme pour vous forcer à utiliser des points d'arrêt.

3.3 Un programme avec plusieurs threads

Nous nous intéressons maintenant à une nouvelle version de notre application dans laquelle nous allons utiliser plusieurs threads :

- Un thread est chargé de lire le texte et de compter le nombre de caractères de chaque mots.
- Plusieurs threads comptent le nombre d'apparitions de chaque taille.
- Un buffer permet au thread *lecteur* de transmettre les informations aux threads *compteurs*.

Notre nouveau programme est appelé `ComplexCode` et fait partie du package `multithreadTest`.

- Si vous testez le programme avec le fichier `aText.txt` utilisé précédemment, un premier bug doit apparaître. Identifiez ce bug en utilisant le débogueur d'Eclipse, et corrigez le.
- Si vous testez avec le fichier `aBigText.txt` fourni, vous devriez alors constater un autre problème qui n'apparaît pas forcément en utilisant le fichier `aText.txt`. Identifiez et corrigez ce nouveau problème¹.

Il est clair que les étudiants brillants que vous êtes sont capables de résoudre ces problèmes sans l'aide d'un débogueur. Cependant, l'objectif du TP étant de se familiariser avec l'utilisation d'un débogueur, essayez de jouer le jeu.

4 Débogage mémoire en C

Plusieurs outils permettent de détecter les accès mémoires incorrects en C. L'outil le plus connu dans ce domaine est `valgrind`.

4.1 Valgrind

Valgrind exécute un programme dans une *machine virtuelle* et enregistre toutes les erreurs d'accès mémoire effectuées par ce programme. Parmi les erreurs mises en évidence par `valgrind`, on peut citer notamment (source wikipedia) :

- L'accès à des données non initialisées
- L'accès à des zones mémoires non allouées ou déjà libérées
- La libération d'une zone mémoire déjà libérée
- etc.

Valgrind exécute un programme déjà compilé :

```
$ valgrind ./my_prog
```

Pour simplifier la lecture des erreurs, il est bien d'avoir compilé ce programme avec l'option de debug `-g`.

Vous trouverez sur Moodle deux exemples de code qui incluent des erreurs d'accès mémoire très simples (`mem-error1.c`, `mem-error2.c` et `mem-error3.c`). Si vous tentez d'exécuter ces codes, ils fonctionneront correctement selon toute vraisemblance. Cependant, ces erreurs cachées sont des bombes à retardement (le programme risque de crasher lorsqu'il sera exécuté dans un autre contexte).

Utilisez `valgrind` pour repérer ces erreurs.

En cas d'erreur de type *fuite mémoire*, pensez à utiliser l'option `--leak-check=full`.

4.2 AddressSanitizer

`AddressSanitizer` est un autre outil permettant de détecter les accès mémoires incorrects. Son fonctionnement est différent de `valgrind` : `AddressSanitizer` instrumente le code de l'application et nécessite donc de recompiler celle-ci.

1. Si le problème n'apparaît pas par défaut, essayez de réduire la valeur de la variable `BUFFER.SIZE` dans la classe `ProdCons` pour la fixer, par exemple, à 4.

AddressSanitizer² est intégré à GCC depuis la version 4.8. Il est par exemple utilisé de manière intensive pour contrôler le bon fonctionnement des navigateurs **chromium** et **firefox**.

Pour pouvoir utiliser **AddressSanitizer**, il vous faut recompiler votre code de la manière suivante :

```
$ gcc -g -fsanitize=address mem-error1.c -o mem-error1
```

Il suffit ensuite d'exécuter le programme normalement.

Utilisez les 3 mêmes programmes que précédemment pour observer les messages d'erreur générés par **AddressSanitizer**.

Comparaison entre AddressSanitizer et Valgrind Les 2 outils sont complémentaires. Certaines erreurs peuvent être détectées par **valgrind** et pas par **AddressSanitizer**, et inversement.

AddressSanitizer offre des performances bien supérieures à **valgrind** qui est pénalisé par sa couche de virtualisation. En contrepartie, l'utilisation de **AddressSanitizer** nécessite de recompiler l'application.

Références

La description du débogueur Eclipse s'inspire d'un document rédigé par C. Pain-Barre et H. Garreta (http://infodoc.iut.univ-aix.fr/~cpb/enseignement/java/docs/debuguer_avec_eclipse.pdf)

2. Si le sujet vous intéresse, vous trouverez l'article décrivant **AddressSanitizer** ici : <https://research.google.com/pubs/pub37752.html>