INF203 - Travaux pratiques, séance 5

GDB - Makefile

Outil de mise au point et d'observation de l'exécution d'un programme : gdb

`[TP5/GDB]`
`gdb` est un outil appelé *debogueur*, permetant de suivre et d'analyser l'état d'un programme durant son exécution. L'objectif de cette partie du TP est de vous familiariser avec `gdb`, et d'apprendre les principales commandes. Référez-vous si nécessaire au cours et : **[a]** afin de vous aider à les retenir, notez les commandes que vous utiliserez au fur et à mesure. ■
[b] Avec quelle option de `clang` faut-il compiler le programme pour pouvoir utiliser `gdb` ? ■

Utilisation de gdb pour corriger une erreur de segmentation

Les erreurs de segmentation sont difficiles à diagnostiquer, elles interviennent lors d'un accès invalide à la mémoire mais la position de cet accès invalide n'est pas indiquée par l'erreur elle-même. Les trouver sans utiliser d'outil adapté peut donc être fastidieux : il faut procéder par tatonnements en multipliant les affichages de l'état interne du programme ...
Nous allons voir que `gdb` va nous permettre de localiser une erreur de segmentation de manière simple et rapide.

Le fichier `arguments.c` contient un programme qui affiche à l'écran la valeur (entière) de chacun des arguments qui lui sont donnés sur la ligne de commande ainsi que la somme de ces valeurs (de manière analogue à ce que vous avez fait au début du TP4). Lisez-le, vérifiez que vous le comprenez, compilez-le (sans oublier l'option de compilation) et exécutez-le.
[c] Quel est le message d'erreur affiché à l'écran ? ■

Nous allons maintenant utiliser `gdb` pour trouver d'où vient l'erreur précédente. Exécutez la commande suivante :

```
gdb a.out
```

Vous vous retrouvez dans l'interface de `gdb`, qui est une interface textuelle. L'outil attend alors la saisie de commandes (propres à `gdb`) lui indiquant quoi faire. Commençons par exécuter le programme :

```
run 17 42
```

Comme tout-à-l'heure, le programme s'arrête lors de l'arrivée de l'erreur de segmentation. La différence est qu'à l'arrêt du programme, nous revenons dans l'interface de `gdb` qui nous fournit alors un certain nombre d'informations :

- l'indication qu'une erreur de segmentation a eu lieu ;
- la fonction dans laquelle le programme était en train de s'exécuter ainsi que la valeur de ses arguements ;
- la ligne en cours d'exécution, ayant provoqué l'erreur.

[d] Quelle était la fonction en cours d'exécution au moment de l'erreur ? Quelle était la valeur de son argument ? A votre avis, pourquoi la ligne en cours d'exécution a-t-elle provoqué une erreur de segmentation ? ■
Nous allons maintenant remonter à la source de l'erreur. Pour cela, `gdb` nous permet de revenir en arrière dans les appels de fonctions en cascade. Exécutez la commande :

```
up
```

`gdb` vous indique alors que la fonction en cours d'exécution a été appelée à la ligne 22 du `main` et vous donne également la valeur des arguments transmis à celui-ci. Nous pouvons alors afficher la valeur des variables du programme à ce moment là :

- la valeur fautive :

```
print argv[i]
```

— le compteur de boucle :
```
print i
```
— le détail des arguments transmis au `main` :
```
print argc
print argv[1]
print argv[2]
print argv[3]
```

Enfin, pour voir un extrait du programme autour de la ligne courante, utilisez la commande :
```
list
```
[e] Comprenez l'erreur, expliquez-la et corrigez le code. ■
[f] Quelles sont les versions abrégées des commandes que vous avez utilisées ? ■

Au secours, je veux partir d'ici !
`gdb` dispose d'une aide intégrée dans laquelle sont détaillées toutes les commandes disponibles. Exécutez
```
help
```
puis
```
help all
```
afin de constater le nombre important de commandes possibles. Vous explorez donc un tout petit sous-ensemble d'entre elles ! Pour chercher comment sortir de `gdb`, exécutez par exemple :
```
apropos exit
```
... et quittez `gdb`.

Points d'arrêt et suivi de l'exécution d'une boucle

Le fichier `syracuse.c`, contient un programme qui calcule la suite de Collatz (ou de Syracuse) à partir de la valeur initiale donnée en argument. Lisez le contenu du fichier, compilez-le et exécutez-le. Vous pouvez constater que ce programme affiche toujours 1, qui est la valeur du "dernier" terme de la suite. Nous allons utiliser `gdb` pour suivre l'évolution du programme sans avoir besoin de le modifier (sans ajouter de code pour afficher une trace, par exemple). Exécutez les commandes :
```
gdb a.out
list 25
break 26
```
Si vous avez oublié de compiler le programme avec la bonne option, `gdb` proteste par des messages d'erreur .... Dans ce cas, quittez `gdb`, recompilez et recommencez ...

La commande `break 26` a placé un point d'arrêt ligne 26, *juste avant* son exécution.
Lancez l'exécution du programme (n'oubliez pas de donner un argument). Lorsque le programme s'interrompt (au point d'arrêt), indiquez que vous voulez observer la variable `x` à chaque arrêt.
[g] Quelle est la commande de `gdb` à utiliser (ce n'est pas `print`) ? ■
Reprenez l'exécution du programme :
[h] Quelle est la commande pour reprendre l'exécution après un arrêt ? Indiquez les valeurs successives de `x` à chaque arrêt. ■

Exercice complémentaire :

Se déplacer pas à pas
Il est parfois utile de pouvoir se déplacer non pas de point d'arrêt en point d'arrêt, mais pas à pas. Il y a deux commandes pour cela, `next` et `step`. Retrouvez leur description dans le cours, ou avec l'aide en ligne de gdb (`help next` et `help step`). Expérimentez ces deux façons de progresser dans l'exécution.

Supprimez le point d'arrêt, puis remettez un point d'arrêt *conditionnel* à la ligne 26, lorsque la valeur de `x` vaut 42.
[i] Quelle est la syntaxe pour placer un point d'arrêt conditionnel ? Donnez 3 valeurs initiales de la suite permettant de passer par 42. ■

Compilation séparée, Makefile

[TP5/MAKEFILE]

Compilation séparée.

Compilez séparément les fichiers ***entrees_ sorties_ codage.c***, ***code1.c***, ***code2.c*** et ***codage.c*** :

```
clang -Wall -Werror -c entrees_sorties_codage.c
clang -Wall -Werror -c code1.c
clang -Wall -Werror -c code2.c
clang -Wall -Werror -c codage.c
```

Normalement, les trois premières compilations se passent bien, mais la dernière produit un message d'erreur.
[j] Quelle est la fonction utilisée dans ***codage.c*** alors qu'elle n'est pas déclarée ? ■
Utilisez la commande `grep` pour savoir dans quel fichier cette fonction est déclarée :
[k] Donnez la commande complète commençant par `grep` que vous utilisez. Dans quel fichier la fonction est-elle déclarée ? ■

Dans ***codage.c***, ajoutez une ligne (commençant par `#include`) permettant d'inclure le fichier ***.h*** dans lequel la fonction est déclarée. Vérifiez que la compilation de `codage.c` se passe maintenant normalement.

Créez maintenant le fichier exécutable `codage` par la commande

```
clang -o codage codage.o code1.o code2.o entrees_sorties_codage.o
```

Exécutez ***codage***.
Puis supprimez les fichiers `code1.o`, `code2.o`, `codage.o`, `entrees_sorties_codage.o` et `codage`.
[l] Expliquez pourquoi il n'est pas "dangereux" (risque de perte du travail effectué) de supprimer ces fichiers : les fichiers `.o` et l'exécutable `codage`. En serait-il de même de l'exécutable `installeTP.sh`, par exemple ? ■

Utilisation du Makefile
Observez le contenu du fichier `Makefile`. **Notez bien** que les lignes de *commande* commencent par un caractère de tabulation, et **non pas par des espaces**.
[m] Dessinez, comme vu en TD, le graphe de dépendance de l'exécutable `codage`, et demandez à votre enseignant de vérifier ce graphe avant de continuer. ■

Exécutez la commande

```
make
```

[n] Quels fichiers ont-ils été successivement crées ? ■
Recommencez :

```
make
```

[o] Que signifie le message obtenu ? ■

[p] Que fait la commande `touch`
— avec comme argument un nom (de fichier) inexistant ?
— avec comme argument un nom de fichier existant ?
■

Notez la date de dernière modification de `code1.c`
[q] Quelle commande utilisez-vous pour connaître la date de dernière modification d'un fichier ? ■
Modifiez la date de dernière modification de `code1.c` (avec `touch`), et vérifiez que ce fichier est maintenant le plus récent du répertoire, puis exécutez

```
make
```

[r] Quels fichiers sont-ils successivement recréés ? Vérifiez que cela correspond au chemin du graphe de dépendance entre `code1.c` et `codage`. ■

Refaites la même chose (`touch ...` puis `make`) avec chacun des fichiers `.c` et `.h` apparaissant dans le graphe,

et vérifiez que les fichiers recréés sont conformes au graphe de dépendance.

Exercice complémentaire :

Écrivez votre propre codage :
- sur le modèle de `code1.c` et `code2.c`, créez un fichier `code3.c` avec une nouvelle fonction de codage, de votre choix (conseil : faites simple).
- créez le fichier `code3.h` correspondant.
- modifiez `codage.c` pour pouvoir faire appel à votre codage.
- le cas échéant, modifiez aussi la fonction `lire_choix` du fichier `entrees_sorties_codage.c` afin de pouvoir lire le choix `3`.
- en prenant modèle sur ce qui existe dans ce fichier, complétez le `Makefile` afin de prendre en compte les fichiers `code3.c` et `code3.h`.

Compilez et testez !
[s] Indiquez les modifications que vous avez apporté au `Makefile` ■

La commande de la semaine : `sed`.

La commande ***sed*** permet de transformer un fichier en appliquant différents traitements (suppression, substitution) sur un sous-ensemble de ses chaînes de caractères.

Créez un fichier `noms.txt` contenant (exactement) :

```
Je suis NOM_A, il est NOM_B
Nous sommes un BINOME
qui travaille jusqu'au bout son TP de INF203
```

Essayez les commandes suivante :

```
sed /NOM_A/d noms.txt
sed /NOM/d noms.txt
```

[t] Expliquez l'effet de cette commande. ■

Essayez aussi :

```
sed s/NOM/ploum/ noms.txt
sed s/NOM/ploum/g noms.txt
```

[u] Quelle est la différence entre les deux dernièrs commandes (avec/sans le `g`) ? ■
[v] Comment utilisez-vous `sed` pour afficher le fichier ***Candide_ chapitre1.txt*** du TP1 en remplaçant toutes les occurrences de "Candide" par "Romeo" et toutes les occurrences de "Cunegonde" par "Juliette" ? ■