esiea
ECOLE D'INGENIEURS
DU MONDE NUMERIQUE

INF1032
Méthodes de conception d'algorithmes
Enseignant
Nicolas Bodin

Débogueurs

Vous connaissez l'option `-g` de GCC qui permet d'attacher un débogueur à votre exécutable, mais vous ne savez peut-être pas vraiment comment utiliser ce dernier. Et bien, l'heure est venue... L'heure est venue pour vous de découvrir le fonctionnement de cet outil, de tracer les fuites mémoires jusque dans les moindres recoins, de rendre vos exécutables aussi fiables qu'une fusée Ariane. Bref, l'heure est venue pour vous de prendre votre autonomie.

1 Introduction à GDB

GDB (GNU Debugger) est, comme son nom l'indique, un outil pour déboguer votre code. Sans rentrer dans les détails, en utilisant des options de compilation particulière, GDB est capable de tracer votre programme pas à pas et vous êtes alors capables de l'exécuter ligne par ligne afin de voir ou se cache cette vilaine erreur. Sachez toutefois que GDB est un outil très puissant et que ce tutoriel n'est pas un manuel complet. Nous allons simplement voir ses fonctionnalités principales afin de vous permettre de résoudre un problème plus rapidement.

Il convient dans un premier temps de bien cibler le protocole à suivre en cas d'erreur lors de l'exécution du programme. Tout d'abord rien ne sert de lancer GDB si votre programme ne compile même pas (de toute façon GDB serait incapable de tracer un programme inexistant). Ensuite, il faut cibler ce qui ne fonctionne pas dans le programme.

Mon programme n'affiche rien :

- lui ai-je vraiment demandé d'afficher quelque chose ?
- n'y a-t-il pas une boucle infinie évidente dans mon code ?
- ai-je lancé le bon programme ? Je ne suis peut être pas dans le bon dossier ou je travaille encore avec une ancienne version de mon programme.

Dans ces cas là, GDB ne sera pas d'une grande utilité, sauf peut être pour trouver une boucle infinie. En effet, dans le cas d'un programme qui fonctionne mais qui n'offre pas le résultat escompté, l'utilisation de GDB est relativement délicate mais peut se révéler utile. Si vous avez en revanche un *segfault* ou tout autre crash de votre programme, GDB est **LA** bonne solution.

1.1 Utilisation basique

Il faut dans un premier temps placer des symboles de déboguage dans le code. Rien de bien méchant en réalité puisqu'il suffit d'ajouter l'option `-g` sur votre ligne de compilation. Par exemple :

```
1 gcc prog.c -Wall -o prog.exe -g
```

Notez en revanche que l'utilisation conjointe des options `-g` et `-O3` est incompatible. La deuxième étape consiste à lancer GDB. Pour cela rien de plus simple, il suffit de taper dans la console `gdb` suivi du nom de votre exécutable :

```
1 gdb ./prog.exe
```

Si votre exécutable doit prendre des arguments, vous pouvez les spécifier à l'aide de l'option `-args`. Devez la placer entre `gdb` et le nom de votre exécutable. Voici un exemple.

```
1 gdb --args ./prog.exe arg1 arg2
```

Vous voilà dans GDB ! Votre programme est alors chargé et contrôlé par GDB. Cela signifie que grâce au débogueur, vous pouvez suivre votre programme pas à pas comme vous pourriez le faire à la main. Sauf que GDB possède des options intéressantes pour aller beaucoup plus vite !

Exercice 1.
Tout au long de ce tutoriel GDB, vous allez manipuler le code *GDB.c* disponible sur Moodle. Manipulez GDB uniquement au travers des questions d'un exercice pour ne pas être perdu dans le TP.

1) Dans un premier temps, compilez le code avec l'option `-g`.
2) Lancez GDB sans mettre d'options à votre programme.

1.2 Interruption inattendue de l'exécution

Votre code produit un *segfault* ou une autre interruption inopinée. Il suffit de lancer votre programme et une fois qu'il a crashé, de demander à GDB où exactement dans le code on s'est arrêté. Pour cela, il faut dans un premier temps lancer votre programme avec la commande `run`. Vous pouvez également utiliser le raccourci `r`, mais ce uniquement si vous êtes un vrai programmeur. Votre programme se lance et plante lamentablement, demandez à GDB où votre exécution s'est arrêtée avec la commande `where`. GDB va alors répondre une ligne de code qui vous appartient, ou pas !

Dans le cas simple, cette ligne est une ligne que vous avez écrite, vous savez donc exactement où elle se trouve puisque GDB vous donne son fichier et son numéro de ligne. Il vous reste alors à comprendre pourquoi l'exécution s'est soudainement interrompue. Rappelons que :

- un *stack overflow* est une explosion de la pile certainement dû à un trop grand nombre d'appels récursifs. Dans ce cas, la commande `where` vous indique le nombre d'appels effectués, et cela vous indique clairement la cause du stack averflow. Il faut alors modifier votre code, sûrement une condition d'arrêt de fonction récursive à revoir !
- un *segmentation fault* vous indique un accès en mémoire à un endroit qui ne vous était pas réservé. Vous pouvez le retrouver dans les cas suivants :
  - une modification du contenu d'un pointeur NULL ;
  - un débordement de tableau ;
  - une libération de mémoire d'un pointeur NULL.

Exercice 2.

1) Lancez votre programme avec la commande `run`.

2) Observez le beau plantage et lancer la commande `where`.
3) Sur quelle ligne de quel fichier l'erreur est-elle apparue ?

La commande `where` vous donne la ligne, vous pouvez alors vérifier la valeur des variables se trouvant dans votre programme. Pour cela utilisez la commande `print` (ou son raccourci `p`) suivie du nom de votre variable (par exemple `print i` ou `p i` affiche la valeur de la variable `i` au moment du crash). Vous pouvez également afficher un pointeur (l'adresse est alors affichée en hexa, NULL en cas de valeur nulle) et son contenu (via l'étoile). Utilisez donc tout cela ainsi que vos connaissances pour trouver la cause du *segfault*.

Exercice 3.

Il faut donc déboguer ce segfault.

1) Quelles sont les variables disponibles à l'affichage ?
2) Affichez-les et identifiez l'erreur.
3) Quittez GDB à l'aide de la commande `quit` (ou son raccourci `q`).
4) Supprimez la ligne en erreur du fichier source (elle ne servait pas à grand chose à part provoquer un segfault), recompilez-le et lancez-le à nouveau dans GDB, toujours sans argument.

Dans un cas un peu plus complexe, lorsque votre programme plante, il s'arrête sur une ligne qui ne vous appartient pas. La plupart du temps, il s'agit d'une ligne dans un fichier de librairie standard, lors de l'appel à une de ses fonctions (`free()`, `printf()`, `strlen()` ...). Il vous faut alors remonter d'un ou plusieurs niveaux pour trouver quelle ligne de votre code a fait appel à une fonction de la librairie standard. Pour cela la commande `up` est à votre disposition. Utilisez-là jusqu'à trouver votre ligne de code, et vous pouvez alors afficher vos variables comme précédemment. À l'inverse, si vous êtes remonté trop haut dans votre pile et que vous souhaitez en redescendre, utilisez la commande `down`.

Exercice 4.

1) Utilisez la commande `where` et indiquez combien de fonctions vous devez remonter avant de tomber dans le fichier *GDB.c*.
2) Utilisez la commande `up` pour retomber dans une fonction du fichier *GDB.c*.
3) Une fois l'erreur identifiée, corrigez-là puis quittez GDB.

1.3 Suivi de l'exécution

Dans un cas encore plus complexe, vous allez devoir exécuter votre programme pas à pas, c'est à dire le lancer, puis bloquer son exécution pour le suivre ligne par ligne. Cela peut être utile par exemple dans le cas d'une boucle infinie, pour suivre chaque itération de boucle, vérifier les compteurs etc. Vous allez pour cela utiliser des points d'arrêt (*breakpoints*). Ce sont ces points d'arrêts qui permettent de stopper l'exécution et d'attendre vos ordres. Voici le moyen de s'en servir.

— Si vous devez mettre un point d'arrêt dès le début de votre programme pour suivre ses premières exécutions : entrez la commande `break main`. Cela place un point d'arrêt juste après le main(), et vous permet par exemple de vérifier la validité des arguments de votre programme.
— Si vous devez placer un point d'arrêt sur une ligne précise : vous devez entrer la commande `break` suivi du nom du fichier dans lequel placer votre point d'arrêt, le

symbole `':'` , et enfin le numéro de ligne. Par exemple `break fichier.c:60` va placer un point d'arrêt sur la ligne 60 du fichier `fichier.c`.
— Chaque point d'arrêt possède un numéro d'identification, le premier étant 1, et les suivants étant incrémentés de 1. Vous avez la possibilité de supprimer un point d'arrêt avec la commande `delete` suivi du numéro du point d'arrêt.

Vous pouvez alors lancer l'exécution de votre programme qui va se dérouler jusqu'à votre point d'arrêt. Maintenant vous avez la possibilité d'exécuter votre programme ligne par ligne, toujours en ayant la possibilité d'afficher la valeur de vos variables avec la commande `print`. Vous pouvez également utiliser la commande `display` suivie du nom de la variable qui vous permet d'afficher votre variable tout le temps, pour chaque déplacement dans votre code. L'arrêt de l'affichage se fait avec `undisplay`. Pensez que la ligne de code que vous voyez affichée dans GDB est en attente d'exécution. Pour l'exécuter, vous avez quatre solutions :

— vous souhaitez exécuter la ligne sans rentrer dans les détails, par exemple cette ligne appelle la fonction machin() et vous ne voulez pas suivre le déroulement complet de la fonction machin(), vous voulez simplement l'exécuter. Il faut pour cela appeler la commande `next`. Cela a pour effet d'exécuter l'instruction en cours, et de patienter sur la suivante.
— si au contraire vous voulez descendre dans la fonction machin() pour suivre le fil de l'exécution, il faut utiliser la commande `step`. Dans ce cas, si votre ligne possède un appel de fonction, vous allez entrer dans cette fonction, sinon, l'effet est identique à `next` et exécute simplement la ligne de code.
— si vous êtes dans une fonction et que vous souhaitez exécuter le reste des instructions de cette fonction pour en sortir, il faut alors utiliser la commande `finish`
— si vous souhaiter dérouler l'ensemble du programme jusqu'au prochain point d'arrêt ou la fin du programme, utilisez la commande `continue`.

Exercice 5.

Lancer le programme dans GDB avec les arguments 5 et 3. Sélectionnez l'option 1 puis observez le résultat de l'évolution de cette ligne. Quelque chose ne va pas ...Un appel à un ami vous indique de regarder dans la fonction `update_game()` car c'est elle qui met à jour le plateau de jeu.

1) Placez un *breakpoint* au début de cette fonction.
2) Déroulez l'exécution de code jusqu'à l'appel de la fonction comptant le nombre de cases adjacentes.
3) Placez un nouveau *breakpoint* sur l'appel de la fonction `nb_cases_adj`.
4) Affichez la variable `adj` de façon constante.
5) Utilisez la commande `continue` (ou `c`) ou `next` (`n`) pour dérouler la boucle sur toutes les cases du plateau. Observez au fur et à mesure la valeur de `adj`.
6) Il est donc temps d'observer plus en détails la fonction `nb_cases_adj`. Utilisez les commandes `step` et `next` pour trouver l'erreur.

Vous avez maintenant toutes les bases pour déboguer correctement votre programme. Le principal inconvénient de GDB est le fait de ne pas voir le code en même temps qu'on l'exécute. La plupart du temps, on a envie de voir à l'avance la prochaine ligne qui va être exécutée et GDB étant un débogueur en console, il n'offre pas la double visualisation code / débogueur. Néanmoins vous pouvez utiliser la commande `list` qui vous permet de lister 10 lignes de votre code, entourant la ligne sur laquelle vous êtes en attente.

2 Autres débogueur

Il existe des débogueurs un peu (voire beaucoup) plus graphique que GDB. Vous pouvez par exemple trouver CGDB. Il s'agit du même débogueur, toujours en console, mais la moitié supérieure de votre console vous permet d'afficher votre code. Vous avez également la possibilité d'y switcher pour placer plus facilement vos point d'arrêt. Ceux qui possèdent un PC portable ont un grand intérêt à l'utiliser et à le préférer au classique GDB.

Dans une approche encore un peu plus graphique, vous pouvez utiliser DDD (the Data Display Debugger). Il embarque lui aussi GDB mais cette fois il s'agit d'une application qui s'ouvre dans une nouvelle fenêtre, et pas en console ! Les raccourcis sont simples et clair, et vous avez la possibilité de tout faire à la souris, ce qui vous permet d'éviter de retenir les commandes !

Si vous utilisez une IDE tel que `Code::Blocks` (c'est mal !), sachez qu'un débogueur y est intégré. Cette fois, tout est graphique, très simple à utiliser et très *user friendly* ! En revanche, comme pour tout IDE, cela nécessite la création d'un projet qui peut rapidement être une vraie usine à gaz pour des programmes relativement simples. À proscrire si votre niveau en C est faible car l'utilisation d'un IDE nuit gravement à votre autonomie !

3 Résumé des commandes

Vous retrouverez dans le tableau ci-après la liste des commandes présentées dans ce TP. Sachez également que chaque commande possède un raccourci qu'il est possible d'utiliser en lieu et place de la commande complète. Sachez enfin que si vous faites simplement `ENTREE`, la dernière commande saisie est à nouveau effectuée, ce qui est très pratique pour l'exécution pas à pas ! N'hésitez pas à vous servir fréquemment du débogueur, c'est un outil puissant qui va vite devenir indispensable !

Commande	Raccourci	Description
`quit`	q	Quitter le débogueur
`break <fichier:ligne>` `break <fonction>`	b	Place un point d'arrêt
`run <args>`	r	Lance le programme
`continue`	c	Continue l'exécution
`next`	n	Exécute la ligne sans rentrer dans les fonctions présentes (*step over*)
`step`	s	Exécute la ligne et rentre dans la fonction (*step into*)
`finish`		Termine la fonction courante
`print <var>`	p	Affiche la valeur de la variable `var`
`display <var>`		Affiche la valeur de la variable `var` de façon continue
`undisplay <var>`		Stoppe l'affichage de la variable `var`
`list`		Affiche 10 lignes de code autour de la ligne actuelle.
`where`		Affiche la pile d'appel du programme dans son état courant.
`up`		Remonte le flot d'exécution d'une fonction
`down`		Inverse de up
`delete <no>`		Supprime le point d'entrée numéroté `no`
`kill`		Termine le programme courant