Debuggers

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 ces outils, 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.

Cela commence bien évidemment par l'installation de ces deux outils : GDB et Valgrind. Pour cela, mettez à jour système puis installez les paquets éponymes.

1 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 capable de l'exécuter ligne par ligne afin de voir où 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 :

```
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 :

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

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

```
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, entre autres, 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 lancez 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. Vous pouvez la faire suivre d'un entier indiquant le nombre de fonctions que vous souhaitez remonter. Utilisez-là afin de trouver une ligne de code vous appartenant 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 en affichant les variables disponibles 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'arrêt 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 `potoo()` et vous ne voulez pas suivre le déroulement complet de la fonction `potoo()`, 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 entrer dans la fonction `potoo()` 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.

Lancez 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 les dix lignes de votre code entourant la ligne sur laquelle vous êtes en attente.

1.4 Autres débogueurs

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.

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 !

1.5 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 pressez simplement le touche `ENTREE`, sans avoir saisi de commande, 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	Short	Description
`quit`	`q`	Quitter le débogueur
`break <file:line>` `break <func>`	`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

2 Valgrind

Si vous trouvez que GDB est certes puissant, mais peu pratique pour localiser les segfaults, ou si vous souhaitez vous assurer que votre programme ne plantera jamais, quelque soit la plateforme, Valgrind est **LA** solution. Cet outil sert, entre autres, à tracer toute modification de la mémoire dynamique utilisée par votre programme. Cela passe donc par l'allocation, la modification et la libération de cette dernière. La gestion des fichiers est également prise en compte puisqu'il s'agit en réalité là aussi d'allocation dynamique.

Valgrind comporte cependant un gros point faible : il est incapable de tracer la mémoire allouée de façon statique. Cela signifie donc que même après un passage au peigne fin de votre exécutable, il peut rester des erreurs mémoire qui risquent d'engendrer des segfaults. Cependant, vos programmes comportent une proportion de plus en plus grande d'allocation dynamique, ce qui devrait fiabiliser [1] vos exécutables. Valgrind peut donc être dégainé dans les cas suivants :

- s'assurer que le programme ne comporte aucune fuite mémoire ;
- s'assurer que le programme ne comporte aucune erreur mémoire ;
- déboguer une erreur mémoire.

1. étonnamment

2.1 Utilisation basique

Là encore, l'option `-g` est obligatoire lors de la compilation. Sans elle, vous n'aurez pas d'information sur les lignes de code et variables en erreur. La seconde étape consiste à exécuter votre programme au travers de Valgrind. Tout comme GDB, votre exécutable doit être attaché à Valgrind dès l'appel à ce dernier :

```
valgrind <options> ./prog.exe <arguments>
```

Nous allons découvrir les options principales au cours de ce TP mais pour vous mettre en appétit, en voici quelques unes :

- `-v` active le mode verbose (parle beaucoup) ;
- `--leak-check=full` permet de détecter les fuites mémoires ;
- `--track-origin=yes` permet de tracer les zones mémoire non initialisées.

Attention à bien placer ces options AVANT le nom de votre exécutable sans quoi elles deviendrait des options de votre exécutable et non de Valgrind.

2.2 Le cas parfait

Afin de voir le comportement normal (c'est à dire sans erreur ni fuite mémoire) de Valgrind, vous étudions l'affichage de ce dernier sur un code parfaitement propre.

Exercice 6.

1) Créez un code source allouant dynamiquement puis libérant un tableau de la taille et du type que vous souhaitez. Le tout dans un `main()`, pas de fioritures !
2) Ajoutez à votre code des lignes permettant d'ouvrir un nouveau fichier en mode `w` puis de le fermer.
3) Compilez votre code avec les options classiques et l'option `-g`.
4) Lancez Valgrind sans option sur votre exécutable et observez le résultat !

Sur un code contenant une allocation/libération dynamique et une ouverture/fermeture de fichier, Valgrind nous affiche une série d'informations que nous allons détailler.

```
==1337== Memcheck, a memory error detector
==1337== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1337== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright
 ↪  info
==1337== Command: ./a.out
```

Valgrind nous souhaite ici la bienvenue et nous rappelle, sur sa dernière ligne, la commande qu'il a exécuté. Cette information reste pratique pour vérifier les options que nous avons passées à notre exécutable (ici, aucune). Le nombre (ici, 1337) correspond au PID (IDentifiant de Processus) et changera à chaque exécution de votre programme.

```
==1337== HEAP SUMMARY:
==1337==     in use at exit: 0 bytes in 0 blocks
==1337==   total heap usage: 2 allocs, 2 frees, 562 bytes allocated
==1337==
==1337== All heap blocks were freed -- no leaks are possible
```

Le *Heap Summary* est le résumé de l'utilisation du tas. Il permet de vérifier si l'on a correctement libéré toute la mémoire allouée dynamiquement. S'il manque des `free()` ou des `fclose()`, le nombre de `frees` sera strictement inférieur au nombre d'`allocs`. En cas de fuite mémoire, le nombre de `bytes in use at exit` sera également strictement positif. Ici, tout s'est bien passé, comme nous l'indique la dernière ligne ainsi que le nombre d'`allocs` (ici, 2) identique au nombre de `frees`.

```
==1337== For counts of detected and suppressed errors, rerun with: -v
==1337== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Ces deux dernières lignes nous donnent un résumé des erreurs mémoire détectées. Là encore le décompte est nul, ce qui signifie qu'aucun segfault n'a été détecté. Il s'agit donc ici du message parait, nous indiquant que tout se passe bien et que notre exécutable est (quasiment) sans faille. Étudions maintenant le cas plus intéressant ou quelque chose se passe mal.

2.3 Erreurs mémoire

Nous allons nous concentrer sur quelques erreurs que vous êtes le plus susceptible de réaliser. Pour chacun des types d'erreurs suivants, nous étudierons le message d'erreur affiché par Valgrind, puis nous proposerons une solution :

- erreur d'écriture ou de lecture sur une adresse invalide ;
- accès mémoire en bordure d'une zone précédemment allouée ;
- actions selon une variable non initialisée.

Évidemment, il existe d'autres types d'erreur, mais leur détection sera similaire. Pour tous les exemples suivants, nous supposons que notre code est rédigé dans un fichier *test.c* et compilé sans erreur puis lancé sous la supervision de Valgrind.

En présence d'une erreur mémoire, le résumé des erreurs affiche des résultats positifs. Il suffit donc de lire la dernière ligne du message de Valgrind pour s'apercevoir que le code n'est pas parfait. Afin d'identifier rapidement les erreurs, nous vous conseillons de lancer Valgrind avec l'option `-v` qui affichera beaucoup plus d'informations. Même en cas d'arrêt brutal du programme, Valgrind affichera les erreurs mémoire détectées jusqu'au segfault fatal.

Dans tous les cas, afin de trouver la liste des erreurs, partez du bas du message Valgrind et remontez le message jusqu'à atteindre la première erreur. Corrigez-là, recompilez et exécutez à nouveau votre code afin de vérifier s'il existe encore des erreurs. Répétez ces étapes jusqu'à avoir un code propre.

2.3.a Accès à une adresse invalide

Lorsqu'un pointeur n'est pas alloué correctement, ou lorsqu'une variable n'est pas initialisée, il est possible d'accéder à une zone mémoire invalide. Cela entraîne la plupart du temps un segfault et le programme s'arrête brutalement. Étudions par exemple le code suivant.

```
5  int *p = NULL;
6  *p = 0;
```

Lorsqu'il est compilé, aucun message d'erreur ou d'avertissement ne s'affiche. Une fois exécuté sous la supervision de Valgrind, le programme s'arrête brutalement et Valgrind affiche une série d'informations nous indiquant qu'un segfault est responsable de cet arrêt. La dernière ligne du débogueur nous indique qu'il existe une erreur mémoire. En remontant le message, on tombe sur le détail de cette erreur.

```
==1337== 1 errors in context 1 of 1:
==1337== Invalid write of size 4
==1337==    at 0x80483F9: main (test.c:6)
==1337==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

La première ligne indique le nombre d'erreurs détectées (premier '1') dans le cas particulier détaillé juste en dessous. Ce cas particulier est appelé *context* et cette même première ligne nous indique que ce message détaille le contexte 1/1. Autrement dit, dans ce cas, si on corrige cette erreur, toutes les erreurs du programme seront corrigées puisque c'est la seule.

La seconde ligne (`Invalid write of size 4`) nous indique que l'erreur est une erreur d'écriture sur une zone mémoire de 4 octets. Cette information nous permet d'affirmer que la partie en erreur sera à gauche du symbole égal car il s'agit visiblement d'une erreur d'affectation (un `invalid read` ne nous indique pas de quel côté du symbole égal se situe l'erreur). De plus, 4 octets étant la taille d'un `int`, il y a fort à parier que la variable en question est de ce type.

La ligne suivante détaille dans quelle fonction (ici, `main()`), dans quel fichier (ici, *test.c*) et sur quelle ligne (ici 6) l'erreur a eu lieu. La dernière ligne nous explique pourquoi : on a essayé d'écrire sur l'adresse `0x0` qui n'est pas allouée (de façon statique ou dynamique). La solution consiste donc ici à allouer correctement la zone mémoire (soit en déclarant une variable au préalable, soit en réalisant une allocation dynamique sur le pointeur `p`).

2.3.b Dépassement de tableau

Il s'agit d'une des erreurs les plus fréquentes et les plus vicieuses. Le dépassement de tableau consiste à allouer une zone mémoire de n cases et d'aller lire ou écrire à la case $n+i$, $i \geq 0$. Ce genre d'erreur ne provoque pas systématiquement un arrêt du programme. L'erreur peut être invisible sur votre machine et provoquer un comportement aléatoire ou un arrêt inattendu sur le PC de votre enseignant ou de votre binôme. Étudions l'exemple suivant (oui, il manque les `free()`, mais ce genre d'oubli sera détaillé plus tard).

```
6  int i;
7  short **M = (short **)calloc(7,sizeof(short*));
8  for (i=0; i<7; i++)
9    M[i] = (short*)calloc(3, sizeof(short));
10 for (i=0; i<7;i++)
11   M[i][6] = 37;
```

Encore une fois, sa compilation ne génère aucun avertissement ni aucune erreur. Regardons maintenant la dernière ligne de Valgrind :

```
==1337== ERROR SUMMARY: 14 errors from 1 contexts (suppressed: 0 from 0)
```

Le débogueur nous informe, dans son `ERROR SUMMARY`, qu'il existe 14 erreurs provenant d'un seul contexte. Cela signifie qu'une même ligne de code a provoqué 14 fois le même type d'erreur mémoire. En remontant les messages d'erreur, on tombe sur cette information :

```
==1337== 14 errors in context 1 of 1:
==1337== Invalid write of size 2
==1337==    at 0x1086C4: main (test.c:11)
==1337==  Address 0x522d0cc is 6 bytes after a block of size 6 alloc'd
==1337==    at 0x4C31B25: calloc (in
 ↪  /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==1337==    by 0x108692: main (test.c:9)
```

L'erreur en question est un `Invalid write of size 2` provoqué à la ligne 11. Il s'agit donc visiblement d'une erreur lors de l'affectation d'une valeur à une variable de type `short`. Cherchons maintenant pourquoi cette erreur est survenue. Les trois dernières lignes nous informent où a été allouée la mémoire et quelle taille a été réservée. On voit donc que le programmeur a tenté d'accéder à une zone mémoire 6 octets après la fin d'un tableau contenant 6 cases, alloué à la ligne 9, dans la fonction `main` du fichier *test.c*.

Un coup d'œil sur cette ligne 9 nous permet de voir que chaque ligne de la matrice se compose de 3 cases (donc 6 octets). Un accès à la case 6 (qui correspond donc bien 6 octets trop loin par rapport à la fin de la ligne) est donc impossible. Il faut donc modifier soit la taille du tableau, soit l'indice de la case à affecter.

Exercice 7.
Déboguez le programme *invalidRW.c* disponible sur Moodle.

2.3.c Action selon une variable non initialisée

Valgrind possède quelques talents cachés lors de la surveillance de l'initialisation de variables, qu'elles soient sur la pile ou le tas. Considérons la portion de code suivante.

```
4  int max (int a, int b)
5  {
6    if (a < b)
7      return -1;
8    return 1;
9  }
10 int main()
11 {
12   int test;
13   if (max(test,3) == 1)
14     printf ("a >= 3\n");
15   return 0;
16 }
```

La compilation de ce code génère un avertissement sur la variable `a` qui est utilisée sans être initialisée. Supposons que l'on soit en présence d'un étudiant distrait qui oublie de lire les warnings de GCC. [2] Voyons ce qu'en pense Valgrind.

2. Heureusement, cela n'arrive jamais. Aucun étudiant de l'école ne devrait avoir besoin de lire la suite

```
==1337== Use --track-origins=yes to see where uninitialised values come
 ↪  from
==1337== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==1337==
==1337== 1 errors in context 1 of 1:
==1337== Conditional jump or move depends on uninitialised value(s)
==1337==    at 0x10864A: max (test.c:6)
==1337==    by 0x108670: main (test.c:13)
==1337==
==1337== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Le `HEAP SUMMARY` est parfait [3] mais ce n'est pas le cas du `ERROR SUMMARY`. Valgrind indique qu'une action dépendant d'une valeur non initialisée a eu lieu à la ligne 6, cette fonction ayant été appelée par la ligne 13. La ligne 6 a l'air bien construite, la variable `a` est un paramètre qui a l'air normal... L'erreur vient donc d'ailleurs mais Valgrind ne nous le dit pas, en tous cas pour le moment.

En effet, même si l'on a déjà beaucoup d'informations, Valgrind nous recommande d'utiliser l'option `--track-origins=yes` afin de voir quelle variable est en cause. Si on relance le débogueur avec cette option, on obtient le complément d'information suivant.

```
==1337== Uninitialised value was created by a stack allocation
==1337==    at 0x10861A: main (test.c:11)
```

Et oui ! Dans certains cas, Valgrind est capable de surveiller ce qui se passe sur la pile. Malheureusement, il ne peut pas détecter les dépassements de tableaux alloués sur la pile, mais il gère parfaitement la non-initialisation.

2.4 Fuites mémoire

Une *fuite mémoire* désigne toute allocation mémoire qui n'a pas été désallouée avant la fin de l'exécution du programme. Dans notre cas, il s'agit la plupart du temps d'oubli d'appels aux fonctions `free()` et `fclose()`. Si ce genre d'oubli n'est pas dramatique dans notre contexte (la fermeture du programme entraîne la libération automatique de toute mémoire, statique et dynamique), il peut vite le devenir dans le cas d'un serveur ou d'un programme récursif. Dans ces cas, il est rapidement possible de saturer toute la mémoire RAM à disposition. L'OS se débrouille alors comme il peut et prend la plupart du temps de la mémoire disque, ce qui a pour effet de figer le programme, voir l'ordinateur. Ce dernier devient alors inutilisable et seul un *hard reboot* permet de continuer de travailler.

Revenons maintenant sur l'exemple donné dans la section 2.3.b, et rappelons-le ici. Nous en avons profité pour corriger l'erreur mémoire, ce qui nous permettra de nous concentrer sur les fuites mémoire.

de cette section. Quoique ...

3. Notons que la fonction `printf()` utilise visiblement de l'allocation dynamique.

```
6  int i;
7  short **M = (short **)calloc(7,sizeof(short*));
8  for (i=0; i<7; i++)
9    M[i] = (short*)calloc(3, sizeof(short));
10 for (i=0; i<7;i++)
11   M[i][2] = 37;
```

Ce code ne possédait pas de libération mémoire pour la matrice `M`. Voyons ce que Valgrind a à redire sur ce genre de comportement.

```
==1337== HEAP SUMMARY:
==1337==     in use at exit: 98 bytes in 8 blocks
==1337==   total heap usage: 8 allocs, 0 frees, 98 bytes allocated
==1337==
==1337== Searching for pointers to 8 not-freed blocks
==1337== Checked 70,088 bytes
==1337==
==1337== LEAK SUMMARY:
==1337==    definitely lost: 56 bytes in 1 blocks
==1337==    indirectly lost: 42 bytes in 7 blocks
==1337==      possibly lost: 0 bytes in 0 blocks
==1337==    still reachable: 0 bytes in 0 blocks
==1337==         suppressed: 0 bytes in 0 blocks
==1337== Rerun with --leak-check=full to see details of leaked memory
==1337==
==1337== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Bonne nouvelle : la correction de notre code a permis de supprimer les erreurs mémoire. Mauvaise nouvelle : on se fait sévèrement gronder par Valgrind quand même. Le `HEAP SUMMARY` nous informe que 98 octets ont oubliés d'être libérés. Ces octets ont été alloués au travers de 8 allocations dynamiques. Il manque donc 8 `free()` à notre code.

Le `LEAK SUMMARY` n'est pas très parlant. En revanche, Valgrind propose à nouveau une option : `--leak-check=full`. Lorsqu'on relance avec cette option, on obtient alors la provenance de tous les blocs mémoire non libérés au travers des messages suivants.

```
==1337== 98 (56 direct, 42 indirect) bytes in 1 blocks are definitely
 ↪  lost in loss record 2 of 2
==1337==    at 0x4C31B25: calloc (in
 ↪  /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==1337==    by 0x108661: main (test.c:7)
```

On voit ainsi que les blocs non libérés ont été alloués à la ligne 7, ce qui correspond au premier `calloc`. On voit ici la limitation du débogueur : pour lui, tout dépend de cette seule ligne, il ne voit pas les allocations mémoire présentes sur la ligne 9. Cependant, si on corrige cette erreur en ajoutant au code l'instruction `free(M);` et en relançant Valgrind sur le nouvel exécutable, on voit que le débogueur trouve encore des fuites mémoire.

```
==1337== 42 bytes in 7 blocks are definitely lost in loss record 1 of 1
==1337==    at 0x4C31B25: calloc (in
 ↪  /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==1337==    by 0x1086D2: main (test.c:9)
```

Cette fois la ligne 9 est correctement identifiée. On voit également que les 7 blocs manquants sont ici présents. La correction est alors connue de tous.

Notons donc l'obligation de relancer Valgrind pour toute erreur mémoire détectée, ou pour tout oubli de libération. En effet, il est fréquent de voir disparaître un ensemble d'erreurs à l'aide d'un seul correctif. Inversement, il est aussi possible (mais moins fréquent) de voir apparaître de nouvelles erreurs suite à un correctif qui est pourtant judicieux.

Exercice 8.
Déboguez le programme *fuites_memoire.c* disponible sur Moodle.

3 Conclusion

Vous venez de découvrir au travers de ce TP deux débogueurs particulièrement utiles sur un système Linux. GDB est un débogueur qui vous permet de détecter rapidement pourquoi un programme crashe. Il permet également de suivre pas à pas l'exécution de celui-ci et d'afficher l'ensemble des variables présentes pendant que le programme tourne.

Valgrind vient la plupart du temps en prévision de segfault et de fuites mémoire. Il permet d'identifier l'ensemble des lectures / écritures mémoire sur des zones non allouées et permet également de vérifier que toute mémoire allouée a été correctement libérée.

L'utilisation de ces deux outils est complémentaire. Il est évident que si vous passez systématiquement votre programme à Valgrind avant de le tester, vous avez peu de chances de vous retrouver avec un segfault. Cela peut toutefois arriver avec, par exemple, une écriture sur une zone mémoire statique ou une mauvaise gestion des arguments d'un programme. Cela peut alors être débogué avec GDB.

D'un autre côté, vérifier systématiquement qu'il n'existe aucune erreur mémoire demande beaucoup de temps. Il est donc malheureusement fréquent de devoir exécuter un code qui n'a jamais vu Valgrind auparavant. Si votre code tourne parfaitement, passez-le tout de même à Valgrind, ça peut servir. Si votre code génère un segfault, GDB sera votre meilleure arme pour déboguer rapidement. Dans tous les cas, essayez de prendre le temps d'utiliser Valgrind systématiquement ; et croisez les doigts pour ne jamais avoir besoin de GDB.

Exercice 9. Synthèse.
Rien ne vaut un bon exercice de synthèse ! Votre très cher enseignant a rédigé la correction du TP de gestion de notes que vous aviez l'occasion de rendre avant ce matin, 8h45. Il est bien évidemment tout bogué, à vous de le nettoyer.

Pour cela, récupérer et compilez les fichiers *notes.c* et *notes.h* et lancez votre programme avec en arguments le fichier *Notes.csv*, tous donnés sur Moodle. Déboguez le tout et supprimez toutes les fuites mémoire.