

# TP : Débogage avec *gdb*

Valentin Delis

2025-2026

L'objectif de ce TP est d'apprendre à utiliser un débogueur, qui est un programme qui aide à trouver les sources des problèmes de vos codes. La première section de ce TP est un tutoriel qui vous présentera à travers un code référence les différentes fonctionnalités de *gdb*. Ensuite, on vous propose d'utiliser *gdb* pour déboguer quelques programmes fournis.

## 1 Debogage de programmes avec *gdb* : un exemple complet

Un des aspects complexe du C est la gestion mémoire, et donc par conséquence la capacité à comprendre ses malfonctions associées (mais pas limitées à celles-ci!).

Pour cela, il existe plusieurs approches. La plus basique est de trouver la ligne concernée par le problème avec des affichages de débogage avec la fonction (*f*)*printf*. C'est assez chronophage et parfois inefficace pour comprendre la (les) raison(s) du problème, qui peut être algorithmique, purement mémoire, purement syntaxique, ou un mélange de tout ça.

Dans ce TP, on va s'intéresser à un programme capable de vous aider à traquer les erreurs : le dévermineur *gdb*.

Dans cette première section, on prendra pour exemple le programme `matrice.c`, fourni dans le répertoire *gdb*. Ce programme lit deux matrices entières dont les tailles et les coefficients sont stockés dans le fichier `code/entree.txt`, puis calcule et affiche leur produit.

On exécutera ce programme sur l'exemple suivant (contenu donc dans le fichier `code/entree.txt`).

```
3 2
1 0
0 1
1 1

2 4
2 3 4 5
1 2 3 4
```

Avant de vous lancer, lisez rapidement le fichier `code/matrice.c`, sans chercher pour le moment à trouver des problèmes dans le code.

### 1.1 Lancement de *gdb*

Pour pouvoir lancer l'exécution d'un programme sous *gdb*, il faut "préparer" celui-ci, en l'instrumentant avec l'option spéciale "-g" lors de la compilation avec gcc.

```
$ gcc -Wall -Wextra -g matrice.c -o matrice
```

Puis on lance *gdb* sur ce programme en appelant simplement *gdb* et le nom de l'exécutable :

```
$ gdb ./matrice
GNU gdb (Debian 13.1-3) 13.1
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
```

```

There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/code/bugs/>.
Find the GDB manual and other documentation resources online at:
  <http://www.gnu.org/software/code/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./matrice...
(gdb)

```

Une fois dans *gdb*, on peut exécuter le programme à l'aide de la commande **run** de *gdb*. Ici, notre programme a besoin du contenu du fichier *entree.txt* (avec les appels à *scanf*). Bonne nouvelle, *gdb* permet de passer directement cet argument à la commande **run** :

```

(gdb) run < entree.txt
Program received signal SIGSEGV, Segmentation fault.
0x00005555555552f7 in affiche (M=0x55555555a2b0, nb_lignes=32767, nb_col=4160740048) at matrice.c:38
38         printf("%2d\t",M[i][j]);

```

L'exécution se déroule alors comme si le programme tournait "normalement", avec un lancement classique dans le terminal. Ici, cela se termine malheureusement mal !

Lorsque l'utilisateur de *gdb* retrouve la main dans le programme<sup>1</sup>, il peut choisir de terminer la session en tapant la commande **quit**.

On est souvent amené à exécuter plusieurs fois un programme pour le déboguer. Par défaut, *gdb* réutilisera donc les arguments du précédent appel à **run** si on utilise **run** sans arguments !

À tout moment, la commande **show args** affiche la liste des arguments passés lors du dernier appel de **run** :

```

(gdb) show args
Argument list to give program being debugged when it is started is "< entree.txt\".
(gdb)

```

Si rien ne s'y oppose et que le programme s'exécute normalement, on atteint alors la fin du programme. *gdb* affiche alors à la fin de l'exécution.

## 1.2 Comprendre les sources d'erreur

Ici, le programme s'arrête de façon anormale (erreur de segmentation). Dans ce cas, *gdb* permet d'identifier l'endroit exact où le programme s'est arrêté. Relancez le programme dans *gdb*.

Le débogueur affiche toujours :

```

(gdb) run < entree.txt
Starting program: /quelque/part/matrice < entree.txt

Affichage de A:

Program received signal SIGSEGV, Segmentation fault.
0x00005555555552f7 in affiche (M=0x55555555a2b0, nb_lignes=32767, nb_col=4160740048) at matrice.c:38
38         printf("%2d\t",M[i][j]);
(gdb)

```

On en déduit que l'erreur de segmentation s'est produite à l'exécution de la ligne 38 du programme source, lors d'un appel à la fonction *affiche* avec les arguments  $M = 0x55555555a2b0$ ,  $nb\_lignes = 32767$ ,  $nb\_col = 4160740048$ .

1. Soit le programme s'est bien terminé, avec un message en ce sens, soit il s'est interrompu à cause d'une erreur et elle est affichée. C'est le cas qui plutôt nous intéresse ici !

**Affichage de la pile d'appel** Dans un tel cas, on utilise alors la commande `backtrace` (ou son raccourci `bt`), qui affiche l'état de la pile des appels lors de l'arrêt du programme. Une commande strictement équivalente à `backtrace` est la commande `where`. Lancez les pour tester.

Vous devriez avoir un affichage de ce genre :

```
(gdb) bt
#0  0x00005555555552f7 in affiche (M=0x55555555a2b0, nb_lignes=32767, nb_col=4160740048) at matrice.c:38
#1  0x00005555555554fe in main () at matrice.c:78
(gdb)
```

On apprend ici que l'erreur a été provoquée par la ligne 38 du programme, à l'intérieur d'un appel à la fonction `affiche` qui, elle, avait été appelée à la ligne 78 par la fonction `main`. L'erreur survient donc à l'affichage de la première matrice lue. `gdb` fournit déjà une idée de la source du bogue en constatant que les valeurs des arguments de la fonction `affiche` ont l'air anormales. Vous n'avez donc plus d'excuse en cas de *segmentation fault* pour ne pas trouver la source du problème!!

**Affichage du contenu de certaines variables** Pour en savoir plus, on peut faire afficher les valeurs de certaines variables. On utilise pour cela la commande `print` (en raccourci `p`) qui permet d'afficher la valeur d'une variable, d'une expression etc. Par exemple ici, essayez d'afficher `i` et `j` :

```
(gdb) print i
$1 = 0
(gdb) print j
$2 = 32588
(gdb) print M[i][j]
Cannot access memory at address 0x55555557a000
```

Que peut-on déduire ici de l'affichage ?

**Trouver l'erreur !** L'erreur provient clairement du fait que l'on tente de lire l'élément d'indice `[0][32588]` de la matrice, qui n'est pas défini. Pourquoi ? *Aide : c'est dans le fichier d'entrée ! ...* parce que le fichier `entree.txt` contenait une matrice à 3 lignes et 2 colonnes.

Par défaut, `print` affiche l'objet dans son format : un entier est affiché sous forme décimale, un pointeur sous forme hexadécimale... On peut toutefois préciser le format d'affichage à l'aide d'un spécificateur de format sous la forme

```
(gdb) print /f expression
```

où la lettre `f` précise le format d'affichage. Les principaux formats correspondent aux lettres suivantes : `d` pour la représentation décimale signée, `x` pour l'hexadécimale, `o` pour l'octale, `c` pour un caractère, `f` pour un flottant. Un format d'affichage spécifique au débogueur pour les entiers est `/t` qui affiche la représentation binaire d'un entier.

Essayez de jouer avec ces formats pour afficher les cases de la matrice, leurs adresses etc. Par exemple :

```
(gdb) print j
$3 = 328
(gdb) p /t j
$4 = 111111101001100
(gdb)
```

**Indicateurs d'affichage** Les identificateurs `$1...$4` qui apparaissent en résultat des appels à `print` donnent un nom aux valeurs retournées et peuvent être utilisés par la suite ! Cela évite de retaper des constantes et minimise les risques d'erreur. Par exemple :

```
(gdb) print nb_col
$6 = 4160740048
(gdb) print M[i][$6-1]
Cannot access memory at address 0x555935550e0c
```

L'identificateur \$ correspond à la dernière valeur ajoutée et \$\$ à l'avant-dernière. On peut visualiser les 10 dernières valeurs affichées par print avec la commande `show values`. Testez la.

**Affichages contiguës en mémoire** Une fonctionnalité très utile de `print` est de pouvoir afficher des zones-mémoire contiguës des tableaux dynamiques. Pour une variable `x` donnée, la commande `print x@longueur` affiche la valeur de `x` ainsi que le contenu des longueur-1 zones-mémoires suivantes. Testez avec `M` dans le programme. Par exemple

```
(gdb) print M[0][0]@10
$7 = {1, 0, 0, 0, 0, 0, 0, 33, 0, 0, 1}
```

affiche la valeur de `M[0][0]` et des 9 entiers suivants en mémoire. De même,

```
(gdb) print M[0]@8
$8 = {0x55555555a2d0, 0x55555555a2f0, 0x55555555a310, 0x21, 0x1, 0x0, 0x0, 0x21}
```

affiche la valeur de `M[0]` (de type `int*`) et des 7 objets de type `int*` qui suivent en mémoire.

**Variables avec le même nom** Quand il y a une ambiguïté sur le nom d'une variable (dans le cas où plusieurs variables locales ont le même nom, ou que le programme est divisé en plusieurs fichiers source qui contiennent des variables portant le même nom), on peut préciser le nom de la fonction ou du fichier source dans lequel la variable est définie au moyen de la syntaxe

```
nom_de_fonction::variable
'nom_de_fichier'::variable
```

Pour notre programme, on peut préciser par exemple

```
(gdb) print affiche::nb_col
$9 = 4160740048
```

La commande `whatis` permet, elle, d'afficher le type d'une variable. Elle possède la même syntaxe que `print`. Par exemple,

```
(gdb) whatis M
type = int **
```

**Affichage de tuples et fonctions** Dans le cas de types structures, unions ou énumérations, la commande `ptype` détaille le type en fournissant le nom et le type des différents champs (alors que `whatis` n'affiche que le nom du type).

Enfin, on peut également afficher le prototype d'une fonction du programme à l'aide de la commande `info func` :

```
(gdb) info func affiche
All functions matching regular expression "affiche":

File exemple.c:
void affiche(int **, unsigned int, unsigned int);
(gdb)
```

### 1.3 Appeler des fonctions avec gdb

À l'aide de la commande `print`, on peut également appeler des fonctions du programme en choisissant les arguments. Ainsi pour notre programme, on peut détecter que le bogue vient du fait que la fonction `affiche` a été appelée avec des arguments étranges. En effet, si on appelle `affiche` avec les arguments corrects, on voit qu'elle affiche bien la matrice souhaitée :

```
(gdb) print affiche(M, 3, 2)
1      0
0      1
1      1
$10 = void
```

On remarque que cette commande affiche la valeur retournée par la fonction (ici void). Une commande équivalente est la commande `call` :

```
(gdb) call fonction(arguments)
```

## 1.4 Modifier des variables

On peut aussi modifier les valeurs de certaines variables du programme à un moment donné de l'exécution grâce à la commande

```
(gdb) set variable nom_variable = expression
```

Cette commande affecte à *nom\_variable* la valeur de *expression*. Cette affectation peut également se faire de manière équivalente à l'aide de la commande `print` :

```
(gdb) print nom_variable = expression
```

qui affiche la valeur de *expression* et l'affecte à *variable*.

## 1.5 Se déplacer dans la pile des appels

À un moment donné de l'exécution, *gdb* a uniquement accès aux variables définies dans ce contexte, c'est-à-dire aux variables globales et aux variables locales à la fonction en cours d'exécution. Si l'on souhaite accéder à des variables locales à une des fonctions situées plus haut dans la pile d'appels (par exemple des variables locales à *main*, ou locales à la fonction appelant la fonction courante), il faut au préalable se déplacer dans la pile des appels.

La commande `where` affiche la pile des appels. Par exemple, dans le cas de notre programme, on obtient

```
(gdb) where
#0  0x804865a in affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:38
#1  0x8048881 in main () at exemple.c:78
```

On constate ici que l'on se situe dans la fonction *affiche*, qui a été appelée par *main*. Pour l'instant, on ne peut donc accéder qu'aux variables locales à la fonction *affiche*. Si l'on tente d'afficher une variable locale à *main*, *gdb* produit le message suivant :

```
(gdb) print nb_lignesA
No symbol "nb_lignesA" in current context.
```

La commande `up` permet alors de se déplacer dans la pile des appels. Ici, on a

```
(gdb) up
#1  0x8048881 in main () at exemple.c:78
```

Plus généralement, la commande

```
(gdb) up [nb_positions]
permet de se déplacer de n positions dans la pile. La commande
```

```
(gdb) down [nb_positions]
```

permet de se déplacer de *n* positions dans le sens inverse.

La commande `frame numero` permet de se placer directement au numéro *numero* dans la pile des appels. Si le numéro n'est pas spécifié, elle affiche l'endroit où l'on se trouve dans la pile des appels. Par exemple, si on utilise la commande `up`, on voit grâce à `frame` que l'on se situe maintenant dans le contexte de la fonction `main` :

```
(gdb) up
#1 0x8048881 in main () at exemple.c:78
(gdb) frame
#1 0x8048881 in main () at exemple.c:78
```

On peut alors afficher les valeurs des variables locales définies dans le contexte de `main`. Par exemple

```
(gdb) print nb_lignesA
$9 = 1073928121
(gdb) print nb_colA
$10 = 134513804
```

## 1.6 Poser des points d'arrêt

Un point d'arrêt est un endroit où l'on interrompt temporairement l'exécution du programme afin d'examiner (ou de modifier) les valeurs des variables à cet endroit. La commande permettant de mettre un point d'arrêt<sup>2</sup> est `break` (raccourci en `b`). On peut demander au programme de s'arrêter avant l'exécution d'une fonction (le point d'arrêt est alors défini par le nom de la fonction) ou avant l'exécution d'une ligne donnée du fichier source (le point d'arrêt est alors défini par le numéro de la ligne correspondant). Dans le cas de notre programme, on peut poser par exemple deux points d'arrêt, l'un avant l'exécution de la fonction `affiche`, et l'autre avant la ligne 24 du fichier, qui correspond à l'instruction de retour à la fonction appelante de `lecture_matrice` :

```
(gdb) break affiche
Breakpoint 1 at 0x80485ff: file exemple.c, line 30.
(gdb) break 24
Breakpoint 2 at 0x80485e8: file exemple.c, line 24.
```

En présence de plusieurs fichiers source, on peut spécifier le nom du fichier source dont on donne le numéro de ligne de la manière suivante

```
(gdb) break nom_fichier:numero_ligne
(gdb) break nom_fichier:nom_fonction
```

Quand on exécute le programme en présence de points d'arrêt, le programme s'arrête dès qu'il rencontre le premier point d'arrêt. Dans notre cas, on souhaite comprendre comment les variables `nb_lignesA` et `nb_colA`, qui correspondent au nombre de lignes et au nombre de colonnes de la matrice lue, évoluent au cours de l'exécution. On va donc exécuter le programme depuis le départ à l'aide de la commande `run` et examiner les valeurs de ces variables à chaque point d'arrêt.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/canteaut/COURS_C/DEBUG/exemple < entree.txt
Breakpoint 2, lecture_matrice (nb_lignes=3, nb_col=2) at exemple.c:24
(gdb)
```

2. Sous Emacs, pour mettre un point d'arrêt à la ligne numéro *n* (ce qui signifie que le programme va s'arrêter juste avant d'exécuter cette ligne), il suffit de se placer à la ligne *n* du fichier source et de taper `C-x SPC` où `SPC` désigne la barre d'espace.

Le premier message affiché par *gdb* demande si l'on veut reprendre l'exécution du programme depuis le début. Si l'on répond oui (en tapant *y*), le programme est relancé (avec par défaut les mêmes arguments que lors du dernier appel de *run*). Il s'arrête au premier point d'arrêt rencontré, qui est le point d'arrêt numéro 2 situé à la ligne 24 du fichier. On peut alors faire afficher les valeurs de certaines variables, les modifier... Par exemple, ici,

```
(gdb) print nb_lignes
$11 = 3
(gdb) print nb_col
$12 = 2
```

La commande *continue* (raccourci en *c*) permet de poursuivre l'exécution du programme jusqu'au point d'arrêt suivant (ou jusqu'à la fin). Ici, on obtient

```
(gdb) continue
Continuing.

Affichage de A:

Breakpoint 1, affiche (M=0x8049af8, nb_lignes=1073928121,
    nb_col=134513804) at exemple.c:30
(gdb)
```

On remarque ici que les variables correspondant aux nombres de lignes et de colonnes avaient la bonne valeur à l'intérieur de la fonction *lecture\_matrice*, et qu'elles semblent prendre une valeur aléatoire dès que l'on sort de la fonction. L'erreur vient du fait que les arguments *nb\_lignes* et *nb\_col* de *lecture\_matrice* doivent être passés par adresse et non par valeur, pour que leurs valeurs soient conservées à la sortie de la fonction.

## 1.7 Gérer les points d'arrêt

Pour connaître la liste des points d'arrêt existant à un instant donné, il faut utiliser la commande *info breakpoints* (qui peut s'abréger en *info b* ou même en *i b*).

```
(gdb) info breakpoints
Num Type      Disp Enb Address      What
1  breakpoint keep y   0x080485ff in affiche at exemple.c:30
2  breakpoint keep y   0x080485e8 in lecture_matrice at exemple.c:24
```

On peut enlever un point d'arrêt grâce à la commande *delete* (raccourci *d*) :

```
(gdb) delete numero_point_arrêt
```

En l'absence d'argument, *delete* détruit tous les points d'arrêt.

La commande *clear* permet également de détruire des points d'arrêt mais en spécifiant, non plus le numéro du point d'arrêt, mais la ligne du programme ou le nom de la fonction où ils figurent. Par exemple,

```
(gdb) clear nom_de_fonction
```

enlève tous les points d'arrêt qui existaient à l'intérieur de la fonction. De la même façon, si on donne un numéro de la ligne en argument de *clear*, on détruit tous les points d'arrêt concernant cette ligne.

Enfin, on peut aussi désactiver temporairement un point d'arrêt. La 4e colonne du tableau affiché par *info breakpoints* contient un *y* si le point d'arrêt est activé et un *n* sinon. La commande

```
disable numero_point_arrêt
```

désactive le point d'arrêt correspondant. On peut le réactiver par la suite avec la commande

```
enable numero_point_arrêt
```

Cette fonctionnalité permet d'éviter de détruire un point d'arrêt dont on aura peut-être besoin plus tard, lors d'une autre exécution par exemple.

## 1.8 Les points d'arrêt conditionnels

On peut également mettre un point d'arrêt avant une fonction ou une ligne donnée du programme, mais en demandant que ce point d'arrêt ne soit effectif que sous une certaine condition. La syntaxe est alors

```
(gdb) break ligne_ou_fonction if condition
```

Le programme ne s'arrêtera au point d'arrêt que si la condition est vraie. Dans notre cas, le point d'arrêt de la ligne 24 (juste avant de sortir de la fonction *lecture\_matrice*) n'est vraiment utile que si les valeurs des variables *nb\_lignes* et *nb\_col* qui nous intéressent sont anormales. On peut donc utilement remplacer le point d'arrêt numéro 2 par un point d'arrêt conditionnel :

```
(gdb) break 24 if nb_lignes != 3 || nb_col != 2
Breakpoint 8 at 0x80485e8: file exemple.c, line 24.
(gdb) i b
Num Type           Disp Enb Address      What
1  breakpoint      keep y   0x080485ff in affiche at exemple.c:30
    breakpoint already hit 1 time
3  breakpoint      keep y   0x080485e8 in lecture_matrice at exemple.c:24
    stop only if nb_lignes != 3 || nb_col != 2
(gdb)
```

Si on relance l'exécution du programme avec ces deux points d'arrêt, on voit que le programme s'arrête au point d'arrêt numéro 1, ce qui implique que les variables *nb\_lignes* et *nb\_col* ont bien la bonne valeur à la fin de la fonction *lecture\_matrice* :

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/canteaut/COURS_C/DEBUG/exemple < entree.txt

Affichage de A:

Breakpoint 1, affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
at exemple.c:30
(gdb)
```

On peut aussi transformer un point d'arrêt existant en point d'arrêt conditionnel avec la commande `cond`

```
(gdb) cond numero_point_arret condition
```

Le point d'arrêt numéro *numero\_point\_arret* est devenu un point d'arrêt conditionnel, qui ne sera effectif que si condition est satisfaite.

De même pour transformer un point d'arrêt conditionnel en point d'arrêt non conditionnel (c'est-à-dire pour enlever la condition), il suffit d'utiliser la commande `cond` sans préciser de condition.

## 1.9 Exécuter un programme pas à pas

*gdb* permet, à partir d'un point d'arrêt, d'exécuter le programme instruction par instruction. La commande *next* (raccourci *n*) exécute uniquement l'instruction suivante du programme. Lors que cette instruction comporte un appel de fonction, la fonction est entièrement exécutée.

Par exemple, en partant d'un point d'arrêt situé à la ligne 77 du programme (il s'agit de la ligne

```
printf("\n Affichage de A:\n");
```

dans la fonction `main`), 2 *next* successifs produisent l'effet suivant



```
(gdb) where
#0 main () at exemple.c:77
(gdb) next

Affichage de A:
(gdb) next

Program received signal SIGSEGV, Segmentation fault.
0x804865a in affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:38
(gdb)
```

La premier **next** exécute la ligne 77; le second exécute la ligne 78 qui est l'appel à la fonction affiche. Ce second **next** conduit à une erreur de segmentation.

La commande **step** (raccourci **s**) a la même action que **next**, mais elle rentre dans les fonctions : si une instruction contient un appel de fonction, la commande **step** effectue la première instruction du corps de cette fonction. Si dans l'exemple précédent, on exécute deux fois la commande **step** à partir de la ligne 78, on obtient

```
(gdb) where
#0 main () at exemple.c:78
(gdb) step
affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804) at exemple.c:30
(gdb) step
(gdb) where
#0 affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:35
#1 0x8048881 in main () at exemple.c:78
(gdb)
```

On se trouve alors à la deuxième instruction de la fonction affiche, à la ligne 35.

Enfin, lorsque le programme est arrêté à l'intérieur d'une fonction, la commande **finish** termine l'exécution de la fonction. Le programme s'arrête alors juste après le retour à la fonction appelante. Par exemple, si l'on a mis un point d'arrêt à la ligne 14 (première instruction *scanf* de la fonction *lecture\_matrice*), la commande **finish** à cet endroit fait sortir de *lecture\_matrice* :

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/canteaut/COURS_C/DEBUG/exemple < entree.txt

Breakpoint 2, lecture_matrice (nb_lignes=3, nb_col=134513804) at exemple.c:14
(gdb) where
#0 lecture_matrice (nb_lignes=3, nb_col=134513804) at exemple.c:14
#1 0x804885b in main () at exemple.c:76
(gdb) finish
Run till exit from #0 lecture_matrice (nb_lignes=3, nb_col=134513804)
    at exemple.c:14
0x804885b in main () at exemple.c:76
Value returned is $1 = (int **) 0x8049af8
(gdb)
```

## 1.10 Afficher la valeur d'une expression à chaque point d'arrêt

On a souvent besoin de suivre l'évolution d'une variable ou d'une expression au cours du programme. Plutôt que de répéter la commande **print** à chaque point d'arrêt ou après chaque **next** ou **step**, on peut utiliser la commande **display** (même syntaxe que **print**) qui permet d'afficher la valeur d'une expression à chaque fois que le programme s'arrête. Par exemple, si l'on veut faire afficher par **gdb** la valeur de `M[i][j]` à chaque exécution de la ligne

```
printf ("%2d\t", M[i][j]);
```

dans les deux boucles for de affiche), on y met un point d'arrêt et on fait

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/canteaut/COURS_C/DEBUG/exemple < entree.txt

Affichage de A:

Breakpoint 1, affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
  at exemple.c:38
(gdb) display i
1: i = 0
(gdb) display j
2: j = 0
(gdb) display M[i][j]
3: M[i][j] = 1
(gdb) continue
Continuing.

Breakpoint 1, affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
  at exemple.c:38
3: M[i][j] = 0
2: j = 1
1: i = 0
(gdb) c
Continuing.

Breakpoint 1, affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
  at exemple.c:38
3: M[i][j] = 0
2: j = 2
1: i = 0
(gdb) next
3: M[i][j] = 0
2: j = 2
1: i = 0
(gdb)
```

On remarque que la commande **display** affiche les valeurs des variables à chaque endroit où le programme s'arrête (que cet arrêt soit provoqué par un point d'arrêt ou par une exécution pas-à-pas avec **next** ou **step**). A chaque expression faisant l'objet d'un **display** est associée un numéro. La commande **info display** (raccourci **i display**) affiche la liste des expressions faisant l'objet d'un **display** et les numéros correspondants.

```
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
3:   y  M[i][j]
2:   y  j
1:   y  i
(gdb)
```

Pour annuler une commande **display**, on utilise la commande **undisplay** suivie du numéro correspondant (en l'absence de numéro, tous les **display** sont supprimés) :

```
(gdb) undisplay 1
(gdb) info display
Auto-display expressions now in effect:
Num Enb Expression
3:   y  M[i][j]
2:   y  j
(gdb)
```

```
(gdb)
```

Comme pour les points d'arrêt, les commandes

```
(gdb) disable disp numero_display
(gdb) enable disp numero_display
```

respectivement désactive et active l'affichage du **display** correspondant.

## 1.11 Exécuter automatiquement des commandes aux points d'arrêt

On peut parfois souhaiter exécuter la même liste de commandes à chaque fois que l'on rencontre un point d'arrêt donné. Pour cela, il suffit de définir une seule fois cette liste de commandes à l'aide de **commands** avec la syntaxe suivante :

```
(gdb) commands numero_point_arret
commande_1
...
commande_n
end
```

où *numero\_point\_arret* désigne le numéro du point d'arrêt concerné. Cette fonctionnalité est notamment utile car elle permet de placer la commande continue à la fin de la liste. On peut donc automatiquement passer de ce point d'arrêt au suivant sans avoir à entrer continue. Supposons par exemple que le programme ait un point d'arrêt à la ligne 22 (ligne

```
scanf ("%d", &M[i][j]);
```

de la fonction *lecture\_matrice*. A chaque fois que l'on rencontre ce point d'arrêt, on désire afficher les valeurs de *i*, *j*, *M[i][j]* et reprendre l'exécution. On entre alors la liste de commandes suivantes associée au point d'arrêt 1 :

```
(gdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>echo valeur de i \n
>print i
>echo valeur de j \n
>print j
>echo valeur du coefficient M[i][j] \n
>print M[i][j]
>continue
>end
(gdb)
```

Quand on lance le programme, ces commandes sont effectuées à chaque passage au point d'arrêt (et notamment la commande **continue** qui permet de passer automatiquement au point d'arrêt suivant). On obtient donc

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /quelque/part/exemple < entree.txt
Breakpoint 1, lecture_matrice (nb_lignes=3, nb_col=2) at exemple.c:22
valeur de i
$38 = 0
valeur de j
$39 = 0
valeur du coefficient M[i][j]
$40 = 0
```

```

Breakpoint 1, lecture_matrice (nb_lignes=3, nb_col=2) at exemple.c:22
valeur de i
$41 = 0
valeur de j
$42 = 1
valeur du coefficient M[i][j]
$43 = 0

Breakpoint 1, lecture_matrice (nb_lignes=3, nb_col=2) at exemple.c:22
valeur de i
$44 = 1
valeur de j
$45 = 0
valeur du coefficient M[i][j]
$46 = 0

...

Breakpoint 1, lecture_matrice (nb_lignes=3, nb_col=2) at exemple.c:22
valeur de i
$53 = 2
valeur de j
$54 = 1
valeur du coefficient M[i][j]
$55 = 0

Affichage de A:

Program received signal SIGSEGV, Segmentation fault.
0x804865a in affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:38
(gdb)

```

Il est souvent utile d'ajouter la commande `silent` à la liste de commandes. Elle supprime l'affichage du message *Breakpoint ...* fourni par *gdb* quand il atteint un point d'arrêt. Par exemple, la liste de commande suivante

```

(gdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just "end".
>silent
>echo valeur de i \n
>print i
>echo valeur de j \n
>print j
>echo valeur du coefficient M[i][j] \n
>print M[i][j]
>continue
>end

```

produit l'effet suivant à l'exécution :

```

(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/canteaut/COURS_C/DEBUG/exemple < entree.txt
valeur de i
$56 = 0
valeur de j
$57 = 0
valeur du coefficient M[i][j]
$58 = 0

```

```

valeur de i
$59 = 0
valeur de j
$60 = 1
valeur du coefficient M[i][j]
$61 = 0
...
valeur de i
$71 = 2
valeur de j
$72 = 1
valeur du coefficient M[i][j]
$73 = 0

Affichage de A:

Program received signal SIGSEGV, Segmentation fault.
0x804865a in affiche (M=0x8049af8, nb_lignes=1073928121, nb_col=134513804)
    at exemple.c:38
(gdb)

```

Notons enfin que la liste de commandes associée à un point d'arrêt apparaît lorsque l'on affiche la liste des points d'arrêt avec [info breakpoints](#).

## 1.12 Correction de `matrice.c`

Maintenant que nous avons vu un large spectre d'actions avec *gdb*, à vous de jouer pour corriger le programme `matrice.c`!

## 2 Débogage : à vous de jouer !

### 2.1 Un premier programme avec un peu d'aide

Le programme<sup>3</sup> contenu dans le fichier source `buginsa.c` prend en argument un entier  $n$ , crée un tableau de taille  $n$ , l'initialise puis désalloue le tableau. Mais il donne une `segmentation fault`. Pourquoi?

1. Lancez `gdb` et chargez le binaire.
2. Lancez l'exécution avec la valeur 3 comme argument, repérer la ligne où a lieu l'erreur.
3. Afficher la pile d'appels de fonction menant à l'erreur
4. Tapez `help breakpoints` pour avoir la liste des commandes permettant d'utiliser les points d'arrêt.
5. Mettez un point d'arrêt au début de la fonction de traitement.
6. Relancez le programme.
7. Lorsque vous êtes arrêté dans la fonction de traitement, mettez une surveillance de la variable  $i$ .
8. Tapez `cont` pour continuer, surveillez l'exécution jusqu'à ce que vous trouviez le problème.

### 2.2 A vous de jouer !

Corrigez les programmes suivants, en vous aidant au maximum de `gdb`.

1. Ouvrir le fichier `exo-bug.c`. Le programme recherche un élément dans un tableau trié de taille  $nb$  par dichotomie. Corriger le problème de boucle infini en vous intéressant aux valeurs des variables `min`, `max` et `mid` dans la fonction.
2. Trouver et corriger le bogue dans `pointer.c`<sup>4</sup>. A la fin, on veut avoir la valeur "PRIM" dans la variable  $s$  du `main`. Quel est le problème? Dessiner l'état mémoire du programme et corriger la fonction (et en conséquence, le `main`) pour que le programme fasse ce que l'on veut.
3. Trouver et corriger le bogue dans `list.c`. Vous pouvez lancer le programme avec

```
$ gcc -Wall -Wextra -g list.c -o list ; ./list < entree_list.in
```

4. Trouver et corriger le bogue dans `listsegef.c`.
5. Trouver et corriger le bogue dans `boucle.c`.
6. Trouver et corriger le bogue dans `perror.c`.
7. Trouver et corriger le bogue dans `prime.c`.
8. Trouver et corriger les bogues dans `list_ftrahay.c` :.
9. Trouver et corriger le bogue dans `fib.c`.
10. Trouver et corriger le bogue dans `array.c`.
11. Trouver et corriger le bogue dans `tp_gdb.c`.
12. Trouver et corriger le bogue dans `stack.c`.  
Indication : vous comprendrez peut-être pourquoi l'utilisation de `i++` est restreinte dans le projet.
13. Trouver le bogue dans `listcyc.c`.

Si `ddd` est présent sur votre machine, vous pouvez l'utiliser à la place de `gdb`, ce qui vous permettra d'avoir une représentation graphique de la mémoire.

### 2.3 Recherche de fuites mémoires

Un programme peut sembler correct, mais des fuites mémoires peuvent empêcher son utilisation à grande échelle si elles remplissent la mémoire. Pour chercher la présence de fuites, on peut utiliser l'outil `valgrind`. Pour cela, on lance

---

3. Source : Tanguy Risset, INSA Lyon

4. Ce genre de bogue est difficile à comprendre sans déverminage, car c'est un pure problème de comportement mémoire en C.

```

$ valgrind ./mon_exec
==48766== Memcheck, a memory error detector
==48766== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==48766== Using Valgrind-3.19.0 and LibVEX; rerun with -h for copyright info
==48766== Command: ./mon_exec
==48766==
==48766== HEAP SUMMARY:
==48766==    in use at exit: 640 bytes in 40 blocks
==48766== total heap usage: 52 allocs, 12 frees, 1,840 bytes allocated
==48766==
==48766== LEAK SUMMARY:
==48766==    definitely lost: 640 bytes in 40 blocks
==48766==    indirectly lost: 0 bytes in 0 blocks
==48766==    possibly lost: 0 bytes in 0 blocks
==48766==    still reachable: 0 bytes in 0 blocks
==48766==    suppressed: 0 bytes in 0 blocks
==48766== Rerun with --leak-check=full to see details of leaked memory
==48766==
==48766== For lists of detected and suppressed errors, rerun with: -s
==48766== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

Ici, on remarque qu'il y a une fuite mémoire : 40 blocs, représentant 640 octets de données, n'ont pas été libérés. On peut obtenir plus d'informations avec l'option `-leak-check=full` (à mettre avant le nom de l'exécutable). Si on a compilé avec les informations de débogage (`-g`), on aura accès aux endroits où ont été alloués les blocs non libérés, par exemple :

```

==49138== 16 bytes in 1 blocks are definitely lost in loss record 4 of 40
==49138==    at 0x48417B4: malloc (vg_replace_malloc.c:381)
==49138==    by 0x109287: split (listleak.c:54)
==49138==    by 0x10946C: merge_sort (listleak.c:100)
==49138==    by 0x109478: merge_sort (listleak.c:101)
==49138==    by 0x109583: main (listleak.c:120)

```

1. Reprendre la version boguée de `fib.c` et constater avec *valgrind* que l'erreur est plus explicite.
2. Rechercher et corriger les fuites mémoires dans `sort.c`.
3. Rechercher et corriger les fuites mémoires dans `listleak.c`.
4. Rechercher et corriger les fuites mémoires dans `eratosthene.c`.
5. Reprendre les fichiers de la section précédente et vérifier qu'ils n'ont pas de fuite mémoire ; les corriger sinon.

## Références

Ce TP reprend en première partie le contenu du tutoriel très complet d'Anne Canteaut. Certains codes de la Section 2.2 proviennent de Walter Rudametkin, d'autres sont totalement créés pour l'occasion !

Si vous avez des codes à proposer<sup>5</sup> pour ce TP, n'hésitez pas !

---

5. Vous serez évidemment crédité !