Le débogueur GDB

Anne Canteaut

INRIA Projet CODES

`Anne.Canteaut@inria.fr`

Le logiciel `gdb` est un logiciel GNU permettant de déboguer les programmes C (et C++). Il permet de répondre aux questions suivantes :

- à quel endroit s'arrête le programme en cas de terminaison incorrecte, notamment en cas d'erreur de segmentation ?
- quelles sont les valeurs des variables du programme à un moment donné de l'exécution ?
- quelle est la valeur d'une expression donnée à un moment précis de l'exécution ?

`Gdb` permet donc de lancer le programme, d'arrêter l'exécution à un endroit précis, d'examiner et de modifier les variables au cours de l'exécution et aussi d'exécuter le programme pas-à-pas.

1 Démarrer gdb

Pour pouvoir utiliser le débogueur, il faut avoir compilé le programme avec l'option `-g` de `gcc`. Cette option génère des informations symboliques nécessaires au débogueur. Par exemple:

```
gcc -g -Wall -ansi -o exemple exemple.c
```

On peut ensuite lancer `gdb` sous le shell par la commande

`gdb` *nom de l'exécutable*

Toutefois, il est encore plus pratique d'utiliser `gdb` avec l'interface offerte par `Emacs`. Pour lancer `gdb` sous `Emacs`, il faut utiliser la commande

```
M-x gdb
```

où `M-x` signifie qu'il faut appuyer simultanément sur la touche `Méta` (`Alt` sur la plupart des claviers) et sur `x`. `Emacs` demande alors le nom du fichier exécutable à déboguer : il affiche dans le mini-buffer

```
Run gdb (like this): gdb
```

Quand on entre le nom d'exécutable, `gdb` se lance : le lancement fournit plusieurs informations sur la version utilisée et la licence GNU. Puis, le prompt de gdb s'affiche :

```
(gdb)
```

On peut alors commencer à déboguer le programme.

On est souvent amené au cours du débogage à corriger une erreur dans le fichier source et à recompiler. Pour pouvoir travailler avec le nouvel exécutable sans avoir à quitter `gdb`, il faut le redéfinir à l'aide de la commande `file` :

`(gdb) file` *nom_executable*

2 Quitter gdb

Une fois le débogage terminé, on quitte gdb par la commande

```
(gdb) quit
```

Parfois, gdb demande une confirmation :

```
The program is running.  Exit anyway? (y or n)
```

Il faut évidemment taper y pour quitter le débogueur.

3 Exécuter un programme sous gdb

Pour exécuter un programme sous gdb, on utilise la commande run :

(gdb) run [*arguments du programme*]

où *arguments du programme* sont, s'il y en a, les arguments de votre programme. On peut également utiliser comme arguments les opérateurs de redirection, par exemple :

```
(gdb) run 3 5 > sortie
```

gdb lance alors le programme exactement comme s'il avait été lancé avec les mêmes arguments :

```
./a.out 3 5 > sortie
```

Comme la plupart des commandes de base de gdb, run peut être remplacé par la première lettre du nom de la commande, r. On peut donc écrire également

```
(gdb) r 3 5 > sortie
```

On est souvent amené à exécuter plusieurs fois un programme pour le déboguer. Par défaut, gdb réutilise donc les arguments du précédent appel de 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 "3
5 > sortie".
(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

```
Program exited normally.
(gdb)
```

```c
#include <stdio.h>
#include <stdlib.h>

/* declaration des fonctions secondaires */
int **lecture_matrice(unsigned int, unsigned int);
void affiche(int **, unsigned int, unsigned int);
int **produit(int **, int **, unsigned int, unsigned int, unsigned
 int, unsigned int);

int **lecture_matrice(unsigned int nb_lignes, unsigned int nb_col)
{
  int **M;
  int i, j;

  scanf("%d", &nb_lignes);
  scanf("%d", &nb_col);
  M = (int**)malloc(nb_lignes * sizeof(int*));
  for (i=0; i<nb_lignes; i++)
    M[i] = (int*)malloc(nb_col * sizeof(int));
  for (i=0; i<nb_lignes; i++)
    {
      for (j=0; j<nb_col; j++)
        scanf("%d",&M[i][j]);
    }
  return(M);
}

void affiche(int **M, unsigned int nb_lignes, unsigned int nb_col)
{
  int i, j;
  if (M == NULL)
    {
      printf("\n Erreur: la matrice a afficher est egale a NULL\n");
      return;
    }
  for (i=0; i < nb_lignes; i++)
    {
      for (j=0; j < nb_col; j++)
        printf("%2d\t",M[i][j]);
      printf("\n");
    }
  return;
}


int **produit(int **A, int **B, unsigned int nb_lignes1, unsigned
int nb_col1, unsigned int nb_lignes2, unsigned int nb_col2)
{
  int **P;
  int i, j, k;

  if (nb_col1 != nb_lignes2)
    {
      printf("\n Impossible d’effectuer le produit : dimensions incompatibles\n");
      return(NULL);
    }
  P = (int**)malloc(nb_lignes1 * sizeof(int*));
  for (i=0; i<nb_lignes1; i++)
    P[i] = (int*)calloc(nb_col2, sizeof(int));
  /* calcul de P[i][j] */
  for (i=0; i < nb_lignes1; i++)
    {
      for (j=0; j < nb_col2; j++)
        {
          for (k=0; k < nb_lignes2; k++)
            P[i][j] += A[i][k] * B[k][j];
        }
    }
  return(P);
}


int main()
{
  int **A, **B, **P;
  unsigned int nb_lignesA, nb_lignesB, nb_colA, nb_colB;

  A = lecture_matrice(nb_lignesA, nb_colA);
  printf("\n Affichage de A:\n");
  affiche(A, nb_lignesA, nb_colA);
  B = lecture_matrice(nb_lignesB, nb_colB);
  printf("\n Affichage de B:\n");
  affiche(B, nb_lignesB, nb_colB);
  P = produit(A, B, nb_lignesA, nb_colA, nb_lignesB, nb_colB);
  printf("\n Affichage du produit de A par B:\n");
  affiche(P, nb_lignesA, nb_colB);
  return(EXIT_SUCCESS);
}
```

4 Terminaison anormale du programme

Dans toute la suite, on prendra pour exemple le programme de la page 3, dont le but est de lire deux matrices entières dont les tailles et les coefficients sont fournis dans un fichier `entree.txt`, puis de calculer et d'afficher leur produit.

On exécutera ce programme sur l'exemple suivant (contenu du fichier `entree.txt`)

```
3 2
1 0
0 1
1 1

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

Pour déboguer, on exécute donc la commande

```
(gdb) run < entree.txt
```

Ici le programme s'arrête de façon anormale (erreur de segmentation). Dans ce cas, `gdb` permet d'indentifier l'endroit exact où le programme s'est arrêté. Il affiche par exemple

```
(gdb) run < entree.txt
Starting program: /home/canteaut/COURS_C/DEBUG/exemple < entree.txt

 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)
```

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 = 0x8049af8, nb_lignes = 1073928121, nb_col = 134513804`. Par ailleurs, la fenêtre `Emacs` utilisée pour déboguer se coupe en 2, et affiche dans sa moitié inférieure le programme source, en pointant par une flèche la ligne qui a provoqué l'arrêt du programme :

```
      for (j=0; j < nb_col; j++)
=>        printf("%2d\t",M[i][j]);
      printf("\n");
```

Dans un tel cas, on utilise alors la commande `backtrace` (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`.

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

On apprend ici que l'erreur a été provoqué 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.

5 Afficher les données

Pour en savoir plus, on peut faire afficher les valeurs de certaines variables. On utilise pour cela la commande `print` (raccourci `p`) qui permet d'afficher la valeur d'une variable, d'une expression... Par exemple ici, on peut faire

```
(gdb) print i
$1 = 0
(gdb) print j
$2 = 318
(gdb) print M[i][j]
Cannot access memory at address 0x804a000.
```

L'erreur provient clairement du fait que l'on tente de lire l'élément d'indice [0][318] de la matrice qui n'est pas défini (puisque le fichier `entree.txt` contenait une matrice à 3 lignes et 2 colonnes).

Par défaut, `print` affiche l'objet dans un format "naturel" (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.

```
(gdb) print j
$3 = 328
(gdb) p /t j
$4 = 101001000
```

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
$5 = 134513804
(gdb) print M[i][$5-1]
Cannot access memory at address 0x804a000.
```

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`.

Une fonctionnalité très utile de `print` est de pouvoir afficher des zones-mémoire contiguës (on parle de 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. Par exemple

```
(gdb) print M[0][0]@10
$4 = {1, 0, 0, 17, 0, 1, 0, 17, 1, 1}
```

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

```
(gdb) print M[0]@8
$5 = {0x8049b08, 0x8049b18, 0x8049b28, 0x11, 0x1, 0x0, 0x0, 0x11}
```

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

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
$6 = 134513804
```

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 **
```

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 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)
```

6 Appeler des fonctions

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
$8 = 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)*

7 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*.

8 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 nume'ro 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
```

9 Poser des points d'arrêt

Un point d'arrêt est un endroit où l'on interrompt temporairement l'exécution du programme enfin d'examiner (ou de modifier) les valeurs des variables à cet endroit. La commande permettant de mettre un point d'arrêt 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
```

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.

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)
```

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.

10 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'abbréger 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.

11 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.

12 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)
```

13 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 38 (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)
```

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.

14 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: /home/canteaut/COURS_C/DEBUG/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`.

15 Les raccourcis des noms de commande

Tous les noms de commande peuvent être remplacés par leur plus court préfixe non-ambiguë. Par exemple, la commande `clear` peut s'écrire `cl` car aucune autre commande de `gdb` ne commence par `cl`. La lettre `c` seule ne peut pas être utilisée pour `clear` car plusieurs commandes de `gdb` commencent par `c` (`continue`, `call`...).

`Emacs` permet la complétion automatique des noms de commande `gdb` : quand on tape le début d'une commande et que l'on appuie sur `TAB`, le nom de la commande est complété s'il n'y a pas d'ambiguïté. Sinon, `Emacs` fournit toutes les possibilités de complétion. Il suffit alors de cliquer (bouton du milieu) pour choisir la commande.

Les commandes les plus fréquentes de `gdb` sont abbrégées par leur première lettre, même s'il existe d'autres commandes commençant par cette lettre. Par exemple, la commande `continue` peut être abbrégée en `c`. C'est le cas pour les commandes `break`, `continue`, `delete`, `frame`, `help`, `info`, `next`, `print`, `quit`, `run` et `step`.

16 Utiliser l'historique des commandes

Il est possible d'activer sous `gdb` l'historique des commandes, afin de ne pas avoir à retaper sans cesse la même commande. Pour activer cette fonctionnalité, il suffit d'entrer la commande

```
(gdb) set history expansion
```

Dans ce cas, comme sous Unix, `!!` rappelle la dernière commande exécutée et `!`*`caractère`* rappelle la dernière commande commençant par ce caractère. Par exemple,

```
(gdb) !p
print nb_lignes
$75 = 1073928121
(gdb)
```

Cette fonctionnalité n'est pas activée par défaut car il peut y avoir ambiguïté entre le signe `!` permettant de rappeler une commande et le `!` correspondant à la négation logique du langage C.

17 Interface avec le shell

On peut à tout moment sous `gdb` exécuter des commandes shell. Les commandes `cd`, `pwd` et `make` sont disponibles. Plus généralement, la commande `gdb` suivante

`(gdb) shell` *`commande`*

exécute *`commande`*.

18 Résumé des principales commandes

commande		action	page
`backtrace`	`bt`	indique où l'on se situe dans la pile des appels (synonyme de `where`)	4
`break` (`M-x SPC`)	`b`	pose un point d'arrêt à une ligne définie par son numéro ou au début d'une fonction.	8
`clear`	`cl`	détruit tous les points d'arrêt sur une ligne ou dans une fonction	10
`commands`		définit une liste de commandes à effectuer automatiquement à un point d'arrêt	14
`cond`		ajoute une condition à un point d'arrêt	11
`continue`	`c`	continue l'exécution (après un point d'arrêt)	9
`delete`	`d`	détruit le point d'arrêt dont le numéro est donné	9
`disable`		désactive un point d'arrêt	10
`disable disp`		désactive un `display`	14
`display`		affiche la valeur d'une expression à chaque arrêt du programme	12
`down`		descend dans la pile des appels	7
`enable`		réactive un point d'arrêt	10
`enable disp`		réactive un `display`	14
`file`		redéfinit l'exécutable	1
`finish`		termine l'exécution d'une fonction	12
`frame`		permet de se placer à un endroit donné dans la pile des appels et affiche le contexte	8
`help`	`h`	fournit de l'aide à propos d'une commande	
`info breakpoints`	`i b`	affiche les points d'arrêt	9
`info display`		donne la liste des expressions affichées par des `display`	13
`info func`		affiche le prototype d'une fonction	6
`next`	`n`	exécute l'instruction suivante (sans entrer dans les fonctions)	11
`run`	`r`	lance l'exécution du programme (par défaut avec les arguments utilisés précédemment)	2
`print`	`p`	affiche la valeur d'une expression	5
`ptype`		détaille un type structure	6
`quit`	`q`	quitte `gdb`	2
`set history expansion`		active l'historique des commandes	17
`set variable`		modifie la valeur d'une variable	7
`shell`		permet d'exécuter des commandes shell	17
`show args`		affiche les arguments du programme	2
`show values`		réaffiche les 10 dernières valeurs affichées	5
`step`	`s`	exécute l'instruction suivante (en entrant dans les fonctions)	11
`undisplay`		supprime un `display`	13
`up`		monte dans la pile des appels	7
`whatis`		donne le type d'une expression	6
`where`		indique où l'on se situe dans la pile des appels (synonyme de `backtrace`)	4