Le débogueur GDB.*

Structures de données - Travaux dirigés sur machines

Introduction à gdb

Table des matières

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 Utilisation de gdb

1.1 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 :

*Memento extrait et adapté de Le débogueur GDB de Anne Canteaut

```
> gcc -g -std=c99 -Werror -W -Wall -ansi -pedantic -o exemple exemple.c
```

On peut ensuite lancer gdb sous le shell par la commande

```
gdb nom_executable
```

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 redéfinir l'exécutable courant à l'aide de la commande `file` :

```
(gdb) file nom_executable
```

1.2 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.

1.3 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 > sortie
```

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

```
./exemple 3 > 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 > 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 > 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)
```

2 Analyse de l'environnement d'éxécution

2.1 Terminaison anormale du programme

Dans toute la suite, on prendra pour exemple le programme suivant

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

int * initFunc(int i) {
        int *p = 0;
        *p = i;
        return p;
}

int main(int argc, char **argv) {
        if ( argc > 1 ) {
                int k = atoi(argv[1]);
                int *p = initFunc(k);
                printf("%p --> %d\n", (void *)p, *p);
                return 0;
        } else
                return 1;
}
```

Pour déboguer, on exécute donc la commande

```
$ gdb ./exemple
...
(gdb) run 2
```

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 2
Starting program: /home/paulin/Documents/L2-SD/gdb-exemple/exemple 2

Program received signal SIGSEGV, Segmentation fault.
0x00005555555546a0 in initFunc (i=2) at exemple.c:6
6               *p = i;
(gdb)
```

On en déduit que l'erreur de segmentation s'est produite à l'exécution de la ligne 6 du fichier source `example.c`, lors d'un appel à la fonction `initFunc()` avec la valeur du paramètre `i=2`.

On peut alors, pour avoir plus de renseignements sur l'état de l'éxécution du programme, utiliser 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) bt
#0  0x00005555555546a0 in initFunc (i=2) at exemple.c:6
#1  0x00005555555546dd in main (argc=2, argv=0x7fffffffeb78) at exemple.c:13
(gdb)
```

On apprend ici que l'erreur a été provoquée par la ligne 6 du fichier source `example.c`, à l'intérieur d'un appel à la fonction `initFunc()` qui, elle, avait été appelée à la ligne 13 par la fonction `main`.
L'erreur survient donc à l'initialisation du contenu de l'adresse mémoire stockée dans `p`. `gdb` donne donc une première indication de l'erreur du programme en identifiant le type d'erreur, la localisation de l'erreur dans le code source mais aussi, à travers la pile d'appel, dans son environnement d'exécution.

2.2 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 p
$1 = (int *) 0x0
(gdb) print *p
Cannot access memory at address 0x0
(gdb)
```

L'erreur provient clairement du fait que l'on tente d'écrire à l'adresse mémoire `0x0` qui est une adresse mémoire inaccessible par les programmes
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 i
$2 = 2
(gdb) print /t i
$3 = 10
(gdb)
```

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 /o i
$4 = 02
```

```
(gdb) print $4
$5 = 2
(gdb)
```

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

```
(gdb) whatis p
type = int *
(gdb) whatis i
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 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 initFunc
All functions matching regular expression "initFunc":

File example.c:
int *initFunc(int);
(gdb)
```

2.3 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 initFunc(1)

Program received signal SIGSEGV, Segmentation fault.
0x00005555555546a0 in initFunc (i=1) at exemple.c:6
6               *p = i;
```

On remarque que cette commande permet de reproduire les erreurs du programmes ou de les éviter selon le type d'erreur. Ici, changer la valeur de l'argument ne change pas le comportement erroné de la fonction. Cela confirme bien que c'est le code de la fonction qui est erroné et non pas les paramètres passés à l'appel.
On notera aussi que cette commande, en cas d'execution correcte de la fonction, affiche la valeur retournée par la fonction.

Une commande équivalente est la commande `call` :

2.4 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 `set variable nom_variable = expression`.
Cette commande affecte à `nom_variable` la valeur de `expression`.

```
(gdb) print i
$6 = 1
(gdb) set variable i = 3*5 + i
(gdb) print i
$7 = 16
(gdb)
```

Cette affectation peut également se faire de manière équivalente à l'aide de la commande print : `print nom_variable = expression` qui affiche la valeur de expression et l'affecte à variable. Il est à noter que expression peut être l'appel à une fonction visible du programme, voire à une fonction d'une des bibliothèque auxquelles le programme est lié.
Dans notre exemple, on peut vérifier que l'erreur est bien une erreur d'initialisation du pointeur p en tapant la suite de commande :

```
(gdb) set variable i = 3*5 + i
(gdb) print i
$7 = 16
(gdb) set variable p = malloc(sizeof(int))
(gdb) print p
$8 = (int *) 0x7ffff7fbef00
(gdb) print *p=4
$9 = 4
(gdb) print p
$10 = (int *) 0x7ffff7fbef00
(gdb) print *p
$11 = 4
(gdb)
```

2.5 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  0x00005555555546a0 in initFunc (i=2) at exemple.c:6
#1  0x00005555555546dd in main (argc=2, argv=0x7fffffffeb78) at exemple.c:13
(gdb)
```

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

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

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

```
(gdb) up
#1  0x00005555555546dd in main (argc=2, argv=0x7fffffffeb78) at exemple.c:13
13                    int *p = initFunc(k);
(gdb)
```

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) frame
#1  0x00005555555546dd in main (argc=2, argv=0x7fffffffeb78) at exemple.c:13
13                    int *p = initFunc(k);
(gdb)
```

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

```
(gdb) print k
$1 = 2
(gdb)
```

3 Points d'arrêt et contrôle de l'exécution

3.1 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 initFunc et l'autre avant la ligne 6 du fichier, qui correspond à l'instruction responsable de l'erreur du programme :

```
(gdb) break initFunc
Breakpoint 1 at 0x555555554691: file exemple.c, line 5.
(gdb) break 6
```

```
Breakpoint 2 at 0x555555554699: file exemple.c, line 6.
(gdb)
```

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
```

Lorsque l'on utilise un environnement de développement intégré, il suffit en général, pour placer un point d'arrêt, de se placer sur la ligne correspondante et de cliquer dans la marge gauche de l'editeur.
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 la variables `p`, qui correspond à l'adresse mémoire dans laquelle on souhaite écrire la valeur `i`, évolue 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/paulin/Documents/L2-SD/gdb-exemple/exemple 2

Breakpoint 1, initFunc (i=2) at exemple.c:5
5               int *p = 0;
(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 1 situé à la ligne 5 du fichier. On peut alors faire afficher les valeurs de certaines variables, les modifier... Par exemple, ici,

```
(gdb) print i
$2 = 2
(gdb) print i=3
$3 = 3
(gdb)
```

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.

Breakpoint 2, initFunc (i=3) at exemple.c:6
6               *p = i;
(gdb)
```

On remarque ici que la variable i a bien la valeur 3 que l'on avait fixé lors de la commande précédente.

3.2 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   0x0000555555554691 in initFunc at exemple.c:5
        breakpoint already hit 1 time
2       breakpoint     keep y   0x0000555555554699 in initFunc at exemple.c:6
        breakpoint already hit 1 time
(gdb)
```

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

```
(gdb) delete numero_point_arret
```

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.

3.3 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 6 (juste avant l'écriture dans la zone mémoire pointée par p) n'est vraiment utile que si la valeur de p est anormale (0 par exemple).
On peut donc utilement remplacer le point d'arrêt numéro 2 par un point d'arrêt conditionnel :

```
(gdb) break 5 if i!=2
Breakpoint 3 at 0x555555554691: file exemple.c, line 5.
(gdb)  info breakpoints
Num     Type           Disp Enb Address            What
2       breakpoint     keep y   0x0000555555554699 in initFunc at exemple.c:6
        breakpoint already hit 1 time
3       breakpoint     keep y   0x0000555555554691 in initFunc at exemple.c:5
        stop only if i!=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 2 et pas au point d'arrêt numéro 3 placé avant. Ceci implique que la variable `i` a bien bien la bonne valeur à l'appel de initFunc :

```
(gdb) r 2
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/paulin/Documents/L2-SD/gdb-exemple/exemple 2

Breakpoint 2, initFunc (i=2) at exemple.c:6
6               *p = i;
(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.

3.4 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.
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.
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.

3.5 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. La commande `info display` (raccourci `i display`) affiche la liste des expressions faisant l'objet d'un display et les numéros correspondants.

3.6 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 la commande `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`.
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.
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`.