3IF Environnment de programmation: Utilisation d'un débogueur

Wikipedia.fr :

> Un débogueur (ou débugueur, de l'anglais debugger) est un logiciel qui aide un développeur à analyser les bugs d'un programme. Pour cela, il permet d'exécuter le programme pas-à-pas, d'afficher la valeur des variables à tout moment, de mettre en place des points d'arrêt sur des conditions ou sur des lignes du programme ...

1 Déboguage d'un programme en C

1.1 Compiler un programme pour le debugging

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

Dans cet exemple, vous allez debugger le programme de la figure 1.1, une fois ce code copié dans le fichier `prog.c`:

On compile donc le programme à l'aide de la commande:

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

et lancez `gdb` sur ce programme ainsi:

```
gdb prog
  << message d'accueil... >>
(gdb)
```

1.2 Exécuter le programme sous `gdb`

On peut exécuter le programme à l'aide de la commande `run` de `gdb`:

```
(gdb) run
Starting program: ./prog
13 et 20

Program exited normally.
```

```c
#include <stdio.h>

int max(int a, int b)
{
  int c=a;
  if (a<b){ c=b; };
  return c;
}

int f(int x)
{
  return 2*x+max(x+4,2*x);
}

int main()
{
  printf("%d et %d\n",f(3),f(5));
  return 0;
}
```

Figure 1: `prog.c`

L'exécution se déroule comme si le programme tournait "normalement". Lorsque l'utilisateur de `gdb` a la main, il peut choisir de terminer la session en tapant `quit`.

On peut exploiter `gdb` pour examiner le programme à différentes étapes de son exécution: pour cela, il faut introduire des *points d'arrêt* où l'exécution s'interrompra.

1.3 Poser des points d'arrêt

C'est la commande `breakpoint` qui permet d'indiquer des points d'arrêt. Vous pouvez fournir un numéro de ligne dans le code source, ou bien le nom d'une fonction (l'exécution s'interrompra alors à chaque appel à cette fonction).

Dès lors, si vous lancez l'exécution, `gdb` interrompt l'exécution du programme et redonne la main à l'utilisateur lorsqu'il rencontre un point d'arrêt:

```
(gdb) break f
Breakpoint 1 at 0x804840b: file code_gdb.c, line 15.
(gdb) run
Starting program: ./prog

Breakpoint 1, f (x=5) at code_gdb.c:15
15        return 2*x+max(x+4,2*x);
```

NB: vous pouvez également introduire des *"watchpoints"*, à l'aide de la commande `watch`, qui ont pour effet d'interrompre l'exécution lorsque la valeur d'une variable est modifiée: on "surveille" en quelque sorte cette variable.

1.4 Examiner la situation lors d'un point d'arrêt

Lorsque l'exécution du programme est interrompue, vous pouvez examiner l'état de la mémoire à ce moment là, par exemple en affichant la valeur d'une variable à l'aide de la commande `print`:

```
(gdb) print x
$1 = 5
```

Si on utilise la commande `display` au lieu de `print`, la valeur de la variable sera affichée à chaque fois que le programme est interrompu.

On peut également utiliser la commande `list` pour se remémorer l'endroit dans le code où l'exécution a été interrompue:

```
(gdb) list
10        return c;
11      }
12
13      int f(int x)
14      {
15        return 2*x+max(x+4,2*x);
16      }
17
18      int main()
19      {
```

1.5 Continuer l'exécution du programme

La commande `step` permet d'avancer pas à pas dans l'exécution, afin de bien contrôler l'évolution du programme. Elle permet de passer à la ligne suivante dans le source :

```
(gdb) step
max (a=9, b=10) at code_gdb.c:5
5         int c=a;
(gdb) step
7         if (a<b){
(gdb) step
8               c=b;
(gdb) print c
$2 = 9
(gdb) step
10        return c;
(gdb) print c
$3 = 10
```

À noter qu'il existe aussi la commande `next`, qui elle ne "descend" pas dans les appels de fonctions; ainsi, si l'on a interrompu l'exécution juste avant un appel de la forme `f(a,b)`, `next` relance le programme et l'interrompt après l'appel à `f()` (dans le code de la fonction appelante), alors que `step` s'arrête à la première ligne du code définissant la fonction `f`. Également, la fonction `stepi` exécute une instruction machine, par opposition à une ligne de code comme `step`.

L'instruction `continue`, elle, relance l'exécution jusqu'au prochain point d'arrêt.

Examiner la pile d'exécution : la commande `backtrace` permet d'afficher la pile d'exécution, indiquant à quel endroit l'on se trouve au sein des différents appels de fonctions. Ici, le processeur est en train d'exécuter la fonction `max`, qui a été appelée par `f`, elle-même invoquée par la fonction `main` :

```
(gdb) backtrace
#0  max (a=9, b=10) at code_gdb.c:10
#1  0x8048423 in f (x=5) at code_gdb.c:15
#2  0x804844f in main () at code_gdb.c:20
```

Effacer un point d'arrêt: `clear` en indiquant un numéro de ligne ou un nom de fonction, `delete` en indiquant le numéro du breakpoint (`delete` tout court efface – après confirmation – tous les points d'arrêt).

Remarque: pourquoi faut-il utiliser une option spéciale de `gcc` afin de pouvoir utiliser `gdb`? Parce que de nombreuses informations inutiles lors de l'exécution du programme ne sont pas mises, par défaut: ainsi il est a priori inutile de savoir, au cours de l'exécution du programme, à quel endroit le processeur se trouve dans le code source, ou bien quel est le nom de la variable qu'on est en train de modifier. Par contre ce genre de renseignement est utile à `gdb` afin que l'utilisateur "s'y retrouve".

2 Quelques commandes importantes sous gdb

Entre parenthèses, les abréviations que l'on peut utiliser à la place des commandes en toutes lettres.

quit (`q`) quitter `gdb`

run (`r`) lancer l'exécution

break,watch,clear,delete (`b,w,cl,d`) introduire un point d'arrêt, ou bien "surveiller" une variable

step,next,continue (`s,n,c`) avancer d'un pas (en entrant ou pas dans les sous-fonctions), relancer jusqu'au prochain point d'arrêt

print,display (`p,disp`) afficher la valeur d'une variable (une seul fois ou tout le temps)

backtrace,list (`bt,l`) afficher la pile d'exécution, afficher l'endroit où l'on se trouve dans le code.

help `cmd` afficher l'aide sur la commande `cmd`. Vous pouvez bien sûr taper `help help`.

3 Scripts gdb

Si vous lancez la commande `gdb -x <nom_fichier>.gdb <executable>`, gdb évaluera toutes les commandes gdb contenues dans le fichier `<nom_fichier>.gdb`. Cela est très utile pour ne pas avoir à taper systématiquement les même commandes à chaque exécution, et rendre ainsi les sessions de debug plus efficaces.

L'exemple suivant met un point d'arrêt à la fonction `max` et affiche le contenu de la variable `a` à chaque interromption de gdb.

```
break max
run
display a
```

Une autre fonctionnalité très utile est de pouvoir définir vos propres commandes. Vous pouvez par exemple n'afficher des infos de debug que dans certaines conditions, ou afficher plein d'infos sans avoir besoin de retaper des dizaines de lignes de code.

L'exemple ci-dessous crée une commande `abc` qui affichera le contenu des variables `a`, `b` et `c` à chaque fois que vous l'invoquerez.

```
define abc
  print "a: "
  print a
  print "b: "
  print b
  print "c: "
  print c
end
```

4 Déboguage bubblesort

Déboguer le programme fourni `bubblesort.c` qui doit trier et afficher des tableaux de valeurs ainsi que calculer leur moyennes.