Chapitre 1

Utilisation de gdb

1.1 *The name of the game*

gdb — Gnu [1] DeBugger — est un *débogueur* c'est-à-dire un outil de mise au point de programmes. Le terme français officiel pour *bug* — erreur provoquant un dysfonctionnement d'un programme — est *bogue*.

gdb peut s'utiliser *in vivo*, pour contrôler l'exécution d'un programme et inspecter ses variables, ou *post mortem*, pour déterminer les causes de la mort d'un programme. L'utilisation *in vivo* permet de traquer les raisons d'un comportement anormal mais pas fatal du programme. L'utilisation *post mortem* utilise le fichier core que Linux crée lorsqu'il tue un programme qui tente une opération illégale (par exemple, lire ou écrire à une adresse nulle, ou dans de la mémoire qui ne lui a pas été attribuée).

gdb est un débogueur *symbolique*, c'est-à-dire qu'il permet de faire référence aux variables et aux fonctions du programme par leur nom, et pas uniquement par leur adresse en mémoire. Il a besoin pour cela d'informations telles que le nom et le type de la variable se trouvant à telle adresse, le nom, le type des arguments et de la valeur de retour d'une fonction, ou encore le numéro de la ligne du fichier source à laquelle correspond l'instruction que le programme est en train d'exécuter. Toutes ces informations sont fournies par le compilateur C lorsqu'on utilise l'option `-g`. Elles sont conservées et mises à jour lors de l'édition de liens grâce à la même option.

Important : Pour bénéficier de toutes les possibilités de gdb, pensez à compiler et à faire l'édition de liens avec `dcc -g`.

1.2 Une session *post mortem*

Pour vous familiariser avec gdb, nous allons commencer par faire l'autopsie du programme bug dont les derniers mots furent :

```
> ./bug
argh!
Segmentation fault (core dumped)
```

La dernière ligne ci-dessus nous indique que bug a été tué à cause d'une « faute de segmentation ». En clair, cela signifie qu'il a tenté de lire ou d'écrire dans de la

1. voir http://www.gnu.org pour plus de détails sur le projet GNU et la FSF

mémoire qui ne lui était pas attribuée. Ceci étant considéré comme une faute très grave par Linux (et d'autres), bug a été tué immédiatement, et le contenu de toute la mémoire qu'il utilisait est exposé dans le fichier core pour l'édification du peuple.
Utilisons gdb pour autopsier bug et comprendre les causes d'une telle fin :

```
> gdb bug core
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it
under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty"
for details.
This GDB was configured as "i386-redhat-linux"...
Core was generated by `./bug'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libc.so.6...done.
Reading symbols from /lib/ld-linux.so.2...done.
#0 0x80485af in ajoute (l=0x0, c=97 'a') at bug.c:43
43     prec->suiv = nouv;
(gdb)
```

Ici, nous lançons gdb avec deux arguments : le nom du programme et le nom de l'image mémoire que nous voulons examiner. Après les présentations d'usage, gdb nous informe que le fichier core a bien été généré par le programme ./bug lorsqu'il a reçu le signal numéro 11 qui correspond à une faute de segmentation.

Afin de pouvoir faire référence symboliquement aux fonctions de la bibliothèque C standard (printf, exit, etc), il lit ensuite les symboles de certaines bibliothèques.

Enfin, il nous indique que bug exécutait une instruction à l'adresse 0x80485af lorsqu'il a été tué, que cette instruction se trouve dans la fonction ajoute, à la ligne 43 du fichier bug.c. Cette fonction était appelée avec un premier argument l de valeur 0x0, et un deuxième argument nommé c de valeur 97, ce qui correspond au code du caractère a.

gdb nous affiche enfin la ligne 43 de bug.c, et on se doute que l'erreur est sans doute due à une valeur incorrecte du pointeur prec.

Le #0 indique que la fonction ajoute est la dernière fonction de la chaîne d'appels de fonctions du programme. On peut connaître la chaîne complète grâce à la commande bt (back-trace) ou à la commande where :

```
(gdb) bt
#0 0x80485af in ajoute (l=0x0, c=97 'a') at bug.c:43
#1 0x8048501 in main () at bug.c:22
```

On voit ainsi que ajoute a été appelée par main à la ligne 22 du fichier bug.c. On peut se placer dans le contexte de main en remontant d'un niveau d'appel, grâce à la commande up :

```
(gdb) up
#1 0x8048501 in main () at bug.c:22
22      l = ajoute(l, c);
```

Le #1 nous indique que nous sommes un niveau d'appel au dessus de l'instruction courante. On peut examiner les variables l et c grâce à la commande print :

```
(gdb) print l
$1 = 0x0
(gdb) print c
$2 = 97 'a'
```

Pour retourner dans le contexte de `ajoute`, on peut utiliser la commande `down` pour re-descendre d'un niveau d'appel, ou aller directement au niveau 0 avec `frame 0`. Sans argument, la commande `frame` affiche une brève description du niveau courant :

```
(gdb) down
#0 0x80485af in ajoute (l=0x0, c=97 'a') at bug.c:43
43     prec->suiv = nouv;
```

Examinons maintenant la valeur de `prec` :

```
(gdb) print prec
$3 = 0x0
```

`prec` étant nul, il n'est pas étonnant que `prec->suiv` ai provoqué une erreur. Nous maintenant utiliser `gdb` *in vivo* pour déterminer pourquoi `prec` est nul. Nous terminons la session actuelle grâce à la commande `quit` :

```
(gdb) quit
>
```

1.3 Une session *in vivo*

On lance cette fois `gdb` avec pour seul argument le nom du programme à déboguer :

```
> gdb bug
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License,
and you are welcome to change it and/or distribute copies of it
under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty"
for details.
This GDB was configured as "i386-redhat-linux"...
(gdb)
```

Pour étudier le comportement de la fonction `ajoute`, nous demandons à `gdb` de s'arrêter au début de son exécution. On dit que l'on place un point d'arrêt au début de la fonction `ajoute` :

```
(gdb) break ajoute
Breakpoint 1 at 0x8048530: file bug.c, line 30.
```

`gdb` répond en indiquant qu'il vient de placer le point d'arrêt numéro 1 à l'adresse `0x8048530`, ce qui correspond à du code se trouvant à la ligne 30 du fichier bug.c.

Nous demandons maintenant à `gdb` de débuter l'exécution de `bug` grâce à la commande `run` :

```
(gdb) run
Starting program: /opt/users/si/boulanger/bug
```

Rien ne se passe car `bug` lit sur l'entrée standard et est donc en attente de caractères fraîchement tapés :

```
argh!

Breakpoint 1, ajoute (l=0x0, c=97 'a') at bug.c:30
30     Liste prec = VIDE, cour = l;
```

Le mot `argh!` n'étant pas tapé en réponse à l'invite de `gdb`, il est disponible sur l'entrée standard pour `bug` dont l'exécution se poursuit jusqu'à l'appel de la fonction `ajoute`, où le point d'arrêt la stoppe. `gdb` nous indique alors la raison de l'arrêt, le contexte, et la prochaine ligne qui sera exécutée. La commande `next` permet d'exécuter cette ligne et de s'arrêter juste avant la suivante :

```
(gdb) next
31     Liste nouv = malloc(sizeof(struct _Liste));
```

En utilisant de nouveau `next`, nous nous arrêtons après le `malloc`, et nous pouvons examiner la nouvelle valeur de `nouv` :

```
(gdb) next
33     while ((cour != VIDE) && (c > cour->val)) {
(gdb) print nouv
$1 = 0x8049950
```

Comme `nouv` n'est pas nul, nous pouvons demander à `gdb` d'afficher ce sur quoi il pointe. Nous voyons au passage que la commande `print` peut afficher une expression C quelconque :

```
(gdb) print *nouv
$2 = {val = 0 '\000', suiv = 0x0}
(gdb) print nouv->val
$3 = 0 '\000'
(gdb) print nouv->suiv
$4 = (struct _Liste *) 0x0
```

Avant l'appel à `malloc`, au lieu d'utiliser `next` qui exécute la ligne de code suivante, nous aurions pu utiliser la commande `step` qui exécute le code jusqu'à ce que l'on change de ligne dans le code source :

```
(gdb) step
__libc_malloc (bytes=8) at malloc.c:2626
2626 malloc.c: No such file or directory.
```

Que s'est-il passé ? Le code de la fonction `malloc` n'étant pas sur la même ligne que l'appel à `malloc`, `step` a arrêté l'exécution de `bug` à la ligne 2626 du fichier malloc.c dont nous ne disposons pas.

La différence entre `next` et `step` est que cette dernière fait rentrer dans les appels de fonction, ce qui est intéressant pour suivre le déroulement d'un algorithme, mais assez gênant lorsqu'on rentre par inadvertance dans le code de malloc... Heureusement, la commande `finish` permet de sortir rapidement d'une fonction :

```
(gdb) finish
Run till exit from #0 __libc_malloc (bytes=8) at malloc.c:2626
0x8048544 in ajoute (l=0x0, c=97 'a') at bug.c:31
31     Liste nouv = malloc(sizeof(struct _Liste));
Value returned is $1 = (void *) 0x8049950
```

`finish` arrête l'exécution juste après le retour de la fonction et indique la valeur rendue. L'inspection de `nouv` montre bien que l'affectation du résultat de `malloc` n'a pas encore eu lieu :

```
(gdb) print nouv
$2 = 0x4000a610
```

À force de `next`, nous arrivons finalement à l'instruction incriminée, avec `prec` nul :

```
(gdb) next
43     prec->suiv = nouv;
db) print prec
$3 = 0x0
```

Nous comprenons alors notre erreur : quand `prec` est nul, il suffit que `ajoute` rende `nouv`. Pour vérifier la validité de notre correction, nous allons forcer `ajoute` à rendre `nouv`, sans exécuter la déréférence fatale :

```
(gdb) return nouv
Make ajoute return now? (y or n) y
#0 0x8048501 in main () at bug.c:22
22      l = ajoute(l, c);
```

L'opération est tellement radicale (on force `ajoute` à retourner une valeur de façon prématurée) que `gdb` demande confirmation.

Afin d'appliquer la même correction lorsque `prec` est nul, nous posons un point d'arrêt conditionnel à la ligne 43 :

```
(gdb) break 43 if (prec == 0)
Breakpoint 2 at 0x80485a9: file bug.c, line 43.
```

et nous supprimons le premier point d'arrêt grâce à la commande `delete`. `info breakpoints` nous permet de vérifier qu'il ne reste qu'un point d'arrêt à la ligne 43 et qu'il est associé à la condition `prec == 0` :

```
(gdb) delete 1
(gdb) info breakpoints
Num Type     Disp Enb Address What
2  breakpoint keep y 0x080485a9 in ajoute at bug.c:43
     stop only if prec == 0
```

La commande `continue` permet de relancer l'exécution du programme jusqu'au prochain point d'arrêt ou jusqu'à sa fin :

```
(gdb) continue
Continuing.

Breakpoint 2, ajoute (l=0x8049950, c=33 '!') at bug.c:43
43     prec->suiv = nouv;
```

Nous vérifions que cet arrêt est bien justifié par la nullité de `prec`, appliquons de nouveau la correction et voyons `bug` se terminer normalement :

```
(gdb) print prec
$4 = 0x0
(gdb) return nouv
Make ajoute return now? (y or n) y
#0 0x8048501 in main () at bug.c:22
22      l = ajoute(l, c);
(gdb) continue
Continuing.
[!, a, g, h, r]

Program exited normally.
```

Il ne reste alors qu'à quitter `gdb`, et à corriger le code source du programme que voici (dans sa version boguée) :

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

struct _Liste {
  char val;
  struct _Liste * suiv;
};

typedef struct _Liste * Liste;
typedef const struct _Liste * ListeConst;

#define VIDE NULL

static Liste ajoute(Liste l, char c);
static void affiche(ListeConst l);

int main() {
  char c;
  Liste l = VIDE;

  while ((c = getchar()) != '\n') {
    l = ajoute(l, c);
  }
  affiche(l);

  return EXIT_SUCCESS;
}

static Liste ajoute(Liste l, char c) {
  Liste prec = VIDE, cour = l;
  Liste nouv = malloc(sizeof(struct _Liste));

  while ((cour != VIDE) && (c > cour->val)) {
    prec = cour;
    cour = cour->suiv;
  }
  if (nouv == NULL) {
    fprintf(stderr,"# Erreur d'allocation mémoire");
    exit(EXIT_FAILURE);
  }
  nouv->val = c;
  nouv->suiv = cour;
  prec->suiv = nouv;
```

```
  return l;
}

static void affiche(ListeConst l) {
  printf("[");
  while (l != VIDE) {
    printf("%c", l->val);
    l = l->suiv;
    if (l != VIDE) {
      printf(", ");
    }
  }
  printf("]\n");
}
```

1.4 Résumé des commandes de gdb

`help`

affiche l'aide de `gdb`. En donnant une commande en argument de `help`, on obtient des informations détaillées sur cette commande

`set args` *x y z*

indique que le programme doit être invoqué avec les arguments *x y z* (qui peuvent être en nombre quelconque) lorsqu'on utilisera la commande `run`

`break` *fonc*

pose un point d'arrêt au début de la fonction *fonc*

`break` *num*

pose un point d'arrêt à la ligne *num*

`break` *loc* `if` *cond*

pose un point d'arrêt en *loc* (fonction ou numéro de ligne) qui n'arrêtera l'exécution que lorsque l'expression C *cond* aura une valeur différente de 0

`run`

lance l'exécution du programme avec les arguments choisis grâce à `set args`

`next`

exécute la ligne courante du fichier source en « sautant » par dessus les appels de fonctions (*step over*)

`step`

exécute la ligne courante du fichier source en s'arrêtant au début de chaque fonction appelée (*step into*)

`finish`

exécute le programme jusqu'à ce qu'il sorte de la fonction courante (*step out*)

`continue`

poursuit l'exécution du programme jusqu'à sa fin ou jusqu'au prochain point d'arrêt

`kill`

arrête l'exécution du programme de façon définitive (on ne peut pas reprendre avec `continue`)

`bt`

affiche la chaîne d'appels de fonctions, de `main` jusqu'à l'instruction courante. `bt` signifie *back trace*, et `where` en est un synonyme peut-être plus facile à mémoriser

`up`

sélectionne le contexte de l'appel de fonction immédiatement supérieur s'il existe. En donnant un argument numérique à `up`, on peut remonter plusieurs niveaux d'un coup.

`down`

sélectionne le contexte de l'appel de fonction immédiatement inférieur s'il existe. En donnant un argument numérique à `down`, on peut descendre de plusieurs niveaux d'un coup.

`frame`

affiche le contexte d'appel de fonction courant. C'est dans ce contexte que sont évaluées les expressions affichées par `print`

`frame` *num*

sélectionne le contexte d'appel de fonction *num*. Les contextes sont comptés à partir de 0 qui correspond au contexte de la fonction contenant l'instruction courante. Plus *num* est élevé, plus on se rapproche de `main`

`print` *expr*

évalue l'expression C *expr* dans le contexte indiqué par `frame` et affiche le résultat. Si on n'a pas sélectionné volontairement un contexte d'appel de fonction, le contexte est celui de la fonction contenant l'instruction courante (contexte 0)

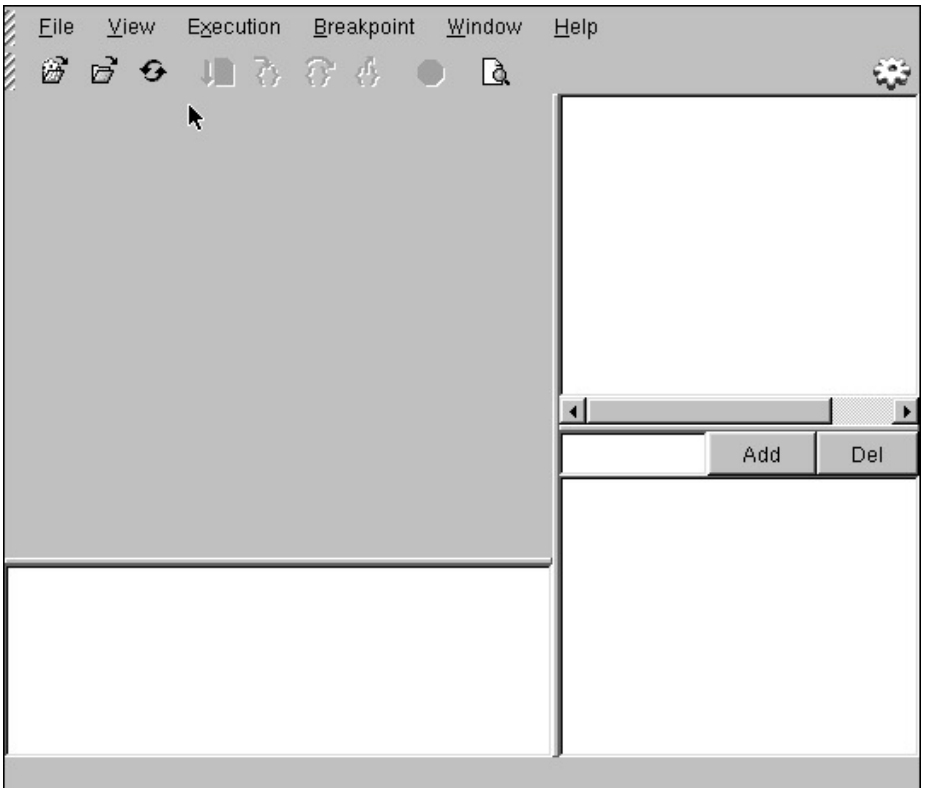
`quit`

quitte `gdb` qui demandera s'il faut tuer le programme le cas échéant

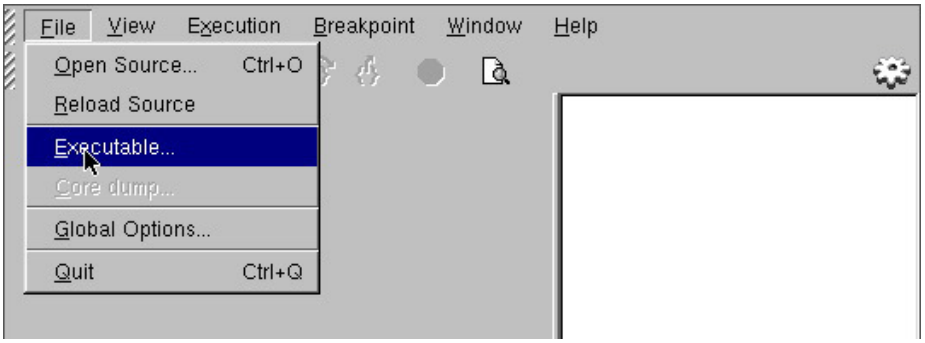
1.5 Une session plus conviviale avec `kdbg`

Les développeurs de l'environnement graphique KDE[2] ont conçu une interface graphique pour `gdb` : `kdbg`. Grâce à `kdbg`, on peut voir en même temps le code source, le contexte d'appel des fonctions, la valeur des variables locales ainsi que la valeur d'expressions choisies. Les commandes de `gdb` sont remplacées par des boutons, bien plus adaptés au comportement de l'homo clicus moderne. Voici ce qu'affiche `kdbg` au démarrage :

2. voir `http://www.kde.org/`

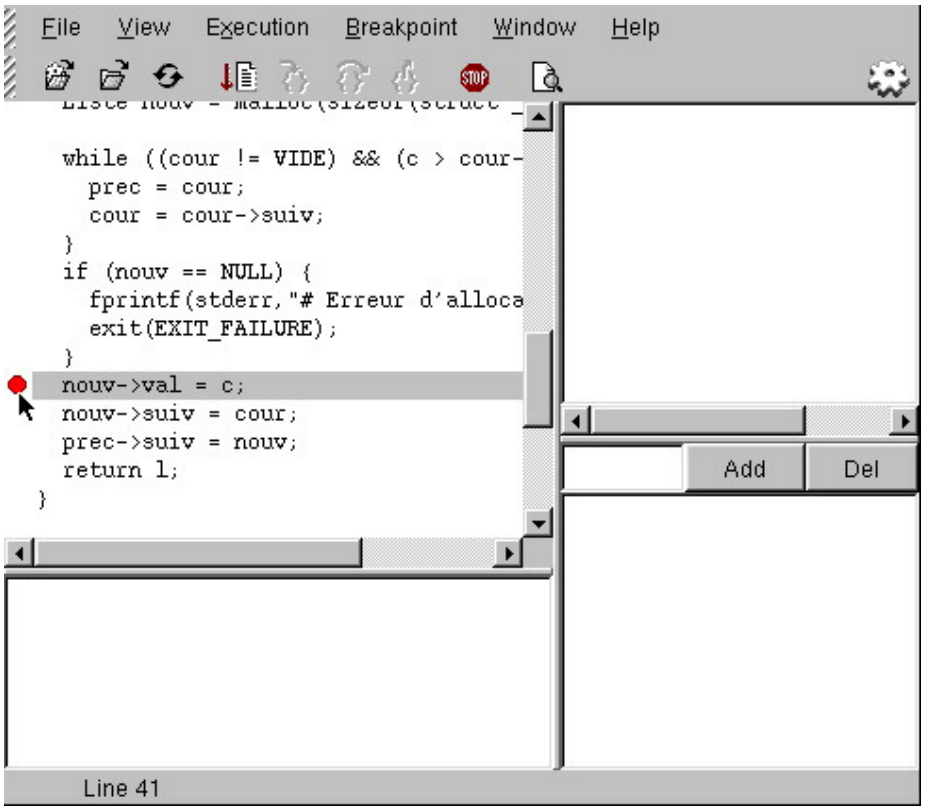


On indique à `kdbg` quel programme on souhaite déboguer grâce à l'article Executable... du menu File. Nous choisirons ici notre programme `bug`.



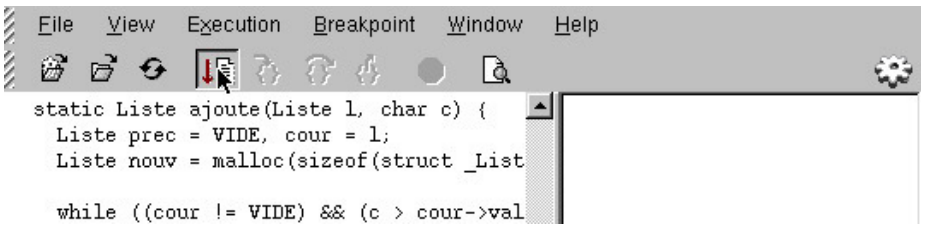
Dès que `bug` est chargé par `kdbg`, le code source de la fonction `main` est affiché. Si l'on souhaite déboguer *post mortem*, il suffit de charger le fichier core grâce à l'article Core dump... du menu File. Nous ne décrivons ici que le déroulement d'une session de déboguage *in vivo*.

Pour placer un point d'arrêt dans la fonction `ajoute`, il suffit de sélectionner la ligne on l'on veut s'arrêter et de cliquer sur le bouton **stop** (on peut aussi cliquer directement dans la marge gauche de la ligne) :

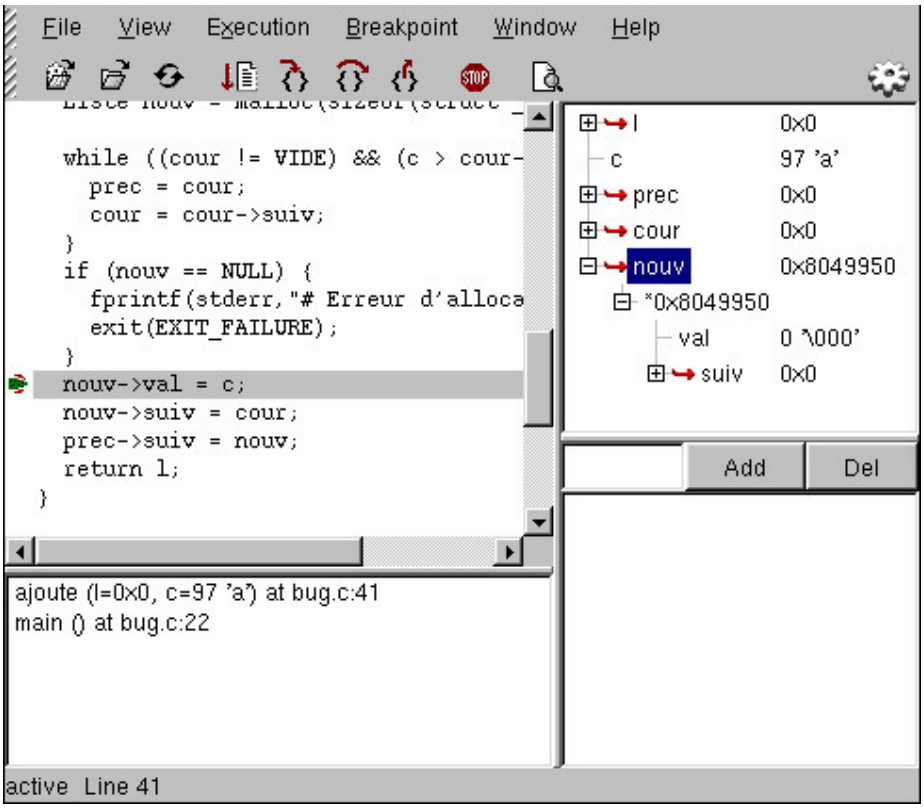


Un point rouge apparaît en regard de la ligne pour indiquer la présence du point d'arrêt. Pour supprimer un point d'arrêt, il suffit de sélectionner la ligne du code source devant laquelle il apparaît et de cliquer à nouveau sur le bouton **stop** (ou de cliquer directement sur le point rouge).

Il ne reste qu'à lancer l'exécution de `bug` :



Le programme s'exécute (le petit engrenage de KDE tourne à droite de la barre d'icône pour l'indiquer), et nous savons qu'il attend que l'on tape des caractères. `kdbg` a ouvert pour nous une fenêtre terminal qui gère les flux `stdin`, `stdout` et `stderr` du programme à déboguer. Nous tapons donc quelques caractères suivis d'un retour chariot dans cette fenêtre, et `bug` s'arrête alors sur notre point d'arrêt :



Dans la partie supérieure gauche de la fenêtre, le triangle indique la ligne avant l'exécution de laquelle le programme est arrêté (il est ici superposé au point rouge indiquant le point d'arrêt). Dans la partie inférieure gauche, `kdbg` indique le contexte d'appel des fonctions (comme la commande `bt` de `gdb`, mais sans les `#`).

La partie supérieure droite affiche la valeur des variables locales et des paramètres de la fonction. Les pointeurs et les types structurés sont précédés du signe ⊞ qui permet d'afficher ce sur quoi ils pointent ou la valeur de leurs attributs en cliquant dessus. On a ici affiché la valeur de la structure pointée par `nouv`, et les ⊞ correspondants se sont transformés en ⊟.

Enfin, la partir inférieure droite permet d'afficher la valeur d'expressions choisies. Il suffit pour cela de taper l'expression dans le champ de saisie et de cliquer sur le bouton **Add**. Pour supprimer une expression, il faut la sélectionner dans la liste et cliquer sur le bouton **Del**.

Le contrôle de l'exécution se fait grâce aux boutons suivants :


lance l'exécution du programme (`run`) ou la poursuit (`continue`),


exécute le programme jusqu'à un changement de ligne dans le code source (`step`). Ce comportement est généralement appelé *step into* puisqu'il « rentre » dans les appels de fonctions,


exécute le programme jusqu'à la ligne de code suivante (`next`). Ce comportement est généralement appelé *step over* puisqu'il « saute » par dessus les appels de fonctions,


exécute le programme jusqu'au retour de la fonction courante (`finish`). Ce comportement est en général appelé *step out* puisqu'il permet de sortir de la fonction.

On remarquera que `kdbg` ne permet pas de forcer le retour d'une fonction (`return`), mais il s'agit d'une possibilité rarement utilisée.

Les autres boutons de cette barre sont :


permet de sélectionner le programme à déboguer,


permet d'ouvrir un fichier source autre que celui contenant l'instruction courante. Cela permet par exemple de placer un point d'arrêt sur une fonction qui ne se trouve pas dans le même fichier que `main`,


lit à nouveau le fichier source affiché, ce qui est utile lorsque ce fichier a été modifié,

place ou supprime un point d'arrêt sur la ligne sélectionnée. Pour placer un point d'arrêt conditionnel, sélectionner le point d'arrêt dans la liste (article List... du menu Breakpoint) et cliquez sur le bouton **Conditional...**,


permet de chercher une suite de caractères dans le fichier source, ce qui est utile pour trouver rapidement une fonction ou une variable,


indique, lorsqu'il tourne, que le programme est en cours d'exécution.

1.6 Besoin d'aide ?

Cette documentation ne vise qu'à vous mettre le pied à l'étrier. `gdb` possède bien d'autres fonctionnalités qui peuvent se révéler très utiles lorsqu'un programme est plus complexe que notre petit exemple. La commande `help` permet d'obtenir beaucoup d'information à faible coût. Pour tout savoir sur `gdb`, le mieux est de consulter sa documentation au format info : `kdehelp 'info:(gdb)'`.