Université Paris-Nord — L1
Institut Galilée — Année 2008-2009
Licence 1 - 1er semestre

TP ddd

1 Déboguer un programme avec ddd

Un débogueur permet avant tout de suivre pas-à-pas l'exécution d'un programme, tout en permettant de visualiser les valeurs des variables de ce programme. Il est surtout utile pour trouver les erreurs présentes dans le programme (quand celui-ci se compile correctement, mais ne fait pas ce qu'on pense qu'il devrait faire).

Le débogueur le plus répandu sous Linux est `gdb`. Il fonctionne en ligne de commande, sous une forme uniquement textuelle, qui plaît à certains mais peut sembler difficile à maîtriser. Il existe également de nombreux débogueurs graphiques, qui en général ne sont que des frontends de `gdb`.

L'objectif de ce TP est de commencer à vous familiariser avec le débogueur **ddd**, *the Data Display Debugger*, dont vous trouverez par ailleurs de nombreux didacticiels en cherchant `ddd` dans Google !

Pour cette initiation, commencez par entrer le programme suivant, qui calcule la somme des carrés des n premiers entiers positifs : entrez ce programme dans un éditeur de texte, et sauvegardez-le sous le nom `sommeCarres.c` !

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

int main()
{
  int n, i, c, s;

  printf("Entrez un nombre positif : ");
  scanf("%ld", &n);

  s = 0 ;

  for (i=1; i<n; i=i+1)
    {
```

```
        c = i*i ;
        s = s+c ;
        i = i+1 ;
    }

  printf("La somme des carres des %ld premiers entiers est %ld\n", n, s);
  return EXIT_SUCCESS;
}
```

1.1 Compilation et exécution

Dans une console, placez-vous dans le répertoire qui contient `sommeCarres.c`, et tapez la commande :

```
gcc sommeCarres.c -o sommeCarres
```

Si `gcc` rapporte des erreurs dans le programme, corrigez-les, puis relancez la commande jusqu'à ce qu'il n'y en ait plus.

Lancez ensuite le programme en tapant la commande :

```
./sommeCarres
```

Entrez un entier. Trouvez-vous le résultat normal ?

1.2 Compilation et déboguage

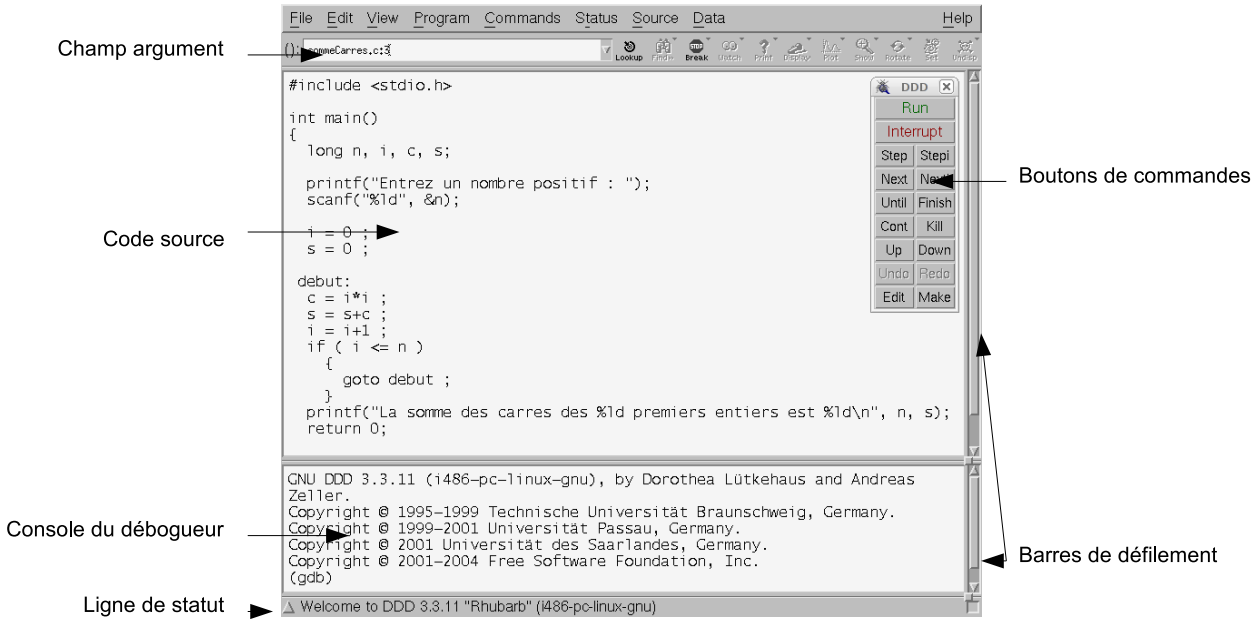
Pour pouvoir déboguer un programme en C, il faut le compiler avec l'**option** `-g` de `gcc`. Dans une console, placez-vous dans le répertoire qui contient `sommeCarres.c`, et tapez la commande :

```
gcc -g sommeCarres.c -o sommeCarres
```

1.3 Exécution du programme dans le débogueur

Après avoir compilé votre programme, lancez le débogueur en tapant la commande `ddd` !

Pour ouvrir le programme à déboguer, passez par le menu "File → Open Program...", sélectionnez le répertoire de votre programme dans la partie gauche, puis `sommeCarres` dans la partie droite, et cliquez sur "Open".



Le **code source** du programme à déboguer apparaît au centre de la fenêtre ; vous pouvez utiliser la barre de défilement pour vous promener dans ce fichier.

La **console de déboguage** contient de l'information sur la version de DDD, ainsi que l'« invite de commandes » de GDB.

```
GNU DDD 3.3.11 (i486-pc-linux-gnu), by Dorothea Lütkehaus and Andreas Zeller.
Copyright © 1995-1999 Technische Universität Braunschweig, Germany.
Copyright © 1999-2001 Universität Passau, Germany.
Copyright © 2001 Universität des Saarlandes, Germany.
Copyright © 2001-2004 Free Software Foundation, Inc.
(gdb)
```

On peut interagir directement avec GDB en tapant des commandes dans cette console, comme on peut également utiliser les **boutons de commande** ou les menus de `ddd`.

La première chose à faire est de lancer le programme, en cliquant sur le bouton de commande **Run**, ou en tapant "r" suivi de la touche "Entrée" dans la console de déboguage. Les entrées et sorties du programme se déroulent dans la fenêtre de déboguage : le programme demande d'entrer un nombre positif, puis affiche le résultat du calcul :

```
(gdb) r
Entrez un nombre positif : 4
La somme des carres des 4 premiers entiers est 10

Program exited normally.
```

Gdb affiche gentiment que le programme s'est arrêté normalement, car il a renvoyé la valeur 0 (*EXIT_SUCCESS*) en retour. Si la dernière ligne du code source était "`return 1 ;`", gdb afficherait le message :

```
Program exited with code 01.
```

Toutes les valeurs de retour du programme différentes de 0 sont considérées comme des codes d'erreur.

Les entrées-sorties du programme peuvent aussi se faire dans une fenêtre séparée, de manière à réserver la console de déboguage aux messages de gdb. Pour cela, activez le menu `View` → `Execution Window`, puis relancez le programme !

1.4 Exécution pas-à-pas

Pour observer maintenant de plus près le déroulement du programme, il faut placer un point d'arrêt (ou *breakpoint*), qui fera s' arrêter `sommeCarres` à l'endroit qui nous intéresse. Cliquez dans l'espace blanc à gauche de l'initialisation de s. Le champ argument () : contient maintenant la position (sommeCarre.c :11). Maintenant, cliquez sur `Break` pour créer un point d'arrêt à cet emplacement. Vous voyez un petit signal stop rouge apparaître dans la ligne 10.

Relancez ensuite le programme comme précédemment, et entrez le nombre 4.

L'exécution stoppe au bout d'un instant, lorsque le point d'arrêt est atteint. Cela est signalé dans la console de déboguage :

```
Breakpoint 1, main () at sommeCarres.c:11
```

La ligne couramment exécutée (en réalité, la ligne qui va être exécutée) est indiquée par une flèche verte :

⇒ s = 0 ;

Vous pouvez maintenant examiner les valeurs des variables. Pour examiner une variable simple, il suffit de placer le pointeur de la souris sur son nom et le laisser dessus. Au bout d'une seconde, une petite fenêtre surgit montrant avec la valeur de la variable. Essayez avec la variable n pour voir sa valeur (4). Les autres variables (s, i et c) ne sont pas initialisées et vous verrez probablement des valeurs fantaisistes si vous pointez dessus.

Pour exécuter la ligne courante, cliquez sur le bouton de commande **Next**. La flèche avance sur la ligne suivante. Maintenant, pointez de nouveau sur s pour voir que la valeur a changé et que cette variable a bien été initialisée à 0.

Continuez à appuyer sur Next, jusqu'à atteindre la ligne du *« i = i + 1 ; »*. On est arrivé à la fin du bloc d'instruction de la boucle *for*. Appuyez à nouveau sur Next, et le pointeur retourne sur la ligne du *for*.

Examinez les valeurs des variables i et n : la condition $i < n$ étant vraie, le prochain apppui sur Next fera entrer le pointeur d'exécution dans la boucle (comme la première fois). Continuez ainsi jusqu'à ce que le programme se termine !

Vous pouvez aussi utiliser le bouton **Until** pour forcer le programme à s'exécuter jusqu'à ce que la ligne suivante soit atteinte : relancez le programme, et lorsque la flèche verte est sur la ligne du *« for »*, cliquez sur *Until*. La première fois, l'exécution se poursuit comme avec le bouton *Next*, mais la seconde fois où vous passez sur le *« for »*, si vous cliquez sur *Until*, le programme continuera à s'exécuter sans s'arrêter jusqu'à atteindre la ligne du *printf*. Vous pouvez vérifier qu'il a effectivement effectué les instructions de la boucle en observant le contenu des variables s, c et i avant et après avoir appuyé sur *Until*.

Pourquoi cette différence de comportement entre les deux appuis sur *Until* ? Simplement parce que les termes *ligne suivante* désignent en fait l'instruction suivante dans le programme **compilé**. La ligne du *for* a été traduite par le compilateur en une série d'instructions machine, dont certaines se trouvent avant le corps de la boucle, et d'autres en fin de boucle. Vous pouvez voir le code machine correspondant au point d'exécution en utilisant le menu **View ⇒ Machine Code Window**. En déroulant le programme, vous pouvez constater que le premier arrêt sur la ligne du *for* correspond à la ligne `<main+55>` du code machine (c'est le début de la boucle dans le programme compilé, qui correspond à l'initialisation de i à 1), alors que les arrêts suivants sur la ligne du *for* correspondent à la ligne `<main+84>` du code machine (c'est la fin de la boucle dans le programme compilé, et l'instruction machine correspond à l'incrémentation de i).

Pour bien voir la différence entre le code source et le code compilé, vous pouvez exécuter le programme en utilisant le bouton *Nexti* au lieu de *Next* : le programme s'arrête à la prochaine instruction machine, plutôt qu'à la prochaine instruction dans le code source. Vous pouvez constater par exemple qu'il faut plusieurs instructions machine pour l'affectation *« c = i*i ; »*, alors qu'il n'en faut qu'une seule pour *« i = i+1 ; »*.

1.5 Examen des variables

Laisser le pointeur traîner au-dessus du nom d'une variable n'est pas le seul moyen de l'examiner. Vous pouvez également utiliser les commandes à côté du champ argument () : cliquez sur une variable afin de la faire apparaître dans le champ argument (), puis cliquez sur le bouton `Print` (il faut que le programme soit lancé et arrêté sur un point d'arrêt) : la valeur s'affiche sur la console de déboguage.

Si vous maintenez le bouton `Print` enfoncé pendant plus d'une seconde, un menu apparaît. Déplacez alors la souris sur les différentes options, et une explication s'affiche dans la barre de statut. Relâchez la souris sur `Whatis()`, et la console affiche le type de la variable (`int`).

Les commandes `Print` n'agissent qu'une fois : il faut cliquer dessus à chaque fois que l'on veut un renseignement sur une variable. En revanche, la bouton `Display` permet d'observer la

variable du champ argument de façon permanente. Lorsque vous l'utilisez, vous voyez apparaître la variable observée dans une nouvelle sous-fenêtre, avec sa valeur. Vous pouvez répéter l'opération avec plusieurs variables, qui vont apparaître les unes au-dessous des autres. Utilisez la barre de défilement pour les examiner. Vous pouvez également réorganiser leur disposition par des glisser-déposer pour en avoir une meilleure vue. Déroulez ensuite le programme pas-à-pas : vous pouvez observer les valeurs des variables qui se modifient à chaque instruction. Lorsqu'une valeur vient d'être modifiée par une instruction, elle apparaît sur fond jaune.

Vous pouvez obtenir une vue compacte de plusieurs variables de la façon suivante : sélectionnez les cadres des variables que vous voulez regrouper, en maintenant la touche *Shift* (appelée aussi *Maj.* et représentée par une flèche vers le haut) enfoncée pendant que vous cliquez sur les cadres ; ensuite, utilisez le bouton `Undisp` → `Cluster()`, ce qui les fait apparaître dans un unique cadre nommé `Displays`. Pour réaliser l'opération inverse, sélectionnez le cadre `Displays`, et utilisez `Undisp` → `Uncluster()`.

Pour supprimer l'affichage d'un cadre dans cette fenêtre, sélectionnez le cadre, et cliquez simplement sur `Undisp`.

Vous pouvez également modifier la valeur d'une variable en la faisant apparaître dans le champ argument () puis en cliquant sur le bouton `Set`.

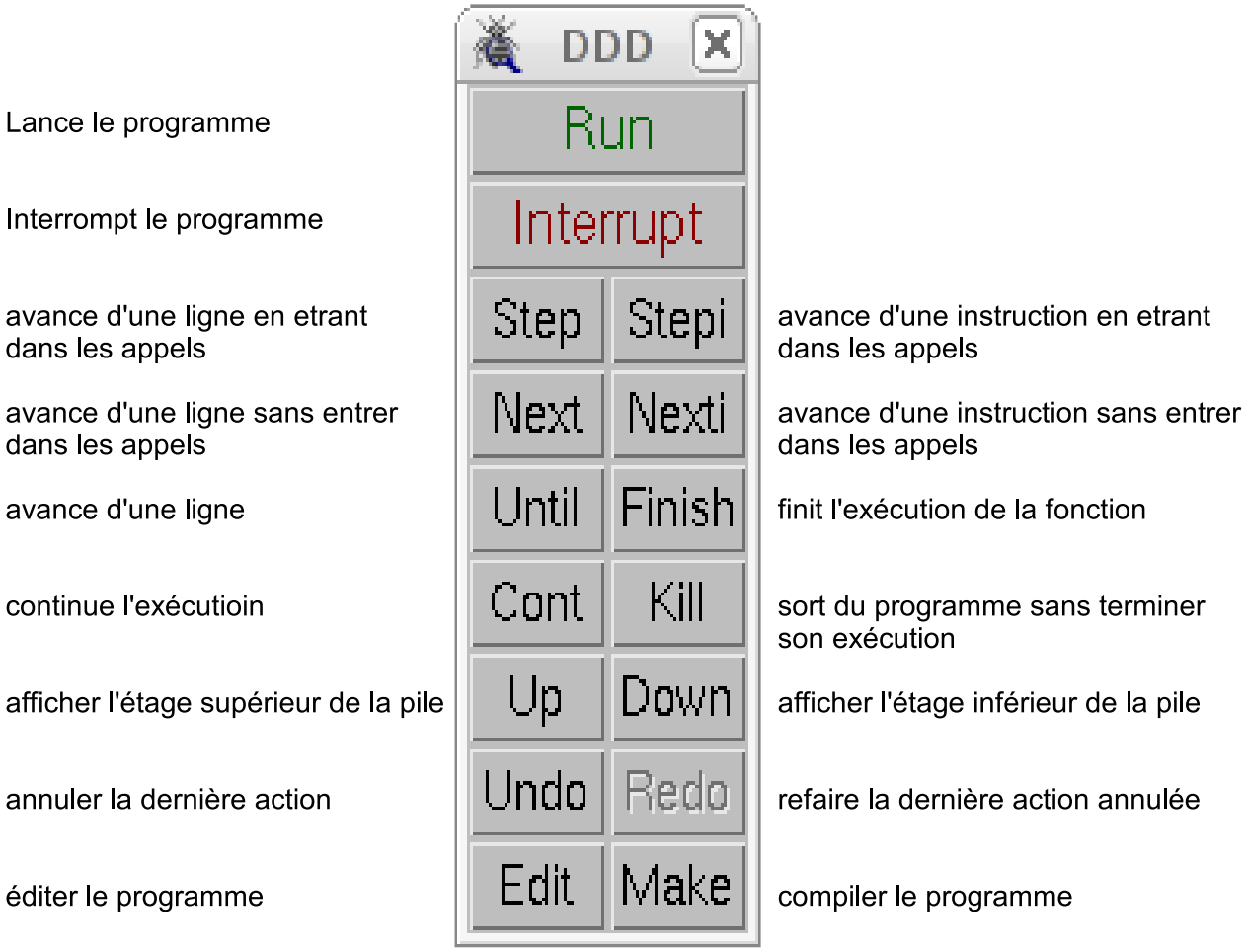
1.6 à l'action !

Affichez dans le cadre `Displays` un cluster contenant toutes les variables du programme.

Examinez particulièrement le contenu de la variable i au second passage sur la ligne $c = i*i;$. Cela doit vous mettre sur la piste d'une première erreur. Corrigez cette erreur et recompilez le programme avant de le relancer dans le débogueur.

Le résultat devrait toujours être faux. Pour voir d'où vient l'erreur, déroulez à nouveau le programme en vous concentrant sur le dernier passage dans la boucle. Corrigez, recompilez, relancez !

Finalement, lorsque vous entrez 0 comme valeur de n, le résultat devrait être faux. Cherchez pourquoi, corrigez, recompilez, relancez !


DDD
Lance le programme
Run
Interrompt le programme
Interrupt
avance d'une ligne en etrant dans les appels
Step
Stepi
avance d'une instruction en etrant dans les appels
avance d'une ligne sans entrer dans les appels
Next
Nexti
avance d'une instruction sans entrer dans les appels
avance d'une ligne
Until
Finish
finit l'exécution de la fonction
continue l'exécutioin
Cont
Kill
sort du programme sans terminer son exécution
afficher l'étage supérieur de la pile
Up
Down
afficher l'étage inférieur de la pile
annuler la dernière action
Undo
Redo
refaire la dernière action annulée
éditer le programme
Edit
Make
compiler le programme