

# Une courte introduction aux outils de développement C++ sous Linux

Travaux pratiques du cours INFO091 – Algorithmique Générale II

Steve KREMER            Gilles GEERAERTS

Année académique 2004–2005

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Le compilateur gcc</b>	<b>2</b>
2.1	Compilation séparée . . . . .	2
2.2	Autres options de g++ . . . . .	3
2.2.1	L'option <code>-Wall</code> . . . . .	3
2.2.2	L'option <code>-g</code> . . . . .	3
2.2.3	L'option <code>-O</code> . . . . .	3
2.2.4	Les <i>defines</i> avec <code>-D</code> . . . . .	3
2.2.5	L'option <code>-I</code> . . . . .	4
2.2.6	Les options <code>-L</code> et <code>-l</code> . . . . .	4
<b>3</b>	<b>Les <i>makefiles</i></b>	<b>4</b>
<b>4</b>	<b>Le débogueur gdb</b>	<b>5</b>
4.1	Un exemple . . . . .	6
4.1.1	Les commandes de <code>gdb</code> . . . . .	7
4.1.2	Trouvons l'erreur... . . . . .	7
4.2	Les fichiers <i>core</i> . . . . .	9
4.3	Une interface à <code>gdb</code> . . . . .	10
<b>5</b>	<b>En guise de conclusion...</b>	<b>10</b>
<b>6</b>	<b>Quelques références</b>	<b>11</b>

## 1 Introduction

Comme tout système d'exploitation qui se respecte, Linux dispose de nombreux outils de développement, dont les noms rappellent leurs lointains ancêtres, qui tournaient sous les premières versions de UNIX (comme vous le savez certainement, les naissances du C et de UNIX sont étroitement liées).

Comme souvent sous Linux, ces outils ne se distinguent pas vraiment par leur convivialité ! Le compilateur `gcc`, par exemple, fonctionne essentiellement en ligne de commande, ce qui implique une période d'apprentissage non négligeable pour l'utilisateur. Celui-ci doit en effet se familiariser avec les différentes options (la *man page* de `gcc` en recense une quantité impressionnante). On est

donc bien loin des interfaces plus ou moins intuitive proposées par Microsoft Visual C++ et ses avatars.

Néanmoins, si un compilateur disposant d'une interface graphique conviviale semble « plus simple » du point de vue de l'utilisateur, il ne faut pas oublier que l'interface cache une grande partie de la complexité du processus de compilation. Suivant le principe éprouvé que l'on ne « fait bien que ce que l'on comprend bien », il est donc très pédagogique de se familiariser avec un compilateur en ligne de commande comme `gcc`. Par ailleurs, une fois la phase d'apprentissage passée, on aura alors à sa disposition toute la flexibilité et toute la puissance d'un excellent compilateur.

Enfin, remarquons que `gcc` dispose lui aussi d'interfaces, comme `kdevelop`, qui ont le grand avantage d'être fortement découplées du compilateur sous-jacent.

Ce petit document n'a pas pour ambition de vous présenter `gcc`, `gdb` et `make` dans leurs moindres détails. Cela n'aurait d'ailleurs aucun sens, car rien ne remplace la pratique quand il s'agit d'apprendre à utiliser ce genre d'outils. Ce document devrait pourtant vous donner les pré-requis nécessaires à une pratique basique de ces outils, qui devrait se révéler largement suffisante dans le cadre du TP et des projets.

## 2 Le compilateur `gcc`

`gcc` est le compilateur par excellence sous Linux. Il permet de compiler une grande quantité de langages, allant du C, à l'Ada, en passant par Java, Fortran... Suivant le langage dans lequel est écrit le fichier source, on appelle `gcc` avec un nom différent. Dans le cadre du C++, on utilise `g++`.

Considérons l'exemple (classique) suivant (que nous sauvons dans le fichier `hello.C`) :

```
#include <iostream>

using namespace std;

int main (void)
{
    cout <<"Hello World !"<<endl ;
    return 1 ;
}
```

Si nous nous contentons de taper :

```
g++ hello.C
```

`g++` ne produit aucun message mais compile notre programme. Il crée un exécutable nommé `a.out` que nous pouvons lancer en tapant :

```
./a.out
```

### 2.1 Compilation séparée

Nous pouvons aussi demander à `g++` de se contenter de *compiler* notre fichier source, sans effectuer la *linkage*, et ce en utilisant l'option `-c`. Par exemple :

```
g++ -c hello.C
```

créera un fichier `hello.o`, qui contiendra le code compilé de `hello.C`. Ce fichier n'est pas exécutable, car il doit encore être lié avec les bibliothèques auxquelles il fait appel (comme `iostream`). Le *linkage* à proprement parler est effectué en rappelant `g++` sur le fichier `.o`, et en spécifiant le nom de l'exécutable, grâce à l'option `-o`. La commande :

```
g++ hello.o -o hello
```

Crée un exécutable nommé `hello`, similaire au `a.out` de l'exemple précédent.

Dans le cas qui nous concerne, nous aurions tout aussi bien pu nous contenter de faire :

```
g++ hello.C -o hello
```

qui aurait produit le même résultat. Néanmoins, il devient intéressant de séparer les étapes de la compilation quand le programme est composé de plusieurs fichiers source, que l'on compile séparément en des modules `.o`. Ces modules seront ensuite *liés* :

```
g++ module1.o module2.o ...moduleN.o -o exec
```

Procéder ainsi revêt un gros avantage quand on veut *recompiler* un programme dont on n'a modifié qu'un seul fichier source parmi un grand nombre de fichiers<sup>1</sup>. On se contente de recompiler la source `.C` grâce à `g++ -c`, et on refait le *linkage*.

## 2.2 Autres options de `g++`

La liste des différentes options de `gcc` est particulièrement longue, et nous ne les présenterons évidemment pas toutes ici. Voici néanmoins les plus courantes :

### 2.2.1 L'option `-Wall`

Les problèmes rencontrés par `gcc` durant la compilation sont classés en deux catégories : les *warnings*, qui n'empêchent pas de continuer à compiler (par exemple : assignation d'un double à un entier, ce qui cause une troncature), et les erreurs, qui arrêtent immédiatement la compilation (par exemple : des erreurs de syntaxe). L'option `-Wall` met `gcc` en mode *paranoïaque* et le force à afficher tous les *warnings*. Remarquons qu'il est également possible de sélectionner certains types de *warnings* seulement.

### 2.2.2 L'option `-g`

Une fois sélectionnée, cette option ajoute du *code de débogage* au code compilé. Voyez la section 4 consacrée à `gdb` pour plus d'informations.

### 2.2.3 L'option `-O`

Cette option enclenche les optimisations de `gcc`. Dans sa version 2.96, `gcc` reconnaît trois niveaux d'optimisation, que l'on choisit grâce à `-O1`, `-O2` ou `-O3`. Un niveau d'optimisation plus élevé rend le code plus efficace, mais ralentit d'avantage la compilation. Comme exemple d'optimisation (au niveau `-O3`), on trouve l'*inlining* des fonctions, et le déroulage des boucles.

Remarquons qu'il est toujours possible de demander un niveau d'optimisation supérieur au maximum disponible. `gcc` optimisera alors du mieux qu'il peut. Par exemple, `-O6` revient à faire `-O3` dans `gcc` 2.96, mais donnera peut-être lieu à une meilleure optimisation dans les versions suivantes.

Attention, il est hautement recommandé de ne pas combiner les options `-g` et `-O`, car le code optimisé donne de mauvais résultat dans le débogueur (car il est trop éloigné du code source).

### 2.2.4 Les *defines* avec `-D`

Cette option est passée au *pré-processeur* et permet de simuler l'effet d'un `#define` dans le code. Par exemple, la commande :

```
gcc -D_monflag_ hello.C
```

compilera `hello.C` en ajoutant la ligne

```
#define _monflag_
```

---

<sup>1</sup>Pour donner un ordre de grandeur, le traitement de texte `AbiWord` (un clone gratuit de `Microsoft Word`) comporte, dans sa version 1.0.3, près de 60 Mo de sources, réparties sur 794 fichiers `.cpp` et 1962 fichiers `.h...`

au début du code.

Un exemple d'utilisation typique est le suivant :

```
void f(...)
{
    ...
#ifdef _DEBUG_
    // Code utilisé pour le débogage
#endif
    ...
}
```

On peut alors compiler cet exemple avec

```
gcc -D_DEBUG_ Fichier.C
```

pendant la phase de développement, puis supprimer le *flag* une fois que les informations de débogage ne sont plus nécessaires.

### 2.2.5 L'option -I

Cette option permet d'étendre le chemin de recherche des fichiers `.h`. Quand il rencontre un `#include`, le pré-processeur recherche le fichier d'en-tête demandé dans une série de répertoires standards (ceux-ci contiennent les fichiers `.h` des bibliothèques standards comme `iostream`). Quand on veut utiliser certaines bibliothèques quelque peu plus « exotiques », il est souvent nécessaire d'indiquer au compilateur où trouver les fichiers d'en-tête, ainsi que le code des bibliothèques à proprement parler (voir à ce sujet le paragraphe suivant)

### 2.2.6 Les options -L et -l

Ces deux options sont destinées au *linker*. `-l` permet d'inclure des bibliothèques de fonctions déjà compilées dans un programme. Par exemple, l'ajout de `-lGL` indique au compilateur de *linker* notre code avec le fichier `libGL.so` ou `libGL.a`, suivant ce qui est disponible.

Il est souvent nécessaire d'indiquer au *linker* où trouver ces fichiers. L'option `-L` permet cela. Il s'agit, en quelque sorte, du pendant de `-I` pour les bibliothèques.

Par exemple, on peut rencontrer des commandes ressemblant à :

```
g++ -I/usr/X11R6/include -I/usr/lib/qt-2.2.1/include -L/usr/X11R6/lib
-L/usr/lib/qt-2.2.1/lib -lglut -lqt -Wall -lccmalloc ...
```

## 3 Les *makefiles*

Comme nous l'avons montré à la section 2.1, il est souvent très utile de faire de la compilation séparée. Néanmoins, recompiler à la main chaque fichier modifié (avec éventuellement une quantité gigantesque d'options) peut vite devenir une gageure. L'utilitaire `make` peut exécuter pour vous une série de tâches répétitives, basées sur des règles simples.

Ces règles sont basées sur la notion de *dépendance*. Par exemple, pour pouvoir compiler mon programme `hello` (c'est mon *but*), je dois disposer du fichier `hello.o` (`hello` dépend de `hello.o`). Une fois que je dispose de ce fichier, je peux spécifier l'*action* à effectuer (à savoir : `gcc -o hello hello.o`). Pour `make`, cette règle s'exprime ainsi<sup>2</sup> :

```
hello: hello.o
    gcc -o hello hello.o
```

On peut sauvegarder cela dans un fichier (appelé *Makefile* ou *makefile*) et appeler :

---

<sup>2</sup>Attention ! Il est indispensable de mettre une tabulation au début de la ligne de l'action. `make` est absolument intraitable à ce sujet !

```
make hello
```

pour exécuter cette règle (si aucune règle n'est spécifiée, `make` exécute la première règle par défaut).

Mais cela ne nous indique pas comment obtenir `hello.o` ! Si rien de plus n'est spécifié, `make` cherchera ce fichier dans le répertoire courant, et se plaindra s'il ne le trouve pas. Mais nous pouvons également expliquer à `make` comment produire `hello.o` à partir de `hello.C` en ajoutant la règle :

```
hello.o: hello.C
    gcc -c hello.C
```

`make` va alors suivre les règles dans le bon ordre. Pour savoir quand il doit (re)construire un fichier, `make` tient compte des dates des fichiers dans les dépendances de chaque règle. Par exemple, s'il existe déjà un fichier `hello.o` mais que le fichier `hello.C` (dont dépend `hello.o`) est plus récent que `hello.o`, `make` en déduit que les sources de ce module ont été modifiées et qu'il faut dès lors le reconstruire.

Imaginons maintenant que nous voulions rendre notre code plus portable en donnant la possibilité de le recompiler sur un système utilisant un autre compilateur C++ (le compilateur `CC` de Sun par exemple). Il nous faudrait alors changer tous les « `gcc` » en « `CC` » dans le *makefile*. `make` nous offre évidemment une alternative plus simple en nous permettant de déclarer des variables. Dans ce cas, nous allons déclarer une variable `COMP` qui va contenir le nom du compilateur. Toutes les règles feront appel à cette variable, et il suffira d'en modifier l'initialisation au début du fichier pour répercuter le changement dans toutes les règles. Au passage, nous créons une variable `FLAGS` pour nous permettre de changer facilement les options que nous passons au compilateur. Nous avons alors :

```
# Déclaration du compilateur
COMP=g++

#Flags du compilateur
FLAGS=-O3

#Règles
hello: hello.o
    $(COMP) $(FLAGS) -o hello hello.o

hello.o: hello.C
    $(COMP) $(FLAGS) -c hello.C
```

Comme on le voit les variables sont déclarées et initialisées grâce à l'opérateur `=`. On accède à la valeur de la variable `var` grâce à `$(var)`. Remarque : les lignes commençant par `#` sont des commentaires.

La dernière fonctionnalité intéressante de `make` que nous envisagerons ici est la possibilité d'écrire des *règles génériques*. Par exemple, la règle :

```
%.o : %.C
    $(COMP) $(FLAGS) -c $<
```

s'exécutera pour tous les fichiers `.C` du répertoire courant. Pour chacun d'eux, un fichier `.o` correspondant sera créé (dans ce cas, la variable `$<`, qui est une variable spéciale de `make`, prendra pour valeur les noms des différents fichiers de dépendance `.C`).

## 4 Le débogueur `gdb`

Une fois un programme compilé, il est malheureusement assez courant que son exécution ne produise pas exactement le résultat attendu... Un bon débogueur est alors fort utile, car il permet

d'exécuter le programme pas à pas, de visualiser les valeurs intermédiaires des variables, et ainsi de mieux cerner l'erreur.

`gdb` est le débogueur associé à `gcc`, c'est pourquoi il nous a semblé assez naturel de donner une brève introduction à cet outil souvent méconnu, mais pourtant ô combien précieux!

## 4.1 Un exemple

Considérons le programme suivant (sauvé dans `liste.C` et compilé en `liste`) :

```
#include <iostream>

using namespace std ;

class Liste
{
    class Noeud
    {
        friend class Liste ;

        int info ;
        Noeud * next ;
    public:
        int Info() {return info;}
        Noeud() {info = -1; next= NULL; }
        Noeud(int i) {info =i; next=NULL;}
        Noeud(int i, Noeud * n) {info =i; next =n;}
    } ;
    Noeud * rac ;

    public:

    Liste(){rac = NULL;}
    void InsereTete(int i){rac=new Noeud(i, rac);}
    void PrintLast()
    {
        Noeud * p ;
        p=rac ;
        while (p != NULL)
        {
            p = p->next ;
        }
        // Faisons une erreur de segmentation...
        cout <<"l'info du dernier élément est:"<< p->Info() ;
    }
} ;

int main (void)
{
    Liste L ;

    for (int i=0; i<20; ++i)
    {
        L.InsereTete(i) ;
    }
}
```

Commande	Effet
<code>l</code>	Affiche la ligne de code courante ainsi que les cinq précédentes et les cinq suivantes.
<code>l num.</code>	Affiche la ligne de code num.
<code>b num.</code>	Met un <i>breakpoint</i> à la ligne num. gdb arrêtera l'exécution du programme quand il rencontre cette ligne.
<code>n</code>	Exécute la ligne de code suivante. Les appels de fonction sont traités comme une seule instruction.
<code>s</code>	Même chose que <code>n</code> mais descend dans les appels de fonction.
<code>p var.</code>	Affiche le contenu de la variable var.
<code>r args.</code>	Exécute le programme chargé avec les arguments args.
<code>k</code>	Tue le programme en cours de débogage.
<code>continue</code>	Continue l'exécution.
<code>bt</code>	Affiche le <i>stack</i> des appels de fonction.
<code>up et down</code>	Permettent de descendre ou de remonter dans le <i>stack</i> des appels.

FIG. 1 – Les commandes les plus utilisées dans gdb

```

}
L.PrintLast() ;
}

```

Comme on peut s'en douter, cet exemple produit une superbe *segmentation fault*! Qu'à cela ne tienne, nous avons compilé notre programme avec l'option `-g`, et nous lançons :

```
gdb liste
```

Nous obtenons le message de bienvenue de gdb :

```

GNU gdb Red Hat Linux (5.2-2)
Copyright 2002 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)

```

#### 4.1.1 Les commandes de gdb

Nous pouvons maintenant passer des commandes à gdb. La figure 1 en présente quelques-unes parmi les plus utilisées

#### 4.1.2 Trouvons l'erreur...

Mettons ceci en pratique. Nous commençons par mettre un *breakpoint* à la ligne 38 (début du `main`), et nous lançons l'exécution :

```

(gdb) b 38
Breakpoint 1 at 0x8048660: file liste.C, line 38.
(gdb) r
Starting program: /home/ggeeraer/tmp/liste

Breakpoint 1, main () at liste.C:39
39      {
(gdb)

```

gdb atteint cette ligne sans encombre et attend nos instructions. Nous décidons d'exécuter le programme ligne par ligne en faisant une série<sup>3</sup> de n...

```
(gdb) n
main () at liste.C:40
40         Liste L ;
(gdb) n
42         for (int i=0; i<20; ++i)
(gdb) n
44             L.InsereTete(i) ;
(gdb) n
45         }
(gdb) n
44             L.InsereTete(i) ;
(gdb) n
45         }
(gdb) n
44             L.InsereTete(i) ;
(gdb) n
45         }
(gdb)
```

Tout semble bien se passer au niveau de la création de la liste. Nous décidons donc de laisser tourner le programme jusqu'à ce qu'il se plante, pour pouvoir observer les valeurs des variables à ce moment là. Cela ne tarde pas arriver, et gdb nous affiche :

```
(gdb) continue
Continuing.
```

```
Program received signal SIGSEGV, Segmentation fault.
0x080487b6 in Liste::Noeud::Info (this=0x0) at liste.C:15
15         int Info() {return info;}
(gdb)
```

gdb nous informe qu'il y a eu une erreur de segmentation dans la fonction `Liste::Noeud::Info` qui été appelée sur un objet dont le pointeur `this` vaut 0! Nous voyons donc immédiatement la cause de l'erreur : un pointeur nul! Tentons de voir d'où il provient. Commençons par observer le *stack* d'appel, grâce à `bt` :

```
(gdb) bt
#0 0x080487b6 in Liste::Noeud::Info (this=0x0) at liste.C:15
#1 0x08048786 in Liste::PrintLast (this=0xbffff824) at liste.C:34
#2 0x080486b0 in main () at liste.C:46
#3 0x42017589 in __libc_start_main () from /lib/i686/libc.so.6
(gdb)
```

La fonction `Liste::Noeud::Info` a donc été appelée par `Liste::PrintLast`. Remontons donc dans le *stack* des appels grâce à la commande `up`. Cela a pour effet de charger le contexte de la fonction appelante, ce qui nous permet d'observer les valeurs de ses variables :

```
(gdb) up
#1 0x08048786 in Liste::PrintLast (this=0xbffff824) at liste.C:34
34     cout <<"l'info du dernier élément est:"<< p->Info() ;
(gdb)
```

---

<sup>3</sup>Remarquons que presser sur `<return>` sans entrer de commande exécute la dernière commande entrée.

gdb affiche la ligne à laquelle l'appel a été effectué. Sans doute est-ce le pointeur p qui est nul ? Vérifions cette hypothèse :

```
(gdb) p p
$1 = (Noeud *) 0x0
(gdb)
```

En effet, c'est bien cela ! Pour trouver la cause de l'erreur, nous pouvons demander à gdb de nous afficher le code, grâce à l :

```
(gdb) l
29     p=rac ;
30     while (p != NULL)
31     {
32         p = p->next ;
33     }
34     cout <<"l'info du dernier élément est:"<< p->Info() ;
35 }
36}    ;
37
38     int main (void)
(gdb)
```

Nous trouvons enfin la cause de l'erreur : il faut changer la condition du while à la ligne 30 en p->next != NULL. Une fois cette ligne modifiée, nous pouvons relancer la compilation à partir de gdb, simplement en tapant make :

```
(gdb) make
g++ -g -c liste.C
g++ -g liste.o -o liste
(gdb)
```

Et nous obtenons maintenant le bon résultat :

```
(gdb) r
Starting program: /home/ggeeraer/tmp/liste
l'info du dernier élément est:0
Program exited normally.
(gdb)
```

## 4.2 Les fichiers core

En cas de plantage sous Linux, il arrive souvent que le message `segmentation fault` soit suivi d'un `core dump`. Cela signifie qu'une « photographie » de la mémoire du programme qui a planté a été sauvegardée dans un fichier appelé `core`<sup>4</sup>. On peut récupérer l'information que fournit ce fichier dans gdb. Il suffit d'appeler :

```
gdb programme core
```

L'avantage est qu'il ne faut plus exécuter le programme sous le débogueur (ce qui est forcément plus lent) pour pouvoir voir le contenu de la mémoire au moment de l'erreur. On a directement accès au contenu des variables et structures qui ont causé le plantage.

Dans ce cas, on n'a naturellement pas l'opportunité d'utiliser des commandes comme `n` ou `s`, car le programme n'est pas exécuté.

---

<sup>4</sup>Les administrateurs des systèmes de type UNIX peuvent empêcher la création de fichier `core` par les utilisateurs, et font souvent usage de cette possibilité. Cette politique se justifie par la taille généralement volumineuse des fichiers `core`. Nous avons ainsi déjà rencontré de tels fichiers dépassant les 400 Mo...

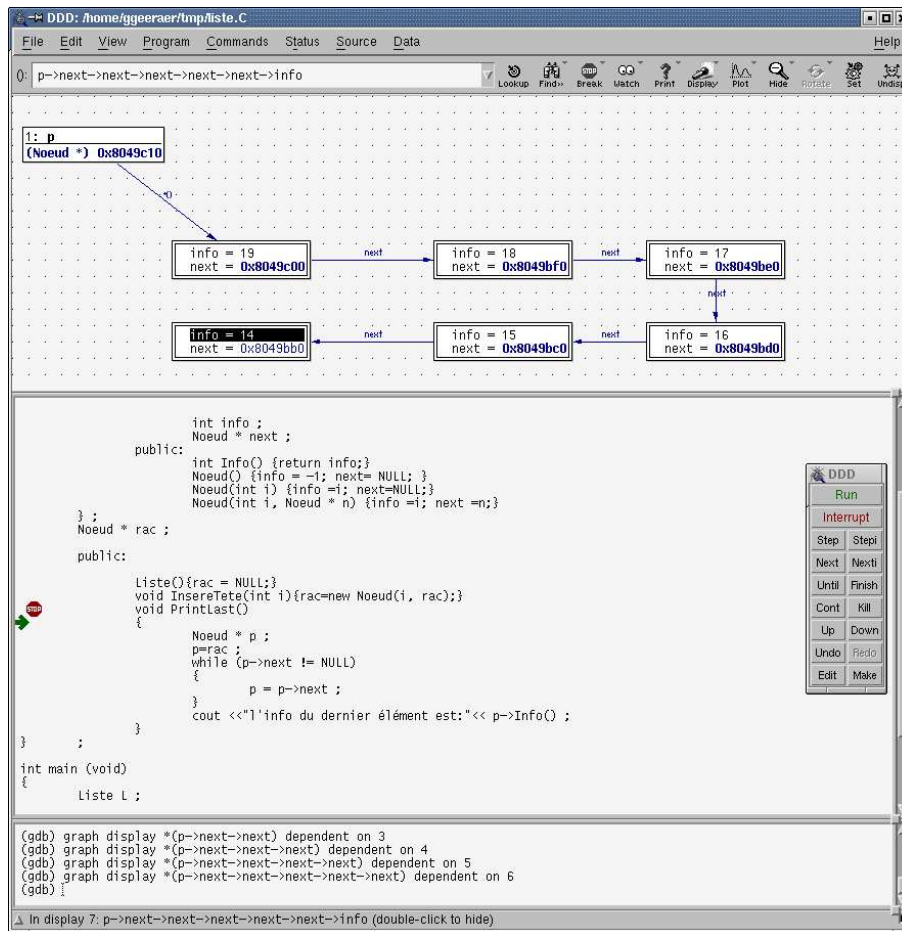


FIG. 2 – ddd en action...

### 4.3 Une interface à gdb

Pour rendre gdb plus convivial, plusieurs interfaces ont été développées. Parmi elles, on trouve ddd<sup>5</sup>, qui offre la possibilité de mettre des *displays* sur les variables. Cette fonctionnalité est illustrée à la figure 2, où l'on travaille sur la liste du précédent exemple.

## 5 En guise de conclusion...

Nous venons de terminer notre rapide tour d'horizon des principaux outils de développement sous GNU/Linux. Nous n'avons bien sûr pas couvert toutes leurs possibilités respectives, mais nous espérons que cette introduction vous donnera l'envie d'en savoir plus et d'expérimenter d'avantage avec ces outils.

Nous n'avons pas non plus passé en revue tous les outils qui existent. Parmi eux, nous pourrions citer :

- Les éditeurs de texte comme `xemacs`, `vi`, `gvim` ou `nedit`. Ceux-ci sont bien évidemment indispensables pour encoder les sources d'un programme, et se révèlent souvent des aides précieux. En effet, ils offrent des fonctionnalités très ergonomiques comme le *syntax highlighting*, l'indentation automatique, etc...
- `cvs` permet de centraliser toutes les sources d'un même programme sur un serveur. `cvs`

<sup>5</sup>Data Display Debugger. Voir :<http://www.gnu.org/software/ddd>

gère les numéros de version des fichiers, permet d'annuler des modifications, et se révèle indispensable quand on travaille en groupe sur un même programme. `cvs` possède plusieurs interfaces graphiques comme `cervisia`.

- `autoconf` qui permet de générer automatiquement des *makefiles*.
- `bash`, `perl`, etc... permettent d'écrire des petits *scripts* très rapidement.
- Et n'oublions pas les outils de plus haut niveau comme `kdevelop`, `anjuta`...

Enfin, rappelons que tous ces outils sont des *logiciels libres*, et qu'à ce titre ils sont totalement gratuits. Les sources en sont même disponibles, ce qui donne la possibilité de les modifier et des améliorer à sa guise.

## 6 Quelques références

- La *home page* de `gcc` : <http://gcc.gnu.org>
- La *home page* de `gdb` : <http://www.gnu.org/software/gdb/gdb.html>
- La *home page* de `make` : <http://www.gnu.org/software/make>
- Un tutorial sur les outils de développement sous Linux : <http://galton.uchicago.edu/~gosset/Compdocs/>
- Le tutorial de l'ACM : <http://www.cs.washington.edu/orgs/acm/tutorials/dev-in-unix/>