LES ESSENTIELS DE LA PROGRAMMATION C SOUS LINUX

DR. CHAFIKA BENZAID

MAITRE DE CONFERENCES, CLASSE B

LSI-Département Informatique,
Faculté d'Electronique & Informatique, USTHB

Email: benzaid@hotmail.com

SOMMAIRE

LA PROGRAMMATION MODULAIRE

1 Introduction

Le principe de la programmation modulaire consiste à fractionner le code du programme en plusieurs fichiers sources au lieu de mettre tout le code dans un seul fichier.

Ce style de programmation facilite grandement la **réutilisabilité** (**Reusability**) et le **partage** du code (**Code Sharing**), et est particulièrement utile pour la **réalisation de bibliothèques** (**Libraries**).

2 Cas du langage C

Dans le cas de la programmation C, un programme peut être décomposé en plusieurs **fichiers d'entêtes** (**Header Files**) portant l'extension « **.h** » et **fichiers sources** portant l'extension « **.c** ».

- Les **fichiers d'entêtes** contiennent des définitions de type et des déclarations de variables et de **prototypes de fonctions**. Ces fichiers sont inclus dans les fichiers sources par le biais de la directive **#include**.
- Les **fichiers sources** contiennent les fonctions elles-mêmes.

3 Exemple

Soit le programme suivant permettant d'afficher le carré des dix premiers entiers non nuls. Le calcul du carré est effectué par la fonction « carre ».

```
#include <stdio.h>

int carre(int nb);

int main(int argc, char* argv[]){
      int resultat;
      int i;

      for (i=1 ; i<=10 ; i++){
            resultat = carre(i);
            printf("Le carré de %d est égal à %d\n", i, resultat);
      }

      return 0;
}

int carre(int nb){
      int res;

      res = nb * nb;

      return res;
}
```

L'écriture modulaire de ce programme peut se faire par la déclaration de trois fichiers. Un fichier d'entête « **carre.h** » contenant le **prototype** de la fonction « **carre** ». Un fichier

source « **carre.c** » contenant le **code** de la fonction « **carre** ». Et, un fichier source « **main.c** » contenant la fonction « **main** ».

```
/**********************    carre.h    *************************/
#include <stdio.h>

int carre(int nb);
```

```
/**********************    carre.c    *************************/
#include "carre.h"

int carre(int nb){
      int res;

      res = nb * nb;

      return res;
}
```

```
/**********************    main.c    *************************/
#include "carre.h"

int main(int argc, char* argv[]){
      int resultat;
      int i;

      for (i=1 ; i<=10 ; i++){
            resultat =carre(i);
            printf("Le carré de %d est égal à %d\n", i, resultat);
      }

      return 0;
}
```

4 Erreurs d'inclusions multiples

Cette erreur se produit en général lors de l'inclusion multiple d'un même fichier d'entête. Pour éviter cela, la méthode consiste à définir une constante à la première inclusion du fichier d'entête et d'ignorer le contenu de celui-ci si la constante a déjà été définie.

Pour cela, on utilise la directive **#ifndef**. Il est recommandé d'appeler la constante « **__CARRE_H** » pour le fichier « carre.h ». En appliquant cette règle, le fichier « carre.h » de l'exemple précédent devient :

```
/**********************    carre.h    *************************/
#include <stdio.h>

#ifndef __CARRE_H
#define __CARRE_H
      int carre(int nb);
#endif
```

GCC

1 Introduction

GCC signifiait, au départ, **G**NU **C** **C**ompiler, mais au fur et à mesure de son évolution, il compile de plus en plus de langage (C++, objective C, fortran, ada, java), donc maintenant cet acronyme veut dire **G**NU **C**ompiler **C**ollection. GCC est portable, il fonctionne donc aussi bien sur UNIX/Linux que sur Windows ou sur MacOS.

2 Du code source à l'exécutable

Le passage depuis un code source en C vers un programme exécutable par la machine est un processus à plusieurs étapes.

2.1 Compilation

Pour compiler un fichier source C, on utilise l'option **–c** du compilateur gcc. Si la compilation réussira, un fichier objet est généré ayant l'extension « **.o** ».

```
$ gcc –c main.c
$ gcc –c carre.c
```

2.2 Edition de liens

La phase d'édition de liens permet de combiner plusieurs codes objets afin de générer le fichier exécutable. Pour invoquer l'éditeur de liens, il suffit d'appeler GCC avec les fichiers objet en paramètre.

```
$ gcc carre.o main.o
```

Le fichier exécutable généré portera par défaut le nom « **a.out** ». L'option **-o** permet de spécifier le nom du fichier exécutable à générer à l'issue de l'étape d'édition de liens.

```
$ gcc –o carre carre.o main.o
```

Dans ce cas, le fichier exécutable portera le nom « **carre** » au lieu de « a.out ».

2.3 Exécution

L'exécution du programme se fait en précisant le chemin d'accès à l'exécutable comme suit :

```
$ ./carre
```

« **./** » se réfère au répertoire courant, donc « **./carre** » permet de charger et exécuter le fichier exécutable « carre » se trouvant dans le répertoire courant.

3 Code assembleur

On obtient le code assembleur par l'utilisation de l'option « **–S »** du compilateur gcc. Le fichier produit portera l'extension « **.s** ». Rappelons que le code assembleur obtenu est **dépendant** du **processeur** utilisé.

Ci-après le code assembleur de « carre.c » créé sur un microprocesseur Intel Core(TM)2.

```
$ gcc -S carre.c
$ cat carre.s
        .file   "carre.c"
        .text
.globl carre
        .type   carre, @function
carre:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $16, %esp
        movl    8(%ebp), %eax
        imull   8(%ebp), %eax
        movl    %eax, -4(%ebp)
        movl    -4(%ebp), %eax
        leave
        ret
        .size   carre, .-carre
        .ident  "GCC: (GNU) 4.1.2 20070925 (Red Hat 4.1.2-33)"
        .section        .note.GNU-stack,"",@progbits
```

4 Options d'avertissements

GCC affiche parfois des messages d'**avertissements**, marqués « **warning:** ». Contrairement aux erreurs, ces avertissements ne sont pas fatals et **n'empêchent pas** la génération du code objet. Toutefois, ils attirent l'attention du programmeur sur les structures de programmation risquées ou potentiellement fausses. En éliminant de telles constructions, vous réduirez les risques de **bugs**. Par exemple, le compilateur affichera un avertissement sur un *type de retour incorrect* pour la fonction *main* ou pour une fonction *non void* qui ne dispose pas d'instruction *return*.

L'invocation de l'option **–Wall** du compilateur gcc permet d'activer tous les messages d'avertissement.

```
$ gcc –Wall –c carre.c
$ gcc –Wall –c main.c
```

Pour illustrer l'impact de l'ignorance de ces avertissements, prenons le cas où on spécifie, dans le fichier main.c, le **format** du deuxième paramètre à afficher par l'instruction printf comme étant un réel au lieu d'être un entier.

```
/**********************    main.c    *************************/
            ...
            printf("Le carré de %d est égal à %f\n", i, resultat);
            ...
```

Bien que GCC le compile sans rien dire, le résultat affiché est erroné.

```
$ gcc -c main.c
$ gcc -o carre main.o carre.o
$ ./carre
Le carré de 1 est égal à 0.000000
Le carré de 2 est égal à 0.000000
Le carré de 3 est égal à 0.000000
Le carré de 4 est égal à 0.000000
Le carré de 5 est égal à 0.000000
```

```
Le carré de 6 est égal à 0.000000
Le carré de 7 est égal à 0.000000
Le carré de 8 est égal à 0.000000
Le carré de 9 est égal à 0.000000
Le carré de 10 est égal à 0.000000
```

Maintenant, si on compile ce code en utilisant l'option **-Wall**, le compilateur affichera l'avertissement suivant :

```
$ gcc -Wall -c main.c
main.c: In function 'main':
main.c:9: attention : format '%f' expects type 'double', but
argument 3 has type 'int'
```

Cet avertissement indique que le format « %f » s'attend à un paramètre de type réel, cependant le paramètre est de type entier.

5 Création de bibliothèques

Nous venons de dire qu'un intérêt de la programmation modulaire est de pouvoir créer des **bibliothèques** ou librairies de fonctions dans le but de les **réutiliser** dans d'autres programmes.

L'idée est alors de regrouper dans un même fichier (**librairie**) les fonctions implémentant des fonctionnalités similaires. Par exemple, regrouper dans un même fichier toutes les fonctions permettant de réaliser des opérations arithmétiques.

A titre d'exemple, prenons le cas où on rajoute à notre programme une autre fonction calculant le cube d'un nombre entier. On place cette fonction dans un fichier nommé « **cube.c** », en définissant son fichier d'entête « **cube.h** ».

```
/**********************    cube.h    **************************/
#include <stdio.h>

#ifndef __CUBE_H
#define __CUBE_H
      int cube(int nb);
#endif
```

```
/**********************    cube.c    **************************/
#include "carre.h"

int cube(int nb){
      int res;

      res = nb * nb * nb;

      return res;
}
```

Maintenant, si on veut créer une bibliothèque, nommée **lib**power**.a**, regroupant toutes les fonctions calculant les puissances d'un nombre (dans notre cas, le carré et le cube d'un entier), on procède comme suit :

1. Compiler ces fichiers afin de générer les fichiers objet :

```
$ gcc -Wall -c carre.c cube.c
```

2. Créer la librairie **libpower.a** à l'aide de la commande **ar** :

```
$ ar r libpower.a carre.o cube.o
ar: creating libpower.a
```

3. On peut vérifier le contenu de la librairie créée en invoquant l'option **t** de la commande **ar** :

```
$ ar t libpower.a
carre.o
cube.o
```

6 Edition de liens avec des bibliothèques externes

Si la bibliothèque externe se trouve dans le même dossier contenant le fichier principal (c.-à-d. le fichier contenant la fonction main), l'édition de lien peut se faire en précisant le nom de la bibliothèque à utiliser.

```
/**********************    main.c    **************************/
#include "carre.h"
#include "cube.h"

int main(int argc, char* argv[]){
      int resultat;
      int i;

      for (i=1 ; i<=10 ; i++){
            resultat = carre(i);
            printf("Le carré de %d est égal à %d\n", i, resultat);
            resultat = cube(i);
            printf("Le cube de %d est égal à %d\n", i, resultat);
      }

      return 0;
}
```

```
$ gcc -o power main.o libpower.a
$ gcc ./power
Le carré de 1 est égal à 1
Le cube de 1 est égal à 1
Le carré de 2 est égal à 4
Le cube de 2 est égal à 8
Le carré de 3 est égal à 9
Le cube de 3 est égal à 27
Le carré de 4 est égal à 16
Le cube de 4 est égal à 64
...
```

Maintenant, on place notre librairie dans un répertoire nommé « **/lib** » se trouvant dans notre home directory (i.e. **~/lib**). Dans ce cas, on doit faire recours aux options **–L** et **–l** du compilateur gcc. Ces options permettent respectivement d'inclure un nouveau chemin de recherche pour l'´editeur de lien et d'indiquer le nom de librairie à rechercher.

- L'option **-l*bibli*** lie avec une bibliothèque externe nommée ***bibli*, pas d'espace** entre **-l** et ***bibli***.
- L'option **-L*rep*** indique où trouver les bibliothèques référencés par **–l*bibli***.

Pour notre exemple, l'édition de lien se fait en exécutant la ligne de commande suivante:

```
$ gcc -o carre main.o -L ~/lib -lpower
```

L'option « **-l** » entoure bibliothèque avec « **lib** » et « **.a** » et cherche dans plusieurs répertoires. Les répertoires de recherche incluent divers répertoires standard du système, plus tous ceux spécifiés avec l'option « **-L** ».

MAKE

1 Introduction

Lorsqu'on réalise un projet en le découpant en plusieurs modules, on constate qu'il est difficile de se rappeler des fichiers qu'il faut recompiler après une modification. L'utilitaire **make** de Linux permet d'automatiser la compilation de gros logiciels sous Linux.

La commande **make** permet de maintenir un programme automatiquement. Pour cela, elle utilisera un **fichier de description** nommé en général **makefile** ou **Makefile** qui contient des **dépendances** et les **actions** (commandes) à mettre en œuvre pour maintenir le programme.

À la différence d'un simple script shell, **make** exécute les commandes seulement si elles sont nécessaires. Le but est d'arriver à un logiciel compilé ou installé sans nécessairement refaire toutes les étapes. En d'autres termes, au lieu de compiler la totalité du code source, **make** ne construit que le code source qui a subi des changements

2 Syntaxe

La syntaxe de la commande **make** est la suivante :

```
$ make [-f nom fichier] [-arg optionnels] [ref..]
```

L'option **-f** indique le **nom** du **fichier de description**. En général, la commande est lancée **sans argument**. Dans ce cas, elle recherche un fichier de description nommé **makefile** ou **Makefile** sur le répertoire courant.

3 Contenu d'un fichier Makefile

Un fichier Makefile contient :

- Un ensemble de **règles** (**Rules**) ayant la syntaxe suivante :

```
cible: dépendances ...
<tab>  commandes
       ...
```

  - La **cible** (**Target**) est le fichier à construire. Elle est le plus souvent un nom d'exécutable ou de fichier objet.
  - Les **dépendances** (**Dependencies**) sont les éléments ou les fichiers sources nécessaires pour créer une cible.
  - Les **commandes** (**Commands**) sont les actions à lancer afin de produire la cible à partir des sources. Ces actions sont lancées suivant la modification d'un de ces fichiers sources.

  Donc, une règle spécifie à Make comment exécuter une série de commandes dans le but de construire une cible à partir des fichiers sources desquels elle dépendait.

- des **variables**, dites aussi des **macros**, qui permettent de fixer des chemins, des noms d'exécutable, …etc. Par convention, un nom de variable est écrit en **majuscules** (**Uppercases**).

  Une variable est **définie** par **son nom** et **sa valeur**.

```
VAR=Valeur
```

Et, elle est référencée (utilisée) dans les cibles, les dépendances, les commandes et les autres parties du makefile comme suit :

```
$(VAR)
```

Il existe certaines variables qui sont prédéfinies et qu'on puisse les modifiées, entre autres :

- La variable **CC** qui contient le nom du compilateur utilisé.
- La variable **CFLAGS** qui contient la liste des options de compilation.
- La variable **LDFLAGS** qui contient la liste des options d'édition de liens.

On écrit un fichier Makefile en plaçant d'abord les variables générales. Ensuite, la règle de construction de l'exécutable. Et, on termine enfin le fichier en déclarant toutes les autres règles.

4 Exemple

Ci-après le fichier Makefile permettant de construire notre application **power** :

```
#*****************************    Makefile    ***************************#
      OBJS=main.o carre.o cube.o
      EXEC=power
      CC=gcc
      CFLAGS=-Wall
power: $(OBJS)
      $(CC) $(OBJS) -o $(EXEC)
carre.o: carre.c carre.h
      $(CC) $(CFLAGS) -c carre.c
cube.o: cube.c cube.h
      $(CC) $(CFLAGS) -c cube.c
main.o: main.c carre.h cube.h
      $(CC) $(CFLAGS) -c main.c
clean:
      rm $(OBJS) $(EXEC)
```

Variables générales

La règle de construction de l'exécutable

Les règles de construction des fichiers objets

Autres règles

- Un ensemble de **variables générales** ont été définies pour quelles soient utilisées par la suite par les règles définies. La commande **make** substituera ces variables par leurs valeurs lorsque les actions sont exécutées :
  - **OBJS** est une variable contenant tous les fichiers objets nécessaires à la construction de l'exécutable. Dans cet exemple, ce sont les fichiers **main.o**, **carre.o**, et **cube.o**.
  - **EXEC** est une variable contenant le nom de l'exécutable à générer. Dans notre cas, cet exécutable sera nommé **power**.
  - **CFLAGS** est une variable de make. Cette variable contient les options de compilation. Pour notre application, on veut que tous les avertissements soient affichés, donc il suffit d'initialiser CFLAGS à **-Wall**.
- La déclaration des variables générales est suivie par une liste des **dépendances** et des **actions** associées. Le ou les fichiers qui dépendent d'autres fichiers sont précédés du caractère « **:** ».

- Il est clair que **power** dépend de **main.o**, **carre.o** et **cube.o** ; car on ne peut passer à l'étape d'édition de liens qu'après avoir compilé chacun des fichiers objets.
- Les fichiers objets doivent être recompilés à chaque fois que le fichier source correspondant est modifié. Ce qui fait que le fichier source doit apparaître dans les dépendances du fichier objet correspondant.
- Une modification de **carre.h** doit entraîner la recompilation des deux fichiers objets **carre.o** et **main.o** car les deux fichiers source incluent ce fichier d'entête. De même, une modification de **cube.h** doit entraîner la recompilation des deux fichiers objets **cube.o** et **main.o**.

- La règle **clean** ne possède aucune dépendance. Seules des actions sont définies. En exécutant

```
$ make clean
rm *.o power
$
```

les **fichiers objets** et l'**exécutable**, produits lors de la construction de l'application avec la commande make, seront **supprimés**.

5 Exécution d'un Makefile

Par **convention**, le fichier exécuté par make est **nommé** `makefile` ou `Makefile`. Dans ce cas, l'exécution de la commande :

```
$ make
gcc –Wall -c main.c
gcc –Wall -c carre.c
gcc –Wall -c cube.c
gcc main.o carre.o cube.o -o power
$
```

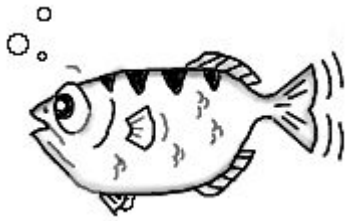
permet de produire la **première** cible trouvée dans le fichier Makefile. Si on veut **construire** une **cible particulière**, il suffit de lancer make avec le nom de la cible. Par exemple, la commande :

```
$ make carre.o
```

permet de construire la cible **carre.o** du fichier Makefile.

Si on donne au fichier exécuté par make un **nom différent** de Makefile, il suffit d'utiliser l'option **-f** de cette commande afin de spécifier le nom du fichier. Par exemple, si notre fichier s'appelle **projet**, alors la commande à exécuter pour donner l'ordre à make de traiter ce fichier est :

```
$ make -f projet
```



GDB

1 Introduction

GDB, pour **G**NU **D**e**B**ugger, est un programme permettant d'inspecter l'exécution d'autres programmes. Ceci est très utile lorsque vous avez un programme qui ne fonctionne pas correctement pour comprendre d'où vient l'erreur. GDB est portable, il fonctionne donc aussi bien sur UNIX/Linux que sur Windows ou sur MacOS.

Vous pouvez utiliser GDB pour exécuter votre code **pas à pas**, poser des points d'arrêt et examiner les valeurs des variables locales.

2 Lancer GDB

Pour utiliser GDB, vous devez compiler en activant les informations de débogage. Pour cela, utilisez l'option **-g** de **gcc** à la compilation.

Si vous utilisez un Makefile comme nous l'avons expliqué plus haut, vous pouvez vous contenter de rajouter l'option **-g** à la liste des options de compilation contenues dans la macro **CFLAGS**, comme ceci :

```
#*****************************    Makefile    ****************************#
      ...
      CFLAGS=-g -Wall
      ...
```

```
$ make
gcc -g -Wall -c main.c
gcc -g -Wall -c carre.c
gcc -g -Wall -c cube.c
gcc main.o carre.o cube.o -o power
$
```

Une fois l'exécutable est créé, il pourra être chargé dans GDB à l'aide de la commande **gdb Nom_Prog**.

```
$ gdb ./power
GNU gdb Red Hat Linux (6.6-35.fc8rh)
Copyright (C) 2006 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.
...
```

Lorsque GDB est lancé, il affiche le prompt :

```
(gdb)
```

3 Principales commandes

Une fois que vous avez lancé GDB, vous devez entrer des commandes pour lui indiquer quoi faire. Une ligne de commande GDB commence par le nom de la commande, qui peut

être suivi par des paramètres. Si vous **validez** une **ligne blanche**, GDB **répète la commande précédente**.

Les commandes de gdb possèdent une **notation courte** (**abréviation**) et une **notation longue** (**nom complet**). Pour exécuter une commande, vous pouvez utiliser l'abréviation aussi bien que le nom complet de la commande.

3.1 Aide

A. help

La commande **help** (ou **h**) permet d'obtenir de l'aide sur gdb.

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
...
```

La commande help peut être suivie du nom d'une commande de gdb afin d'afficher l'aide sur cette commande.

```
(gdb) h run

Start debugged program. You may specify arguments to give it.
Args may include "*", or "[...]"; they are expanded using "sh".
Input and output redirection with ">", "<", or ">>" are also allowed.
With no arguments, uses arguments last specified (with "run" or "set
args").
To cancel previous arguments and run with no arguments, use "set args"
without arguments.
(gdb)
```

3.2 Points d'arrêt

Un **point d'arrêt** (**breakpoint**) est un endroit où l'on interrompt temporairement l'exécution du programme afin d'examiner les valeurs des variables à cet endroit.

A. break

La commande **break** (ou **b**) permet de poser un point d'arrêt. 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.

```
(gdb) break main
Breakpoint 1 at 0x80483d5: file main.c, line 8.

(gdb) b carre.c:carre
Breakpoint 2 at 0x804844e: file carre.c, line 6.
```

- **ligne donnée du fichier source**

  Le point d'arrêt est alors défini par le numéro de la ligne correspondant.

```
(gdb) b 5
Note: breakpoint 1 also set at pc 0x80483d5.
Breakpoint 3 at 0x80483d5: file main.c, line 5.
(gdb) break cube.c:6
Breakpoint 4 at 0x8048466: file cube.c, line 6.
```

B. info break

La commande **info break** (ou **info b**) permet d'afficher la liste des points d'arrêt existant.

```
(gdb) info break
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x080483d5 in main at main.c:8
2   breakpoint     keep y   0x0804844e in carre at carre.c:6
3   breakpoint     keep y   0x080483d5 in main at main.c:5
4   breakpoint     keep y   0x08048466 in cube at cube.c:6
```

C. delete

La commande **delete** (ou **d**) permet de supprimer un point d'arrêt dont on donne le numéro.

```
(gdb) delete 1
```

Si on réaffiche la liste des points d'arrêt définis, on voit clairement que le point d'arrêt de numéro 1 ne figure plus dans la liste.

```
(gdb) info break
Num Type           Disp Enb Address    What
2   breakpoint     keep y   0x0804844e in carre at carre.c:6
3   breakpoint     keep y   0x080483d5 in main at main.c:5
4   breakpoint     keep y   0x08048466 in cube at cube.c:6
```

L'exécution de la commande delete **sans arguments** permet de supprimer tous les points d'arrêt.

```
(gdb) delete
Delete all breakpoints? (y or n) y
```

Si on affiche maintenant la liste des points d'arrêt, gdb va nous informer que cette liste est vide.

```
(gdb) info break
No breakpoints or watchpoints.
```

3.3 Exécution du programme

A. run

La commande **run** (ou **r**) permet d'exécuter un programme sous gdb. Le programme s'exécute alors normalement, jusqu'à ce qu'il se termine, ou lorsqu'il rencontre une erreur ou un point d'arrêt.

```
(gdb) r [args]
```

où **args** sont, s'il y en a, les arguments du programme à exécuter.

Pour notre exemple, si aucun point d'arrêt n'est défini, la commande run exécute le programme jusqu'à sa terminaison :

```
(gdb) r
```

```
...
Le carré de 1 est égal à 1
Le cube de 1 est égal à 1
Le carré de 2 est égal à 4
Le cube de 2 est égal à 8
...
Program exited normally.
(gdb)
```

En définissant un point d'arrêt sur la ligne 8 du fichier principal (i.e. main.c), l'exécution de la commande run exécutera le programme jusqu'à la rencontre de ce point d'arrêt :

```
(gdb) break 8
Breakpoint 1 at 0x80483d5: file main.c, line 8.
(gdb) run
...
Breakpoint 1, main () at main.c:8
8               for (i=1; i<=10; i++) {
(gdb)
```

B. continue

La commande **continue** (ou **c**) relance l'exécution du programme arrêtée par un point d'arrêt. Le programme poursuit son exécution jusqu'au prochain point d'arrêt ou jusqu'à à la fin ou la rencontre d'une erreur.

```
(gdb) c
Continuing.
Le carré de 1 est égal à 1
Le cube de 1 est égal à 1
Le carré de 2 est égal à 4
...
Program exited normally.
(gdb)
```

C. next

La commande **next** (ou **n**) avance l'exécution du programme d'une instruction, mais sans entrer dans le code des sous-programmes.

```
(gdb) n
9     resultat = carre(i);
(gdb)
10    printf("Le carré de %d est égal à %d\n", i, resultat);
(gdb)
Le carré de 1 est égal à 1
11    resultat = cube(i);
(gdb)
12    printf("Le cube de %d est égal à %d\n", i, resultat);
(gdb)
Le cube de 1 est égal à 1
8     for (i=1; i<=10; i++) {
(gdb)
```

D. step

La commande **step** (ou **s**) avance l'exécution du programme d'une ou plusieurs instructions, mais en entrant dans le code des sous programmes.

```
(gdb) s
9     resultat = carre(i);
(gdb)
carre (nb=2) at carre.c:6
6     res = nb * nb;
(gdb)
8     return res;
(gdb)
9     }
(gdb)
main () at main.c:10
10    printf("Le carré de %d est égal à %d\n", i, resultat);
(gdb)
Le carré de 2 est égal à 4
11    resultat = cube(i);
(gdb)
cube (nb=2) at cube.c:6
6     res = nb * nb * nb;
(gdb)
8     return res;
(gdb)
9     }
(gdb)
main () at main.c:12
12    printf("Le cube de %d est égal à %d\n", i, resultat);
(gdb)
Le cube de 2 est égal à 8
8     for (i=1; i<=10; i++) {
```

Contrairement à la commande next, on voit dans le listing ci-dessus que la commande **step** accède au code des fonctions **`carre`** et **`cube`**.

3.4 Inspection du programme

A. list

La commande **list** (ou **l**) permet d'afficher le code actuellement exécuté.

```
(gdb) list
3
4  int main(int argc, char* argv[]) {
5    int resultat;
6    int i;
7
8    for (i=1; i<=10; i++) {
9      resultat = carre(i);
10     printf("Le carré de %d est égal à %d\n", i, resultat);
11     resultat = cube(i);
12     printf("Le cube de %d est égal à %d\n", i, resultat);
(gdb)
```

B. print

La commande **print** (ou **p**) permet d'afficher la valeur d'une variable ou d'une expression lorsque l'exécution du programme est interrompue.

```
(gdb) p resultat
```

La commande ci-dessus permet d'afficher la valeur de la variable **`resultat`**.

C. display

La commande **display** est similaire à **print** sauf que l'affichage de la valeur de la variable ou l'expression se refait après chaque **next** ou **step**.

```
(gdb) display resultat
1: resultat = 1
(gdb) n
Le carré de 1 est égal à 1
11    resultat = cube(i);
1: resultat = 1
(gdb)
```

D. watch

La commande **watch** (ou **wa**) permet de surveiller la variable donnée en argument, en introduisant des **watchpoints**. Ces **watchpoints** ont pour effet d'interrompre l'exécution du programme chaque fois que la valeur de la variable est modifiée.

```
(gdb) watch resultat
Hardware watchpoint 2: resultat
(gdb) n
9     resultat = carre(i);
(gdb)
Hardware watchpoint 2: resultat
Old value = 13946868
New value = 1
main () at main.c:10
10    printf("Le carré de %d est égal à %d\n", i, resultat);
(gdb)
Le carré de 1 est égal à 1
11    resultat = cube(i);
(gdb)
12    printf("Le cube de %d est égal à %d\n", i, resultat);
(gdb)
Le cube de 1 est égal à 1
8     for (i=1; i<=10; i++) {
(gdb)
9     resultat = carre(i);
(gdb)
Hardware watchpoint 2: resultat
Old value = 1
New value = 4
main () at main.c:10
10    printf("Le carré de %d est égal à %d\n", i, resultat);
```

On remarque sur le listing ci-dessus que **contrairement** à **display**, la valeur de la variable **`resultat`** n'est réaffichée que si elle a été modifiée.

La commande **`info break`** permet d'afficher, en plus des points d'arrêt, la liste des watchpoints définis.

```
(gdb) info break
...
2   hw watchpoint  keep y              resultat
...
(gdb)
```

E. backtrace / where

La commande **backtrace** (ou **bt**) permet d'afficher la pile d'exécution, indiquant à quel endroit l'on se trouve au sein des différents appels de fonctions.

```
(gdb) bt
#0  main () at main.c:11
(gdb) s
cube (nb=2) at cube.c:6
6               res = nb * nb * nb;
(gdb) bt
#0  cube (nb=2) at cube.c:6
#1  0x08048411 in main () at main.c:11
(gdb)
```

L'exemple ci-dessus montre qu'on est dans la fonction cube, qui a été appelée par la fonction main.

Elle permet aussi d'afficher les variables locales si l'argument **full** est donné.

```
(gdb) bt full
#0  cube (nb=2) at cube.c:6
        res = 13946868
#1  0x08048411 in main () at main.c:11
        resultat = 4
        i = 2
(gdb)
```

3.5 Quitter GDB

A. quit

On peut quitter GDB en saisissant la commande **quit** (ou **q**).

```
(gdb) quit
$
```

BIBLIOGRAPHIE

[1] B. J. Gough, and R.M. Stallman. **An Introduction to GCC**. Network Theory Ltd. 2004.

[2] **GNU Compiler Collection**. http://gcc.gnu.org/

[3] **GNU Make**. http://www.gnu.org/software/make/

[4] **GNU Project Debugger**. http://www.gnu.org/software/gdb/

[5] M. Mitchell, J. Oldham, and A. Samuel. **Advanced Linux Programming**. ISBN: 0-7357-1043-0, Copyright © 2001 by New Riders Publishing.

http://docs.linux.cz/programming/other/ALP/

La version traduite en français de ce livre, par Michaël Todorovic, se trouve sous le titre **"Programmation Système Avancée sous Linux"**

Veuillez me communiquer les éventuelles erreurs que vous constatez dans le manuel