Outils de programmation

par Cuvelier François

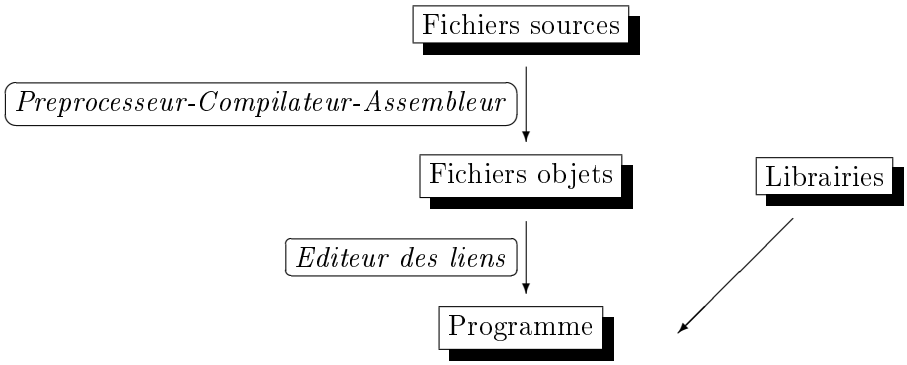
Université Paris Nord - Institut Galilée - LAGA
Av. J.-B. Clément 93430 Villetaneuse
email : cuvelier@math.univ-paris13.fr

Table des matières

Chapitre 1

GCC : compilateur

Le compilateur GNU C permet d'une part, de compiler des programmes écrits en C et C++, d'autre part de créer un programme exécutable à partir d'un fichier source principal. Il effectue pour cela successivement une phase de preprocessing, une phase de compilation, une phase d'assemblage et, enfin une phase d'édition des liens. Les 3 premières phases génerent des fichiers objets (.o) et la dernière un fichier exécutable.



Il est possible d'arreter le processus a une étape intermédiaire grâce à des options (dites *overall*). Par exemple l'option **-c** permet de stopper le processus juste avant l'édition des liens. Nous obtenons alors un fichier objet (.o) créé par la phase d'assemblage. D'autres options permettent de contrôler les différentes phases.

1.1 Exemple simple

Voici le programme que nous voulons exécuter :

```
1 #include <stdio.h> int main() {
2   printf("Hello World!!!\n");
3 }
```

Ce programme est sauvegardé dans le fichier *hello.c*. Pour créer un fichier exécutable, le plus simple consiste en la commande :

Session Shell

```
$ gcc hello.c
```

Le symbole $ désigne le prompt système. Cette commande effectue l'ensemble des phases de la compilation et crée le fichier exécutable *a.out*.

Avec l'option **-o** nous pouvons spécifier le nom du fichier exécutable :

Session Shell

```
$ gcc -o hello hello.c
```

Ici le programme exécutable est *hello*.

Une autre possibilité, utile lors de compilation séparée, est de passer par l'intermédiaire de fichiers objets (*.o) :

Session Shell

```
$ gcc -c hello.c
$ gcc -o hello hello.o
```

1.2 Compilation séparée

Lorsque l'on écrit des programmes plus complexes, il est impératif de partitionner le programme source en plusieurs unités.

Par exemple, pour écrire un programme permettant de résoudre un système linéaire par la méthode de Gauss, nous pouvons le décomposer de la manière suivante :

- *matrice.h matrice.c* : définition et fonctions pour les matrices.
- *vecteur.h vecteur.c* : définition et fonctions pour les vecteurs.
- *gauss.h gauss.c* : implémentation de la méthode de Gauss
- *prg2.c prg2.h* : programme principal

Dans ce cas pour créer le programme exécutable, il faut invoquer GCC de la manière suivante :

Session Shell

```
$ gcc -c matrice.c
$ gcc -c vecteur.c
$ gcc -c gauss.c
$ gcc -c prg2.c
$ gcc -o prg2 matrice.o vecteur.o gauss.o prg2.o
```

Les 4 premières lignes correspondent à la création des fichiers objets : preprocessing-compilation-assemblage. La dernière correspond à la création du fichier exécutable (*prg2*) à partir des fichiers objets (*matrice.o vecteur.o gauss.o prg2.o*) : édition des liens.

Cette façon de procéder par unités permet de

- gagner du temps à la compilation : si nous avons juste modifier le fichier *gauss.c* depuis la dernière compilation il suffit de créer le fichier objet correspondant puis d'éditer les liens pour créer le nouvel exécutable :

Session Shell

```
$ gcc -c gauss.c
$ gcc -o prg2 *.o
```

- réutiliser facilement une ou plusieurs unitées dans d'autres programmes
- tester individuellement chacune des unitées.

L'utilitaire **make** permet, entre autres, de gérer de manière très efficace la création d'un programme et d'éviter à l'utilisateur la saisie de commandes répétitives. Il est présenté au chapitre 3.

1.3 Utilisation d'une librairie

Une librairie est un ensemble cohérent de fonctions et sous-programmes, parfaitement validés et testés, regroupés au sein d'une même archive (extension *.a*) à l'aide de la commande **ar** (voir création d'une librairie). Les librairies peuvent être utilisées lors de l'édition des liens pour créer un programme exécutable.

Pour utiliser une librairie il faut, lors de l'édition des liens, indiquer en argument de la commande GCC le nom de la librairie avec l'option **-l** et le répertoire où elle se trouve avec l'option **-L**. Par exemple pour utiliser la librairie *libtest.a* du répertoire */usr/src/test* avec le programme *prg3.c* il faut lors de l'édition des liens exécuter la commande

Session Shell

```
$ gcc -o prg3 prg3.o -L/usr/src/test -ltest
```

Remarque 1 *On écrit* ***-ltest*** *: omission du prefixe lib.*

1.4 Options de GCC

Nous présentons ici que quelques options de GCC. Pour plus de renseignements se reporter à la documentation fournie avec le compilateur.

1.4.1 Options de contrôle du type de sortie(Output)

-c Compile les fichiers sources mais n'effectue pas l'édition des liens. Créer en sortie les fichiers objets (**.o*) correspondants.

-o *file* Place le résultat de la sortie dans le fichier *file*. Ceci s'applique pour tous types de fichiers en sortie (exécutable, objet)

-v Affiche les commandes exécutées pour lancer les différents processus de compilation.

1.4.2 Options de l'éditeur des liens

-l*nom* Utilise la librairie *libnom.a* lors de l'édition des liens.

1.4.3 Options de recherche de répertoire

-I*dir* Ajoute le répertoire *dir* en tête de liste pour la recherche des fichiers *header* (*.h)

-L*dir* Ajoute le répertoire *dir* à la liste des répertoires pour rechercher les librairies.

1.4.4 Options du preprocesseur

-D *macro valeur*

1.4.5 Options de debogage

-g Permet d'utiliser un logiciel de debogage (**gdb**, ...) à partir de l'exécutable du programme compilé.

1.4.6 Options d'optimisation

-O1 à -O3 permet d'optimiser le programme exécutable : l'optimisation est d'autant plus élevée que le nombre est grand.
Par défaut : ? ?

Chapitre 2

AR : création et gestion des librairies

Une librairie est un ensemble cohérent de fonctions et sous-programmes, parfaitement validés et testés, regroupés au sein d'une même archive (extension *.a*) à l'aide de la commande **ar**

2.1 Exemple simple

2.1.1 Listings

Voici 4 fichiers C :

Listing 2.1 – prg4.c

```c
/* Fichier prog4.c */
 #include <stdio.h>

void f1();
void f2();
void f3();

int main()
{
    /* Appel de la fonction f1 du fichier fic1.c */
   (void) f1();
   /* Appel de la fonction f2 du fichier fic2.c */
   (void) f2();
   /* Appel de la fonction f3 du fichier fic3.c */
   (void) f3();
}
```

Listing 2.2 – fic1.c

```c
/*Fichier fic1.c */
#include <stdio.h>

void f1()
{
```

```
6     printf("Appel de f1\n");
7  }
```

Listing 2.3 – fic2.c

```
1  /* Fichier fic2.c */
2  #include <stdio.h>
3
4  void f2()
5  {
6     printf("Appel de f2\n");
7  }
```

Listing 2.4 – fic3.c

```
1  /* Fichier fic3.c */
2  #include <stdio.h>
3
4  void f3()
5  {
6     printf("Appel de f3\n");
7  }
```

2.1.2 Création d'une librairie

Le programme principal (fichier *prg4.c*) utilise successivement les fonctions f1, f2 et f3 (resp. fichiers *fic1.c*, *fic2.c* et *fic3.c*). Nous allons regrouper le contenu de ces trois fichiers au seins d'une même librairie (fichier *libsimple.a*). Pour cela, il faut tout d'abord créer les fichiers objets :

Session Shell

```
$ gcc -c fic?.c
```

puis créer la librairie avec l'option **r**

Session Shell

```
$ ar r libsimple.a fic?.o
```

Pour obtenir la liste des membres (option **r**) :

Session Shell

```
$ ar t libsimple.a
     fic1.o
     fic2.o
     fic3.o
```

Remarque 2 *Lors de la création d'une librairie, il est fortement recommandé de créer un fichier header (.h) contenant les prototypes des différentes fonctions, les structures de donnés (type, struct, ...). L'ensemble doit être correctement documenté.*

Remarque 3 *Pour accélérer l'édition des liens sur certains systèmes, il est utile d'indexer la librairie créée. Ceci s'effectue par la commande* ***ranlib*** *:*

Session Shell

```
$ ranlib libsimple.a
```

B'in oui mais c'est pou'quoi faire une lib' ? : L'avantage d'une librairie est qu'elle est directement utilisable. C'est à dire que vous n'avez pas besoin des sources pour vous servir des fonctions qu'elle contient. Si vous voulez partager votre travail avec d'autres, un simple coup d'oeil au fichier hearder associé à votre librairie leurs donnera les indications nécessaires pour l'utiliser.

Le fichier *simple.h* associé à la librairie *libsimple.a* serait :

Listing 2.5 – simple.h

```
1 /* Fichier simple.h associe a la librairie libsimple.a */
2 #ifndef _SIMPLE_H
3 #define _SIMPLE_H
4
5 void f1(); /* Affiche "Appel de f1" */
6 void f2(); /* Affiche "Appel de f2" */
7 void f3(); /* Affiche "Appel de f3" */
8 #endif
```

Une modification du programme *prg4.c* permet de tenir compte de la remarque précédente :

Listing 2.6 – prg5.c

```
1  /* Fichier prog5.c */
2  #include <stdio.h>
3  #include "simple.h"
4
5  int main()
6  {
7     /* Appel de la fonction f1 (libsimple.a) */
8    (void) f1();
9     /* Appel de la fonction f2 (libsimple.a) */
10   (void) f2();
11    /* Appel de la fonction f3 (libsimple.a) */
12   (void) f3();
13 }
```

2.1.3 Utilisation d'une librairie

Si le programme *prg5.c* est compilé uniquement avec la commande suivante :

Session Shell

```
gcc -c prg5.c
```

lors de l'édition des liens apparait le message suivant :

Session Shell

```
$ gcc prg5.c -o prg5
     prg5.c: undefined reference to 'f1'
     prg5.c: undefined reference to 'f2'
     prg5.c: undefined reference to 'f3'
```

L'éditeur des liens ne trouve pas la "définition" des fonctions : il faut donc lui spécifier ou se trouve ces fonctions (fichiers objet ou fichiers librairie).

Session Shell

```
$ gcc prg5.c -L. -lsimple -o prg5
$ ./prg5
     Appel de f1
     Appel de f2
     Appel de f3
```

Attention : Sur certains systèmes, lors de l'édition des liens un certain ordre dans les options est à respecter suivant les dépendances entre les fichiers et les librairies. Dans notre exemple le programme *prg5.c* utilise des fonctions de la librairie *libsimple.a* (le fichier *prg5.o* dépend de la librairie *libsimple.a*). La syntaxe respecte la règle suivante : ce qui est écrit à gauche dépend de ce qui est écrit à droite. Voici ce qui se passe lorque cette règle n'est pas repectée :

Session Shell

```
$ gcc -L. -lsimple prg5.c -o simple
     xxx.o(.text+0x9):prg5.c: undefined reference to 'f1'
     xxx.o(.text+0xe):prg5.c: undefined reference to 'f2'
     xxx.o(.text+0x13):prg5.c: undefined reference to 'f3'
     collect2: ld returned 1 exit status
```

ou encore

Session Shell

```
$ gcc -c prg5.c
$ gcc -L. -lsimple prg5.o -o simple
     prg5.o(.text+0x9):prg5.c: undefined reference to 'f1'
     prg5.o(.text+0xe):prg5.c: undefined reference to 'f2'
     prg5.o(.text+0x13):prg5.c: undefined reference to 'f3'
     collect2: ld returned 1 exit status
```

Par contre les commande suivantes sont correctes :

Session Shell

```
$ gcc -c prg5.c
$ gcc prg5.o -L. -lsimple -o simple
```

ou encore

Session Shell

```
$ gcc prg5.c -L. -lsimple -o simple
$
```

2.2 Options

r : Création d'une librairie

t : Afficher le contenu de la librairie (fichiers objet)

d : Suppession de fichiers objets.

s : Permet d'indexer la librairie.

Chapitre 3

MAKE

Cet utilitaire permet, entre autre, de mettre à jour les versions d'exécutables, de fichiers objets ou de librairies dès qu'un fichier source a été modifié. Il présente un interêt évident dans le cas de programmes ou librairies qui sont composés d'un grand nombre d'éléments.

A chaque fois qu'un des fichiers composant le programme (ou la librairie) est modifié, la commande **make** crée le programme (ou la librairie) final(e) en recompilant uniquement les portions dépendantes du fichier modifié.

3.1 Principe de fonctionnement

L'utilitaire **make** utilise un fichier de description (généralement nommé *makefile*) contenant les cibles, les dépendances et les règles :

cibles : noms des modules (programme, librairie, fichier objet,...) qui vont être construit par **make**.

dépendances : ensemble des fichiers nécessaires pour construire une cible.

règles : ensemble des commandes de niveau shell qu'il faut utiliser pour construire les cibles, en fonctions des dépendances correspondantes.

La syntaxe générale de construction d'une cible est :

Listing 3.1 – cible

```
1 cible : dependances
2 ________Regles de production
```

Le symbole devant le mot *Regles* correspond à un **caractère de tabulation**.

Avant de donner un exemple de *makefile* voici des caractères spéciaux permettant de contrôler les affichages du *makefile* :

- Une ligne débutant par le caractère ♯ correspond à des commentaires.
- La commande `echo` permet d'afficher des messages lors du traitement des règles de production.
- Le caractère `@` placé juste après le caractère tabulation des règles de production empêche l'affichage de celle-ci (mais non leur exécution)

3.2 Exemple

Reprenons l'exemple du chapitre précédent où nous voulions construire le programme *prg4* à partir des fichiers *prg4.c*, *fic1.c*, *fic2.c* et *fic3.c*. Voici le diagramme des dépendances pour ce programme :

Fichier	Dépendances
prg4	prg4.o fic1.o fic2.o fic3.o
prg4.o	prg4.c
fic1.o	fic1.c
fic2.o	fic2.c
fic3.o	fic3.c

Un premier makefile peut être :

Listing 3.2 – Makefile version 1

```
1  # Mon premier makefile
2
3  prg4 : prg4.o fic1.o fic2.o fic3.o
4  ________gcc prg4.o fic?.o -o prg4
5
6  prg4.o : prg4.c
7  ________gcc -c prg4.c
8
9  fic1.o : fic1.c
10 ________gcc -c fic1.c
11
12 fic2.o : fic2.c
13 ________gcc -c fic2.c
14
15 fic3.o : fic3.c
16 ________gcc -c fic3.c
```

3.3 Variables définies par l'utilisateur

Leur syntaxe est la suivante :

Listing 3.3 – Exemple de variables utilisateur

```
1 NomDeLaVariable = Valeur
```

Pour l'utiliser : `$(NomDeLaVariable)`. Voici une modification du makefile précédent :

Listing 3.4 – Makefile version 2

```
1  # EXEMPLE N 2
2  #============
3  # Utilisation de variables definies par l'utilisateur
4  #
5
6  COMPILATEUR = gcc
7  OPTIONS     = -O
8  OBJETS      = prg4.o fic1.o fic2.o fic3.o
9  PROGRAMME   = prg4
10
11 $(PROGRAMME) : $(OBJETS)
12 ________$(CC) $(OPTIONS) $(OBJETS) -o $(PROGRAMME)
13
```

```
14 prg4.o : prg4.c
15 ________$(COMPILATEUR) $(OPTIONS) -c prg4.c
16
17 fic1.o : fic1.c
18 ________$(COMPILATEUR) $(OPTIONS) fic1.c
19
20 fic2.o : fic2.c
21 ________$(COMPILATEUR) $(OPTIONS) fic2.c
22
23 fic3.o : fic3.c
24 ________$(COMPILATEUR) $(OPTIONS) fic3.c
```

L'avantage des variables est de permettre des modifications rapides des makefiles. Par exemple si nous voulons utiliser l' option de compilation `-O3` en lieu et place de `-O`, il suffit de modifier la variable `OPTIONS = -O3`.

Il est aussi possible de remplacer une chaine (`Chaine1`) présente dans une variable par une autre (`Chaine2`) :

```
1 $(NomDeLaVariable:Chaine1=Chaine2)
```

Un exemple classique d'utilisation dans un makefile :

```
1 SOURCES = prg4.c fic1.c fic2.c fic3.c
2 OBJETS=$(SOURCES:.c=.o)
```

3.4 Règles internes de dérivation et variables internes

Lors de la compilation d'un programme utilisant de nombreuses sources, il est pesant d'écrire systèmatiquement les mêmes dépendances et les mêmes règles de production. C'est pourquoi l'utilitaire **make** contient une série de règles internes et de variables prédéfinies.

Pour obtenir l'ensemble des règles, il suffit d'exécuter la commande :

Session Shell

```
$ make -p -f - 2>/dev/null </dev/null >liste.txt
```

L'utilitaire **make** contient par exemple une règle permettant de créer un fichier objet à partir d'un fichier C : celle-ci est introduite, dans le fichier *liste.txt* créé ci-dessus, par la ligne suivante :
`.c.o`

Les règles de dérivation font appel à des variables internes :

3.4.1 Variables internes non modifiables

Macro	Signification
`$*`	nom du fichier, sans extension
`$<`	nom du fichier, avec extension
`$?`	Liste des dépendances plus récentes que la cible
`$@`	nom complet du fichier cible

3.4.2 Macros internes modifiables

Macro	Signification
CC	Compilateur C
CPP	Préprocesseur
CXX	Compilateur C++
F77	Compilateur fortran
LD	Editeur de liens
AR	Archivage (librairies)
CFLAGS	Options du compilateur C
CPPFLAGS	Options du préprocesseur
CXXFLAGS	Options du compilateur C++
F77FLAGS	Options du compilateur fortran
LDFLAGS	Options de l'éditeur des liens
COMPILE.c	$(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
LINK.c	$(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
COMPILE.cc	$(CXX) $(CXXFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
LINK.cc	$(CXX) $(CXXFLAGS) $(CPPFLAGS) $(LDFLAGS) $(TARGET_ARCH)
...	

Cette liste n'est pas exhaustive (voir le fichier *liste.txt* pour obtenir l'ensemble des macros). Pour les valeurs par défaut il suffit d'exécuter un makefile du type

Listing 3.5 – Makefile pour obtenir des valeurs par défaut

```
# FICHIER : makeprint
# ===================
# La commande
#
#    make -f makeprint print
#
# affiche les valeurs par defauts de
# variables internes
#

print :
	@echo ''Valeurs par defaut :''
	@echo ''CC = '' $(CC)
	@echo ''CXX = '' $(CXX)
	@echo ''F77 = '' $(F77)
	@echo ''AR = '' $(AR)
	@echo ''LD = '' $(LD)
	@echo ''RM = '' $(RM)
	@echo ''CFLAGS = '' $(CFLAGS)
	@echo ''CXXFLAGS = '' $(CXXFLAGS)
	@echo ''F77FLAGS = '' $(F77FLAGS)
	@echo ''ARFLAGS = '' $(ARFLAGS)
	@echo ''LDFLAGS = '' $(LDFLAGS)
)
```

Sur notre système, nous obtenons :

Session Shell

```
$ make -f makeprint print
Valeurs par defaut :
CC =  gcc
CXX=gcc
F77 =  f77
```

```
AR = ar
LD = ld
RM = rm -f
CFLAGS =
CXXFLAGS=
F77FLAGS =
ARFLAGS = rv
LDFLAGS =
```

3.4.3 Exemples

Voici un makefile utilisant des règles internes de dérivation :

Listing 3.6 – Makefile version 3

```
# EXEMPLE N 3
# ===========
# Utilisation des regles internes de derivation
#

OBJETS = prg4.o fic1.o fic2.o fic3.o
PROGRAMME = prg4

$(PROGRAMME) : $(OBJETS)
```

Voici une modification du makefile précédant redéfinissant une règle de dérivation :

Listing 3.7 – Makefile version 4

```
# EXEMPLE N 4
# ===========
# Redefinition de regles internes de derivation
#

COMPILATEUR = gcc
OPTIONS = -O
OBJETS = prg4.o fic1.o fic2.o fic3.o
PROGRAMME = prg4

$(PROGRAMME) : $(OBJETS)
	$(COMPILATEUR) $(OPTIONS) $(OBJETS) -o $(PROGRAMME)

.c.o :
	$(COMPILATEUR) $(OPTIONS) -c $<
```

Il est naturellement possible de modifier des variables internes.

Listing 3.8 – Makefile version 5

```
# EXEMPLE N 5
# ===========
# Modification des regles internes de derivation
#

CFLAGS = -O3
OBJETS = prg4.o fic1.o fic2.o fic3.o
PROGRAMME = prg4

$(PROGRAMME) : $(OBJETS)
```

3.5 Exemple avec dépendances croisées

Listing 3.9 – croises.c

```c
/* Fichier croises.c */
#include <stdio.h>
#include "c1.h"
#include "c2.h"
#include "c3.h"

int main()
{
    /* Appel de la fonction c1 du fichier c1.c */
   (void) c1("main",1);
   /* Appel de la fonction c2 du fichier c2.c */
   (void) c2("main",1);
   /* Appel de la fonction c3 du fichier c3.c */
   (void) c3("main",1);
   return 1;
}
```

Listing 3.10 – c1.c

```c
/*Fichier c1.c */
#include "c1.h"
#include "c2.h"

void c1(char *mess,int i)
{
   printf("fonction␣c1␣appelee␣par␣fonction␣%s\n",mess);
   if (i==1)
     (void) c2("c1",0);
}
```

Listing 3.11 – c1.h

```c
#ifndef _C1_H
#define _C1_H
#include <stdio.h>
void c1(char *,int );
#endif
```

Listing 3.12 – c2.c

```c
/*Fichier c2.c */
#include "c2.h"
#include "c1.h"

void c2(char *mess,int i)
{
```

```
7   printf("fonction␣c2␣appelee␣par␣fonction␣%s\n",mess);
8    if (i==1)
9      (void) c1("c2",0);
10 }
```

Listing 3.13 – c2.h

```
1 #ifndef _C2_H
2 #define _C2_H
3 #include <stdio.h>
4 void c2(char *,int );
5
6 #endif
```

Listing 3.14 – c3.c

```
1 /* Fichier fic3.c */
2 #include "c1.h"
3 #include "c2.h"
4 #include "c3.h"
5
6 void c3(char *mess, int i)
7 {
8    printf("fonction␣c3␣appelee␣par␣fonction␣%s\n",mess);
9    if (i==1)
10     {
11       (void) c1("c3",0);
12       (void) c2("c3",0);
13     }
14 }
```

Listing 3.15 – c3.h

```
1 #ifndef _C3_H
2 #define _C3_H
3 #include <stdio.h>
4 void c3(char *,int );
5 #endif
```

Fichier	Dépendances
croises	croises.o c1.o c2.o c3.o
croises.o	croises.c c1.h c2.h c3.h
c1.o	c1.c c1.h c2.h
c2.o	c2.c c2.h c1.h
c3.o	c3.c c3.h c2.h c3.h

Listing 3.16 – Makefile pour des dépendances croisées

```
1  # EXEMPLE N 6
2  #=============
3  # Dependances croisees
4  CC = gcc
5  CFLAGS = -g -Wall
6
7  croises : croises.o c1.o c2.o c3.o
8
9  c1.o : c1.h c2.h
10
11 c2.o : c2.h c1.h
12
13 c3.o : c1.h c2.h c3.h
14
15 croises.o : c1.h c2.h c3.h
16
17 clean :
18 ________rm -f croises *.o *~ *.bak
```

3.6 Options

-f *filename* Utilise le fichier *filename* comme fichier de description.

-p Liste des règles de dérivation internes et des dépendances prédéfinies.

-s Exécute mais n'affiche pas les commandes des règles de dérivations.

-n Affiche mais n'exécute pas les commandes des règles de dérivations.

Chapitre 4

GDB : débuggeur

Un débuggeur comme GDB permet de suivre le déroulement d'un programme et d'accéder au contenu des variables en cours d'exécution afin d'analyser finement la situation.

En pratique pour débuguer un programme, il faut générer des informations de débuguage pendant la compilation de celui-ci. Nous utilisons pour celà l'option `-g` du compilateur. Les informations de débuguage sont alors stockées dans les fichiers objets.

4.1 Liste de commandes

4.1.1 Arréter l'exécution

break : Un point d'arrêt (*breakpoint*) permet de stopper un programme à l'endroit où il a été *posé*.

`break` $FUNCTION$ Pose un *breakpoint* à l'entrée de la fonction `FUNCTION`.

`break` $LINENUM$ Pose un *breakpoint* à la ligne `LINENUM` dans le fichier source courant.

`break` $FILENAME : LINENUM$ Pose un *breakpoint* à la ligne `LINENUM` dans le fichier source `FILENAME`.

`break` $FILENAME : FUNCTION$ Pose un *breakpoint* à l'entrée de la fonction `FUNCTION` dans le fichier source `FILENAME`.

`break ... if COND`

watch : Un *watchpoint* peut être utilisé pour areter l'exécution d'un programme dès qu'une expression est modifiée.

`watch EXPR :` Pose un *watchpoint* pour une expression. GDB s'arretera dès que `EXPR` est modifiée.

info : Affiche la liste de tous les *breakpoint* et *watchpoint*.

clear : Efface les *breakpoints*

`clear` $FUNCTION$ **:** Efface les *breakpoints* de la fonction `FUNCTION` du fichier courant.

`clear` $FILENAME : FUNCTION$ **:** Efface les *breakpoints* de la fonction `FUNCTION` du fichier `FILENAME`.

`clear` *LINENUM* : Efface les *breakpoints* définis en ligne `LINENUM` du fichier courant.

`clear` *FILENAME* : *LINENUM* : Efface les *breakpoints* définis en ligne `LINENUM` du fichier `FILENAME`.

delete `[`***breakpoints***`][`***BNUMS...***`]` : Efface les *breakpoints* ou les *watchpoints*

4.1.2 Contrôle de l'exécution

run : Utiliser la commande `run` pour démarrer votre programme sous GDB.

step `[COUNT]` (***raccouci*** **s) :** La commande `step` continue l'exécution du programme jusqu'à rencontrer une ligne de code différente. Avec l'option `[COUNT]` le programme est stoppé après avoir rencontré `[COUNT]` lignes de code et ceci même si des *breakpoints* ont été rencontrés.

next `[COUNT]` (***raccouci*** **n) :** La commande `next` continue le programme jusqu'à la ligne suivante dans le même fichier source. Avec l'option `[COUNT]`, le programme est stoppé après avoir rencontré `[COUNT]` lignes dans le même fichier source.

continue `[IGNORE-COUNT]` (***raccouci*** **c) :**

finish : Arrête l'exécution juste après la fin de la fonction courante. Affiche la valeur retournée par cette fonction (s'il y a lieu).

4.1.3 Examen des données et du code source

print `EXP[=VALUE]` (***raccouci*** **p) :** Affiche (et modifie avec `=VALUE`) le contenu de la variable `EXP`

display `[EXP]` (***raccouci*** **disp) :**

list (***raccouci*** **l) :** Par défaut, affiche 10 lignes d'un fichier source.

`list` *LINENUM* : Affiche les lignes centrées autour de la ligne `LINENUM` du fichier source courant.

`list` *FUNCTION* : Affiche les lignes centrées autour du début de la fonction `FUNCTION`.

`list` : Affiche plus de lignes par rapport aux dernières lignes affichées.

`list -` : Affiche les lignes précédant les dernières affichées.

4.2 Exemple d'utilisation

4.2.1 Listing

Listing 4.1 – test_tableau.c

```
1 /****************** test_tableau.c ********************/
2 /* Description : fichier main .                       */
3 /*                                                    */
4 /******************************************************/
5 /* Dependances : Tableau.h                            */
6 /******************************************************/
7 /* Variables globales : ...                           */
8 /* Fonctions externes utilisees :                     */
9 /*    EntreeTableau (fichier Tableau.c)               */
```

```
10 /*     AfficheTableau (fichier Tableau.c)                */
11 /*     DetruitTableau (fichier Tableau.c)                */
12 /*******************************************************/
13 /* Auteurs : Cuvelier F.                                */
14 /* Version : 1.0                                        */
15 /* Remarques : Exemple du polycopie <Outils de          */
16 /*      programmation>.                                 */
17 /*      Ne fonctionne pas.                              */
18 /*      A utiliser avec GDB.                            */
19 /* Dernieres modifications : 26 janvier 2000            */
20 /* Copyright :                                          */
21 /*         Institut Galilee - Univ. Paris 13            */
22 /*******************************************************/
23
24 #include "Tableau.h"
25
26 int main()
27 {
28   Tableau T;
29
30   EntreeTableau(T);
31   AfficheTableau(T,"1er Tableau");
32
33   T.val[1]=3.14;
34   AfficheTableau(T,"Modification");
35
36   EntreeTableau(T);
37   AfficheTableau(T,"2eme Tableau");
38   DetruitTableau(T);
39
40 }
```

Listing 4.2 – Tableau.c

```
1  /********************* main.c ***********************/
2  /* Description : ...                                */
3  /*                                                  */
4  /****************************************************/
5  /* Dependances : ...                                */
6  /****************************************************/
7  /* Variables globales : ...                         */
8  /* Fonctions externes utilisees : ...               */
9  /****************************************************/
10 /* Auteurs : Cuvelier F.                            */
11 /* Version : 1.0                                    */
12 /* Remarques : Exemple du polycopie <Outils de      */
13 /*      programmation>.                             */
14 /*      Ne fonctionne pas.                          */
15 /*      A utiliser avec GDB.                        */
16 /* Dernieres modifications : 26 janvier 2000        */
17 /* Copyright :                                      */
18 /*         Institut Galilee - Univ. Paris 13        */
19 /****************************************************/
20
21 /* Fichiers includes */
22 #include "Tableau.h"
23
24 void CreerTableau(Tableau T, int dim)
25 {
26   T.dim=dim;
27   T.val=(double *)malloc( dim*sizeof(double) );
28 }
29
30 void DetruitTableau(Tableau T)
31 {
32   T.dim=0;
33   free(T.val);
34 }
35
36 void EntreeTableau(Tableau T)
37 {
38   int dimension,i;
39   printf("Dimension du tableau : ");
40   scanf("%d\n",dimension);
41   CreerTableau(T,dimension);
42   for (i=1;i<T.dim;i++)
43     {
44       printf("Composante %d : ",i);
45       scanf("%g\n",&(T.val[i]));
46     }
47 }
48
```

```
49 void AfficheTableau(Tableau T,char *message)
50 {
51   int i;
52   printf("%c\n",message);
53   for (i=1;i<=T.dim;i++)
54     printf("val[%d]=%g\n",i,T.val[i]);
55 }
```

Listing 4.3 – Tableau.h

```
1  /****************** Tableau.h *********************/
2  /* Description :  Fichier header associe au fichier   */
3  /*     Tableau.c .                                    */
4  /**************************************************/
5  /* Dependances :                                      */
6  /**************************************************/
7  /* Variables globales :                               */
8  /* Fonctions externes utilisees :                     */
9  /**************************************************/
10 /* Auteurs : Cuvelier F.                              */
11 /* Version : 1.0                                      */
12 /* Remarques : Exemple du polycopie <Outils de        */
13 /*      programmation>.                               */
14 /*      Ne fonctionne pas.                            */
15 /*      A utiliser avec GDB.                          */
16 /* Dernieres modifications : 26 janvier 2000          */
17 /* Copyright :                                        */
18 /*         Institut Galilee - Univ. Paris 13          */
19 /**************************************************/
20
21 #ifndef _TEST_H
22 #define _TEST_H
23
24 /* Fichiers includes */
25 #include <stdio.h>
26
27 /* Declaration de types donnees */
28 typedef struct tab{
29   int dim;
30   double *val;
31 } Tableau;
32
33 /* Prototypage des fonctions */
34 void CreerTableau(Tableau T, int dim);
35 void DetruitTableau(Tableau T);
36 void AfficheTableau(Tableau T,char *message);
37 void EntreeTableau(Tableau T);
38
39 #endif
```

4.2.2 Compilation et exécution

Session Shell

```
$gcc -g   -c test_tableau.c -o test_tableau.o
$gcc -g   -c Tableau.c -o Tableau.o
$gcc   test_tableau.o Tableau.o   -o test_tableau
$./test_tableau
Dimension du tableau : 3
handle_exceptions: Exception: STATUS_ACCESS_VIOLATION
```

Le programme ne fonctionne pas ! Nous remarquons l'usage de l'option **-g** lors de la compilation : ceci nous permet d'utiliser le débuggeur.

4.2.3 Débuguage

Session GDB

```
$ gdb test_tableau
```

⇑ Utilisation de GDB

```
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-mandrake-linux"...
(gdb) l
```

⇑ Affiche le source centré sur la fonction *main* (**l** abrégé de **list**)

```
 21      /*    FSIMACS - Institut Galilee - Univ. Paris 13     */
22      /*****************************************************/
23
24      #include "Tableau.h"
25
26      int main()
27      {
28        Tableau T;
29
30        EntreeTableau(T);
(gdb) b main
```

⇑ Pose d'un breakpoint sur la fonction *main*

```
Breakpoint 1 at 0x8048466: file test_tableau.c, line 30.
```

⇑ Le 1er breakpoint est en ligne 30 du fichier *test_tableau.c*

```
(gdb) r
```

⇑ Lance l'exécution du programme (**r** abrégé de **run**)

```
Starting program: /home/cuvelier/gdbTableau/test_tableau

Breakpoint 1, main () at test_tableau.c:30
30        EntreeTableau(T);
```

⇑ Arrêt sur la ligne 30 de la fonction *main*

```
(gdb) p T
```

⇑ Affichage de la variable `T` (**p** abrégé de **print**)

```
$1 = {dim = 134513744, val = 0x80496fc}
```

```
(gdb) n
```

⇑ Continue l'exécution du programme jusqu'à l'instruction suivante dans le même bloc (**n** abrégé de **next**)

```
Dimension du tableau : 3
```

⇑ On entre la dimension du tableau : ici 3

```

Program received signal SIGSEGV, Segmentation fault.
0x2ab1b7bb in _IO_vfscanf () from /lib/libc.so.6
```

⇑ Le programme a généré une *Segmentation fault* à l'exécution de la ligne 30 : il va falloir regarder plus en détail cette ligne

```
(gdb) r
```

⇑ On lance à nouveau le programme

```
The program being debugged has been started already.
Start it from the beginning? (y or n) y
```

⇑ Confirmation de réinitialisation

```
Starting program: /home/cuvelier/gdbTableau/test_tableau

Breakpoint 1, main () at test_tableau.c:30
```

⇑ Rappel des breakpoints déjà posés

```
30          EntreeTableau(T);
(gdb) s
```

⇑ Passe à l'instruction suivante même si elle n'est pas dans le même bloc (**s** abrégé de **step**)

```

EntreeTableau (Tab={dim = 134513744, val = 0x80496fc}) at Tableau.c:39
```

⇑ Affichage automatique des paramètres passés à la fonction `EntreTableau` définie dans le fichier *Tableau.c*

```
39          printf("Dimension du tableau : ");
```

⇑ Arrêt sur la ligne 39 du fichier *Tableau.c*

```
(gdb) n
```

⇑ Passage à l'instruction suivante dans le même bloc

```
40          scanf("%d\n",dimension);
```

⇑ Arrêt à la ligne 40

```
(gdb) n
Dimension du tableau : 3
```

```
66
67  Program received signal SIGSEGV, Segmentation fault.
68  0x2ab1b7bb in _IO_vfscanf () from /lib/libc.so.6
```

⇑ De nouveau le message maudit!!!
L'erreur provient donc de la ligne 40 du fichier *Tableau.c* : `scanf("%d\n",dimension)`. Cette fonction étant une fonction prédéfinie l'erreur provient donc d'une mauvaise utilisation de celle-ci.

```
69
70  (gdb)
```

Si l'erreur(s) ne vous apparait pas clairement, il faut consulter une doc. Comme la variable `dimension` doit être modifiée, il faut la passer par adresse (et non par valeur) ce qui l'équivalent du `var` en langage Pascal : `scanf("%d\n",&dimension)`.

Après modification, il faut compiler le programme puis l'exécuter :

Make et exécution de test_tableau

```
1   $ make
```

⇑ Compilation

```
2
3   cc -g   -c Tableau.c -o Tableau.o
4   cc   test_tableau.o Tableau.o   -o test_tableau
5   $ test_tableau
```

⇑ Exécution du programme

```
6
7   Dimension du tableau : 3
8
9
10  ^C
```

⇑ Après avoir entré la dimension du tableau (ici 3), des *return* successifs semblent inopérants... Un *Ctrl-C* permet de quitter le programme. Une nouvelle fois c'est la fonction EntreTableau qui est en cause.

```
11
```

Il va falloir de nouveau débuguer :

Session GDB

```
1   (gdb) r
2   The program being debugged has been started already.
3   Start it from the beginning? (y or n) y
4
5   Starting program: /home/cuvelier/dbgTableau/test_tableau
6
7   Breakpoint 3, main () at test_tableau.c:30
8   30          EntreeTableau(T);
```

⇑ A la commande run, le breakpoint déjà posé sur la fonction main est rappelé

```
(gdb) l Tableau.c:40
```

⇑ Affichage des 10 lignes centrées sur la ligne 40 du fichier *Tableau.c*

```
35
36      void EntreeTableau(Tableau Tab)
37      {
38        int dimension,i;
39        printf("Dimension du tableau : ");
40        scanf("%d\n",&dimension);
41        CreerTableau(Tab,dimension);
42        for (i=1;i<Tab.dim;i++)
43          {
44            printf("Composante %d : ",i);
(gdb) b Tableau.c:40
```

⇑ Pose d'un breakpoint en ligne 40 du fichier *Tableau.c*

```
Breakpoint 2 at 0x8048553: file Tableau.c, line 40.
(gdb) i b
```

⇑ Information sur les breakpoints (**i b** abrégé de **info breakpoint**)

```
Num Type           Disp Enb Address    What
1   breakpoint     keep y   0x08048466 in main at test_tableau.c:30
        breakpoint already hit 1 time
2   breakpoint     keep y   0x08048553 in EntreeTableau at Tableau.c:40
(gdb) c
```

⇑ Poursuite de l'exécution du programme jusqu'au prochain breakpoint, watchpoint, ou jusqu'à la 1ère erreur ou (in a perfect world) la fin du programme.

```
Continuing.

Breakpoint 2, EntreeTableau (Tab={dim = 134513744, val = 0x80496fc})
    at Tableau.c:40
```

⇑ Le 2ème breakpoint à été rencontré d'où arrêt

```
40        scanf("%d\n",&dimension);
(gdb) n
Dimension du tableau : 3
```

```
43
44
45  Program received signal SIGINT, Interrupt.
46  0x2ab73bb4 in __libc_read () from /lib/libc.so.6
      ⇑ Après avoir entré la dimension du tableau, ça ne marche
      toujours pas !!!
47
```

L'erreur est toujours situé sur la ligne 40 car après la commande **n** (next), le débuggeur aurait du stopper à l'instruction suivante (dans le même bloc) qui est en ligne 41. Il faut donc de nouveau se plonger dans la doc. L'erreur ici porte sur le format de la donnée à entrer. Le `"%d\n"` n'est pas un format valide : le \n pose problème. En fait, le passage à la ligne est automatique lors de l'utilisation de la fonction `scanf`. Il faut donc écrire `scanf("%d",&dimension)`. On remarque la même erreur en ligne 46 dans le même fichier.

Après modification, on compile. On obtient à l'exécution :

Exécution de test_tableau

```
1  $ test_tableau
2  Dimension du tableau : 3
3  Composante 1 : 1
4  Composante 2 : 2
5  Composante 3 : 3
6  Composante 4 : 4
7  Composante 5 : 5
8  Erreur de segmentation
```

Celà ne marche toujours pas : retour au débuggeur :

Session GDB

```
1  (gdb) r
2  The program being debugged has been started already.
3  Start it from the beginning? (y or n) y
4  Starting program: /home/cuvelier/dbgTableau/test_tableau
5
6  Breakpoint 1, main () at test_tableau.c:30
7  30        EntreeTableau(T);
8  (gdb) c
9  Continuing.
10
11 Breakpoint 2, EntreeTableau (Tab={dim = 134513744, val = 0x80496fc}) at Tableau.c:40
12 40        scanf("%d",&dimension);
     ⇑ Arrêt sur le 2ème breakpoint
13
14 (gdb) n
15 Dimension du tableau : 3
16 41        CreerTableau(Tab,dimension);
```

```
(gdb) p dimension
```

⇑ Affichage de la variable `dimension` (abrégé de print)

```
$1 = 3
(gdb) n
```

⇑ Continue jusqu'à la prochaine instruction du même bloc

```
42          for (i=1;i<Tab.dim;i++)
(gdb) p Tab
```

⇑ Affichage de la variable `Tab`

```
$2 = dim = 134513744, val = 0x80496fc
```

⇑ Après l'appel fonction `CreerTableau` nous aurions du avoir `Tab.dim=3` et `Tab.val` modifié!

```
(gdb)
```

L'erreur ici vient du fait qu'il faut passer la variable `Tab` par adresse dans la fonction `CreerTableau` puisqu'elle est modifiée intentionnellement.

Voici l'ensemble des modification à apporter :

Tableau.c

```
void CreerTableau(Tableau *Ta, int dimT)
{
  Ta->dim=dimT;
  Ta->val=(double *)malloc( dimT*sizeof(double) );
}
⋮
```

Tableau.c

```
    int dimension,i;
    printf("Dimension du tableau : ");
    scanf("%d",&dimension);
    CreerTableau(&Tab,dimension);
```

Tableau.h

```
void CreerTableau(Tableau *Ta, int dimT);
```

Une fois ces modifications apportées il faut compiler puis lancer le programme :

Session GDB

```
$ test_tableau
Dimension du tableau : 3
Composante 1 : 1
Composante 2 : 2
val[1]=nan
val[2]=4.87199e-270
```

```
val[3]=3.81707e-103
⋮
val[286]=0
val[287]=0
Erreur de segmentation
```

Et retour au débuggeur :

Session GDB

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/cuvelier/dbgTableau/test_tableau

Breakpoint 1, main () at test_tableau.c:30
30          EntreeTableau(T);
(gdb) c
Continuing.

Breakpoint 2, EntreeTableau (Tab={dim = 134513744, val = 0x80496fc})
    at Tableau.c:40
40          scanf("%d",&dimension);
(gdb) n
Dimension du tableau : 3
41          CreerTableau(&Tab,dimension);
(gdb) n
42          for (i=1;i<Tab.dim;i++)
(gdb) p Tab
```

⇑ Affichage de la variable `Tab` pour vérifier qu'elle a bien été modifiée par la fonction `CreerTableau`

```

$6 = {dim = 3, val = 0x8049990}
```

⇑ La dimension est bonne et l'adresse de val a été modifiée : c'est tout bon!

```

(gdb) p Tab.val[0]
$7 = 0
(gdb) p Tab.val[1]
$8 = 0
(gdb) p Tab.val[2]
$9 = 0
```

```
30
31 (gdb) n
32 44              printf("Composante %d : ",i);
33 (gdb) n
34 45              scanf("%g",&(Tab.val[i]));
35 (gdb) n
36 Composante 1 : 1
37 42          for (i=1;i<Tab.dim;i++)
38 (gdb) p Tab.val[1]
39 $10 = 5.26354424712089O3e-315
40
```

⇑ Vérification du contenu de `Tab.val` après la fonction `CreerTableau` : comme l'allocation mémoire a été effectuée avec la fonction `malloc`, le tableau `Tab.val` est initialisé à 0 (Que se passe-t-il si l'on utilise `calloc` a la place de `malloc`?)

⇑ La variable `Tab.val[1]` n'a pas été lue correctement : il y donc une erreur d'utilisation de la fonction `scanf`. On remarque aussi une erreur dans la boucle : pour initialiser les 3 composantes du tableau il faut que l'indice i prenne les valeurs 0,1 et 2 (`Tab.val[0]` étant la première composante du tableau).

En regardant la doc d'utilisation de la fonction `scanf`, on peut voir que le format n'est pas valable. En effet la variable `Tab.val[i]` est de type `double` : il faut donc utiliser le format "%lf" en lieu et place de "%g". Pour la boucle, il faut écrire `for (i=0;i<Tab.dim;i++)`.

Une fois les modifications effectuées, on recompile et on exécute :

Exécution de test_tableau

```
1  $ test_tableau
2  Dimension du tableau : 3
3  Composante 0 : 1
4  Composante 1 : 2
5  Composante 2 : 3
6  val[1]=nan
7  val[1]=nan
8  val[2]=4.87199e-270
9  val[3]=3.81707e-103
10 ⋮
11 val[286]=0
12 val[287]=0
13 Erreur de segmentation
```

Il faut de nouveau retourner sous le débuggeur :

Session GDB

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

'/home/cuvelier/dbgTableau/test_tableau' has changed; re-reading symbols.
Starting program: /home/cuvelier/dbgTableau/test_tableau

Breakpoint 1, main () at test_tableau.c:30
30        EntreeTableau(T);
(gdb) c
Continuing.

Breakpoint 2, EntreeTableau (Tab={dim = 134513744, val = 0x80496fc})
    at Tableau.c:40
40        scanf("%d",&dimension);
(gdb) n
Dimension du tableau : 3
41        CreerTableau(&Tab,dimension);
(gdb) n
42        for (i=0;i<Tab.dim;i++)
(gdb) l
37      {
38        int dimension,i;
39        printf("Dimension du tableau : ");
40        scanf("%d",&dimension);
41        CreerTableau(&Tab,dimension);
42        for (i=0;i<Tab.dim;i++)
43          {
44            printf("Composante %d : ",i);
45            scanf("%lf",ttt&(Tab.val[i]));
46          }
(gdb)
47      }
48
49      void AfficheTableau(Tableau TA,char *message)
50      {
51        int i;
52        printf("%c\n",message);
53        for (i=1;i<=TA.dim;i++)
54          printf("val[%d]=%g\n",i,TA.val[i]);
55      }
(gdb) b Tableau.c:47
Breakpoint 3 at 0x80485c5: file Tableau.c, line 47.
```

```
(gdb) c
Continuing.
Composante 0 : 1
Composante 1 : 2
Composante 2 : 3

Breakpoint 3, EntreeTableau (Tab={dim = 3, val = 0x8049990}) at Tableau.c:47
47      }
(gdb) p Tab.val[0]
$2 = 1
(gdb) p Tab.val[1]
$3 = 2
(gdb) p Tab.val[2]
$4 = 3
(gdb) n
main () at test_tableau.c:31
31        AfficheTableau(T,"1er Tableau");
(gdb) p T
$5 = {dim = 134513744, val = 0x80496fc}
```

⇑ Le tableau T n'a pas été initialisé

```
(gdb) l
26      int main()
27      {
28        Tableau T;
29
30        EntreeTableau(T);
31        AfficheTableau(T,"1er Tableau");
32
33        T.val[1]=3.14;
34        AfficheTableau(T,"Modification");
35
```

⇑ C'est en ligne 30, à l'appel de la fonction `EntreeTableau` que se situe
le problème : c'est une nouvelle fois un problème de passage de paramètres. Il faut passer
le paramètre `T` par adresse si l'on veut que les modifications soient prises en compte.

L'appel de la fonction EntreTableau doit être modifié, ce qui donne du travail. Voici l'ensemble des modification à apporter :

Tableau.c

```
void EntreeTableau(Tableau *pTab)
```

```
37  ⇑ Pour mieux comprendre, on a changé le nom du paramètre : pTab est un pointeur sur un tableau.
38  {
39    int dimension,i;
40    printf("Dimension du tableau : ");
41    scanf("%d",&dimension);
42    CreerTableau( pTab ,dimension);
43    for (i=0;i< pTab-> dim;i++)
44      {
45        printf("Composante %d : ",i);
46        scanf("%lf", pTab .val[i]));
47      }
48  }
```

Tableau.h

```
34  void EntreeTableau(Tableau *pTab );
```

test_tableau.c

```
26  int main()
27  {
28    Tableau T;
29
30    EntreeTableau( &T );
31    AfficheTableau(T,"1er Tableau");
32
33    T.val[1]=3.14;
34    AfficheTableau(T,"Modification");
35
36    EntreeTableau( &T );
37    AfficheTableau(T,"2eme Tableau");
38    DetruitTableau(T);
39
40  }
```

Une fois ces modifications apportées il faut compiler puis lancer le programme :

Exécution de test_tableau

```
1  $ test_tableau
2  Dimension du tableau : 3
3  Composante 0 : 1
4  Composante 1 : 2
5  Composante 2 : 3
6
```

```
val[1]=2
val[2]=3
val[3]=3.41429e-311

val[1]=3.14
val[2]=3
val[3]=3.41429e-311
```

⇑ Problème à l'affichage !!!

```
Dimension du tableau :          ^ C
```

De nouveau, le débuggeur :

Session GDB

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/cuvelier/dbgTableau/test_tableau

Breakpoint 1, main () at test_tableau.c:30
30          EntreeTableau(&T);
(gdb) c
Continuing.

Breakpoint 2, EntreeTableau (pTab=0x7ffff840) at Tableau.c:40
40          scanf("%d",&dimension);
(gdb) c
Continuing.
Dimension du tableau : 3
Composante 0 : 1
Composante 1 : 2
Composante 2 : 3

Breakpoint 3, EntreeTableau (pTab=0x7ffff840) at Tableau.c:47
47      }
(gdb) p *pTab
$1 = {dim = 3, val = 0x80499a0}
(gdb) p (*pTab).val[0]
$2 = 1
(gdb) p (*pTab).val[1]
$3 = 2
(gdb) p (*pTab).val[2]
$4 = 3
```

```
    ⇑ Vérification du contenu de la variable pTab
31
32  (gdb) n
33  main () at test_tableau.c:31
34  31          AfficheTableau(T,"1er Tableau");
    ⇑ Retour dans la fonction main du fichier test_ tableau.c
35
36  (gdb) p T
37  $6 = {dim = 3, val = 0x80499a0}
    ⇑ Vérification du contenu de la variable T. L'adresse de
    T.val
    est identique à l'adresse précédente de *pTab.val dans la
    fonction
    Entree Tableau. Donc tout s'est passé dans cette fonction .
38
39  (gdb) s
40  AfficheTableau (TA={dim = 3, val = 0x80499a0}, message=0x8048698 "1er Tableau")
41      at Tableau.c:52
42  52          printf("%c\n",message);
43  (gdb) n
44
    ⇑ Rien ne s'affiche : il y a un problème avec la fonction
    printf.
45
46  53          for (i=1;i<=TA.dim;i++)
```

Pour un affichage correct, il faut écrire `printf("%s\n",message);`. On compile et on exécute le programme :

Exécution de test_tableau

```
1   $ test_tableau
2   Dimension du tableau : 3
3   Composante 0 : 1
4   Composante 1 : 2
5   Composante 2 : 3
6   1er Tableau
7   val[1]=2
8   val[2]=3
9   val[3]=3.41429e-311
    ⇑ Erreur dans les indices : en C un tableau commence en 0!
10
11  Modification
12  val[1]=3.14
13  val[2]=3
14  val[3]=3.41429e-311
```

```
15 Dimension du tableau :                    ^ C
```

L'erreur ici est suffisament clair : la boucle d'affichage dans la fonction AfficheTableau du fichier Tableau.c n'est pas correcte, il faut commencer en 0 et finir en T.dim-1 (ici 3-1). La boucle doit être :

Tableau.c

```
49 void AfficheTableau(Tableau TA,char *message)
50 {
51   int i;
52   printf("%s\n",message);
53   for (i=0;i<TA.dim;i++)
54     printf("val[%d]=%g\n",i,TA.val[i]);
55 }
```

Exécution de test_tableau

```
1  $ test_tableau
2  Dimension du tableau : 3
3  Composante 0 : 1
4  Composante 1 : 2
5  Composante 2 : 3
6  1er Tableau
7  val[0]=1
8  val[1]=2
9  val[2]=3
10 Modification
11 val[0]=1
12 val[1]=3.14
13 val[2]=3
14 Dimension du tableau : 2
15 Composante 0 : 1
16 Composante 1 : 2
17 2eme Tableau
18 val[0]=1
19 val[1]=2
```

Oh Miracle ! celà a l'air de marcher correctement.

> Oui, mais ce n'est qu'une impression ! ! ! Ce type de programmation peut vite devenir une usine à gaz. Il va falloir travailler le langage C. Mais ceci est une autre histoire...(tout au moins un autre poly).

Chapitre 5

Annexes

5.1 Programmation en C

5.1.1 fichier main type

Voici l'ossature type du fichier *main.c* :

Listing 5.1 – Fichier main type

```
1  /********************** main.c ************************/
2  /* Description : ...                                  */
3  /*                                                    */
4  /******************************************************/
5  /* Dependances : ...                                  */
6  /******************************************************/
7  /* Variables globales : ...                           */
8  /* Fonctions externes utilisees : ...                 */
9  /******************************************************/
10 /* Auteurs :                                          */
11 /* Version :                                          */
12 /* Remarques :                                        */
13 /* Dernieres modifications :                          */
14 /* Copyright :                                        */
15 /*        Institut Galilee - Univ. Paris 13           */
16 /******************************************************/
17
18 /* Fichiers include */
19
20 /* Variables globales */
21
22 void main()
23 {
24 }
```

5.1.2 fichier C type

Voici l'ossature type du fichier *type.c* :

Listing 5.2 – Fichier C type

```
1  /********************** type.c ************************/
2  /* Description : ...                                  */
3  /*                                                    */
4  /* Il faut ici decrire les fonctions implementees.    */
5  /******************************************************/
6  /* Dependances : ...                                  */
7  /******************************************************/
8  /* Variables globales : ...                           */
9  /* Fonctions externes utilisees : ...                 */
10 /******************************************************/
11 /* Auteurs :                                          */
```

```
12 /* Version :                                              */
13 /* Remarques :                                           */
14 /* Dernieres modifications :                             */
15 /* Copyright :                                           */
16 /*        Institut Galilee - Univ. Paris 13              */
17 /*******************************************************/
18
19 /* Fichiers include */
20 #include "type.h"
21
22 /* Variables globales */
23
24 /* corps des fonctions */
```

5.1.3 fichier header type

Voici l'ossature type du fichier *type.h* associé avec le fichier *type.c* :

Listing 5.3 – fichier header type

```
1  /********************** type.h ***********************/
2  /* Description : ...                                   */
3  /*                                                     */
4  /*******************************************************/
5  /* Dependances : ...                                   */
6  /*******************************************************/
7  /* Auteurs :                                           */
8  /* Version :                                           */
9  /* Remarques :                                         */
10 /* Dernieres modifications :                           */
11 /* Copyright :                                         */
12 /*        Institut Galilee - Univ. Paris 13            */
13 /*******************************************************/
14 #ifndef _TYPE_H
15 #define _TYPE_H
16
17 /* Fichiers include */
18
19 /* Declaration de types donnees */
20
21 /* Prototypage des fonctions */
22
23 #endif
```

5.2 Exercices

5.2.1 Exercice 1 : Compilation séparée

Récupérer le fichier *exo1.tar.gz* contenant les sources utilisées pour cet exercice. Ce fichier est un fichier compressé à l'aide de l'utilitaire **gzip** (extension *.gz*) et archivé à l'aide de la commande **tar** (extension *.tar*). Pour l'utiliser, il faut tout d'abord le copier dans un répertoire (*./exercices* par exemple), décompresser (commande **gunzip**) puis le déarchiver (commande **tar**).

Session Shell

```
1 $ gunzip exo1.tar.gz
2 $ tar -xvf exo1.tar
```

Les fichiers seront dans le répertoire (*./exercices/exo1*). Le but de cet exercice est d'exécuter le programme *TestBessel* de fichier source *TestBessel.c*.

1. Quelles sont les dépendances entre les différents fichiers ?
2. Comment créer simplement l'ensemble des fichiers objets nécessaire à la création du programme exécutable ?

3. Comment, à partir de fichiers objets, créer le programme exécutable *TestBessel* ?
4. Comment doit-on recompiler le programme si le fichier source *bessi.c* a été modifié ?

5.2.2 Exercice 2 : Création et utilisation d'une librairie

Récupérer le fichier *exo2.tar.gz* contenant les sources utilisées pour cet exercice (voir exercice précédant). Les fichiers sont dans le répertoire (*./exercices/exo2*). Le but ici est de créer une librairie nommée *libbessel.a* (et le fichier header associé *bessel.h*) contenant l'ensemble des fonctions relatives aux fonctions de Bessel.

1. Quelles sont les commandes pour créer cette librairie ?
2. Ecrire le fichier header *bessel.h* associé à cette librairie.
3. Modifier le fichier *TestBessel.c* pour n'utiliser que la librairie et le fichier header. Comment créer le programme exécutable à partir de la librairie ?

5.2.3 Exercice 3 : Utilisation de Make

1. Reprendre l'exercice 1 et créer le fichier *makefile* permettant de gérer la compilation des différents modules et de créer le programme exécutable *TestBessel*. Les dépendances doivent être prises en compte.
2. Reprendre l'exercice 2 et créer le fichier *makefile* permettant de créer la librairie et de créer le programme exécutable *TestBessel* à partir de cette même librairie. Les dépendances doivent être prises en compte.