Informatique pour le calcul scientifique : base et outils
Compilation et débogage

Patrick Carribault

patrick.carribault@cea.fr

Ce document décrit la correction du TP. Il rappelle d'abord l'énoncé de chaque question et discute ensuite de la correction sous la forme suivante :

CORRECTION

I – Compilation

Ce premier exercice correspond à la première partie du cours (matinée) : il permet d'appréhender la compilation avec l'outil GCC en détaillant les différentes étapes, la compilation multi-fichiers et la création/manipulation de bibliothèques. Enfin, la dernière partie permet une introduction aux optimisations.

1 – Compilation simple

Le répertoire `PROJET1` contient un fichier C nommé `projet1.c` qui contient un programme très simple d'addition de deux vecteurs de taille N.

1. Utiliser le compilateur GCC pour générer le code préprocessé (option `-E`) dans le fichier `projet1.i`. Comparer le nombre de lignes générés par rapport au nombre de lignes du code source original (outil `wc -l`). Retrouve-t-on dans le fichier généré la macro définissant N ?

CORRECTION

```
$ gcc -E -o projet1.i projet1.c

$ wc -l projet1.i
1945 projet1.i

$ wc -l projet1.c
50 projet1.c

$ grep N projet1.i
```

2. Utiliser le cœur du compilateur de GCC pour générer le fichier assembleur `projet1.s` à partir du fichier `projet1.i`. Quel est le type de fichier généré ?

CORRECTION

```
$ gcc -S -o projet1.s projet1.i

$ file projet1.s
projet1.s: assembler source text
```

3. Continuer la compilation en utilisant l'outil `as` pour assembler le fichier `projet1.s`. Le résultat donne le fichier `projet1.o`.

CORRECTION

```
$ as -o projet1.o projet1.s
```

4. Finir la compilation pour générer l'exécutable `projet1`.

CORRECTION

```
$ gcc -o projet1 projet1.o
```

5. Exécuter le fichier `projet1` : il y a un problème !

CORRECTION

```
$ ./projet1
v[0] = 4
v[1] = 4
v[2] = 4
v[3] = 4
v[4] = 4
v[5] = 4
v[6] = 4
v[7] = 4
v[8] = 4
v[9] = 4
Segmentation fault (core dumped)
```

6. Pour vérifier le problème, recompiler le fichier `projet1.c` (en une seule ligne de commande cette fois-ci) en activant le niveau maximal pour les warnings : `-Wall`

CORRECTION

```
$ gcc -o projet1 -Wall projet1.c
projet1.c: In function ‘main’:
projet1.c:50:1: warning: control reaches end of non-void function [-
Wreturn-type]
 }
 ^
projet1.c:41:14: warning: ‘a’ is used uninitialized in this function
[-Wuninitialized]
```

```
  init_vecteur( a, N ) ;
                ^
```

7. Corriger les problèmes du code (très simple).

CORRECTION

Voir le fichier projet1_corrige.c pour une correction.

```
$ gcc -o projet1_corrige projet1_corrige.c

$ ./projet1_corrige
v[0] = 4
v[1] = 4
v[2] = 4
v[3] = 4
v[4] = 4
v[5] = 4
v[6] = 4
v[7] = 4
v[8] = 4
v[9] = 4
```

8. En examinant le fichier `projet1.c`, on peut observer que N est défini à travers une macro. Comment changer la valeur de cette macro durant la ligne de compilation (penser à l'option –D…) ?

CORRECTION

```
$ gcc -DN=30 -o projet1_corrige projet1_corrige.c

$ ./projet1_corrige
v[0] = 4
v[1] = 4
v[2] = 4
v[3] = 4
v[4] = 4
v[5] = 4
v[6] = 4
v[7] = 4
v[8] = 4
v[9] = 4
v[10] = 4
v[11] = 4
v[12] = 4
v[13] = 4
v[14] = 4
v[15] = 4
v[16] = 4
v[17] = 4
v[18] = 4
v[19] = 4
```

```
v[20] = 4
v[21] = 4
v[22] = 4
v[23] = 4
v[24] = 4
v[25] = 4
v[26] = 4
v[27] = 4
v[28] = 4
v[29] = 4
```

2 – Compilation multi-fichiers

Le répertoire PROJET2 contient le même projet avec une légère réorganisation des fichiers et des répertoires : les fonctions ont été mises dans des fichiers à part avec des fichiers *headers* pour définir les signatures de ces fonctions.

1. Examiner les fichiers et la hiérarchie des répertoires.

CORRECTION

```
$ ls -R
.:
affiche.c  include  init.c  kernel.c  projet2.c

./include:
affiche.h  init.h  kernel.h
```

2. Compiler ce projet en une seule ligne de commande (l'option `-I` permet de définir un répertoire dans lequel se situent les fichiers *header*s).

CORRECTION

```
$ gcc -Iinclude/ -o projet2 projet2.c kernel.c init.c affiche.c
```

3. Refaire la même opération en plusieurs lignes de commandes (une ligne par fichier source et une ligne pour générer l'exécutable final).

CORRECTION

```
$ gcc -Iinclude -c init.c

$ gcc -Iinclude -c affiche.c

$ gcc -Iinclude -c kernel.c

$ gcc -Iinclude -c projet2.c

$ gcc -o projet2 projet2.o init.o affiche.o kernel.o
```

3 – Bibliothèque statique

Le répertoire `PROJET3` va plus loin dans l'organisation du projet : nous souhaitons créer deux bibliothèques pour rassembler les fonctionnalités. C'est pourquoi les répertoires `libaff` (bibliothèque pour l'affichage d'un vecteur) et `libinit` (bibliothèque pour l'initialisation d'un vecteur) ont été créés.

1. Aller dans le répertoire `libaff`. Combien de fichiers source contient ce répertoire ?

CORRECTION

```
$ cd libaff

$ ls
affiche.c
```

2. Créer une bibliothèque statique pour ce fichier. Utiliser l'outil `ar` avec l'option `r` pour cela, la bibliothèque s'appellera `libaff.a`.

CORRECTION

```
$ gcc -I../include -c -o affiche.o affiche.c

$ ar r libaff.a affiche.o
ar: creating libaff.a
```

3. Faire de même pour le répertoire `libinit` pour créer un fichier `libinit.a`.

CORRECTION

```
$ cd ../libinit/

$ gcc -I../include -c -o init.o init.c

$ ar r libinit.a init.o
ar: creating libinit.a
```

4. Créer l'exécutable final en liant avec ces deux bibliothèques (ne pas oublier les options `-L` pour ajouter un répertoire dans la recherche des bibliothèques et `-l` pour lier à une bibliothèque).

CORRECTION

```
$ cd ..

$ gcc -Iinclude -o projet3 projet3.c kernel.c -Llibaff -Llibinit -
laff -linit

$ ./projet3
v[0] = 4
v[1] = 4
```

```
v[2] = 4
v[3] = 4
v[4] = 4
v[5] = 4
v[6] = 4
v[7] = 4
v[8] = 4
v[9] = 4
```

Attention : bien mettre les options –L et –l à la fin de la ligne de commande.

4 – Bibliothèque dynamique

Nous allons à présent créer des bibliothèques dynamiques dans le répertoire PROJET4.

1. Créer une bibliothèque dynamique dans le répertoire `libaff1` (compilation du fichier source avec l'option –`shared`). La bibliothèque s'appellera `libaff.so`.

CORRECTION

```
$ cd libaff1

$ gcc -I../include -fPIC -shared -o libaff.so affiche.c
```

2. Compiler le binaire principal en liant à cette bibliothèque dynamique.

CORRECTION

```
$ cd ..

$ gcc -Iinclude –o projet4 projet4.c init.c kernel.c -Llibaff1 -laff
```

3. Exécuter le programme : un problème survient. Résoudre ce problème grâce à la variable d'environnement LD_LIBRARY_PATH.

CORRECTION

```
$ ./projet4
./projet4: error while loading shared libraries: libaff.so: cannot
open shared object file: No such file or directory

$ export LD_LIBRARY_PATH=./libaff1:$LD_LIBRARY_PATH

$ ./projet4 v[0] = 4
v[1] = 4
v[2] = 4
v[3] = 4
v[4] = 4
v[5] = 4
v[6] = 4
v[7] = 4
v[8] = 4
```

```
v[9] = 4
```

4. En changeant la valeur de cette variable, utiliser la bibliothèque du répertoire `libaff2` (en la créant au préalable, avec le même nom `libaff.so`). La commande `ldd` permet de savoir à quelles bibliothèques dynamiques un exécutable est lié. Lancer de nouveau l'exécutable, l'affichage doit alors changer légèrement…

CORRECTION

```
$ cd libaff2/

$ gcc -I../include -fPIC -shared -o libaff.so affiche.c

$ cd ..

$ export LD_LIBRARY_PATH=./libaff2:$LD_LIBRARY_PATH

$ ./projet4
VERSION2 v[0] = 4
VERSION2 v[1] = 4
VERSION2 v[2] = 4
VERSION2 v[3] = 4
VERSION2 v[4] = 4
VERSION2 v[5] = 4
VERSION2 v[6] = 4
VERSION2 v[7] = 4
VERSION2 v[8] = 4
VERSION2 v[9] = 4
```

5 - Optimisations

Le répertoire PROJET5 va permettre de nous intéresser à la notion d'optimisation. Centrée sur GCC, cette étude est également valable pour d'autres compilateurs, en adaptant les commandes et les options.

1. Ce répertoire contient deux fichiers source : compiler ces fichiers avec des niveaux d'optimisations différents (`-O0`, `-O1`, `-O2` et `–O3`). Pour voir quelles passes le compilateur a appliquées, utiliser l'option `–fdump-tree-all`.

CORRECTION

```
$ gcc –O0 -fdump-tree-all -o projet5a projet5a.c
$ gcc –O1 -fdump-tree-all -o projet5a projet5a.c
$ gcc –O2 -fdump-tree-all -o projet5a projet5a.c
$ gcc –O3 -fdump-tree-all -o projet5a projet5a.c
```

Il suffit de regarder les fichiers générés par ces trois commandes.
Refaire de même avec le fichier projet5b.c

2. Quelles sont les différences entre les différents niveaux d 'optimisations (nombre de passes, …) ?

CORRECTION

Niveau d'optimisation 0 : 16 passes
Niveau d'optimisation 1 : 82 passes
Niveau d'optimisation 2 : 96 passes
Niveau d'optimisation 3 : 103 passes

3. Nous allons d'abord étudier le fichier `projet5a.c`. Il contient un code très simple (similaire au PROJET1). Utiliser la ligne de compilation suivante :

```
gcc –O3 –o projet5a –ftree-vectorizer-verbose=1 projet5a.c
```

L'option supplémentaire permet d'obtenir des informations du compilateur sur les optimisations. Décrire ces optimisations (utiliser les informations du code source pour comprendre).

CORRECTION

```
$ gcc -O3 -o projet5a -ftree-vectorizer-verbose=1 projet5a.c
Analyzing loop at projet5a.c:12
projet5a.c:10: note: vectorized 0 loops in function.
Analyzing loop at projet5a.c:20
projet5a.c:17: note: vectorized 0 loops in function.
Analyzing loop at projet5a.c:28
projet5a.c:26: note: vectorized 0 loops in function.
Analyzing loop at projet5a.c:20
Analyzing loop at projet5a.c:12
Analyzing loop at projet5a.c:12
Analyzing loop at projet5a.c:12
projet5a.c:34: note: vectorized 0 loops in function.
projet5a.c:20: note: Completely unroll loop 9 times
projet5a.c:12: note: Completely unroll loop 9 times
projet5a.c:12: note: Completely unroll loop 9 times
projet5a.c:12: note: Completely unroll loop 9 times
```

Le compilateur analyse toutes les boucles du programme en affichant le numéro de la ligne où débute la boucle en question. On remarque que, pour la même ligne de code source, le compilateur traite plusieurs boucles : à cause de certaines optimisations, une boucle du code source peut donner plusieurs boucles dans le code final (fission de boucle, loop peeling, …). De plus, le compilateur nous indique qu'il n'a pas pu *vectoriser* ces boucles (pour des raisons inconnus pour le moment). Enfin, la plupart des boucles ont été déroulées d'un facteur 9 : en fait, elles ont été complètement déroulées car le nombre d'itérations est 10 (ce nombre est N, mais N est une macro, donc il est remplacé par sa valeur par le préprocesseur). Ceci signifie qu'il n'y a plus de boucles.

4. Répéter la même opération avec le fichier `projet5b.c`. Quelles sont les différences ?

CORRECTION

```
$ gcc -O3 -o projet5a -ftree-vectorizer-verbose=1 projet5b.c
Analyzing loop at projet5b.c:12
projet5b.c:10: note: vectorized 0 loops in function.
Analyzing loop at projet5b.c:20
projet5b.c:17: note: vectorized 0 loops in function.
Analyzing loop at projet5b.c:28
projet5b.c:26: note: vectorized 0 loops in function.
Analyzing loop at projet5b.c:20
Analyzing loop at projet5b.c:12
Analyzing loop at projet5b.c:12
Analyzing loop at projet5b.c:12
projet5b.c:34: note: vectorized 0 loops in function.
```

La sortie est à peu près la même, sauf que le compilateur n'a pas déroulé entièrement les boucles. En examinant le code source, le nombre d'itérations des boucles n'est plus 10, mais N, qui n'est pas connu lors de la compilation.

5. En utilisant une verbosité supérieure, il est possible de connaitre les raisons pour lesquelles le compilateur n'a pas vectorisé les boucles du fichier `projet5b.c`. L'architecture cible n'est pas bien décrite (architecture 32bits Intel générique, sans les capacités réelles du processeur). La liste des capacités peut être obtenue en regardant le fichier `/proc/cpuinfo` (rubrique `flags`). Si le processeur peut supporter un jeu d'instructions vectoriel, alors il apparaitra dans la liste (par exemple SSE2, SSE3, AVX, AVX2, …). Utiliser l'option `–msse3` pour activer le support de la vectorisation. Que peut-on observer ?

CORRECTION

```
$ cat /proc/cpuinfo | grep flags
flags      : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge
mca cmov pat pse36 clflush mmx fxsr sse sse2 nx constant_tsc pni
monitor ssse3

$ gcc -O3 -msse3 -o projet5a -ftree-vectorizer-verbose=1 projet5b.c
Analyzing loop at projet5b.c:12
Vectorizing loop at projet5b.c:12
projet5b.c:12: note: === vect_do_peeling_for_loop_bound ===Setting
upper bound of nb iterations for epilogue loop to 3
projet5b.c:12: note: LOOP VECTORIZED.
projet5b.c:10: note: vectorized 1 loops in function.
projet5b.c:12: note: Completely unroll loop 3 times
Analyzing loop at projet5b.c:20
Vectorizing loop at projet5b.c:20
projet5b.c:20: note: create runtime check for data references *_10
and *_8
projet5b.c:20: note: create runtime check for data references *_13
and *_8
projet5b.c:20: note: created 2 versioning for alias checks.
projet5b.c:20: note: === vect_do_peeling_for_loop_bound ===Setting
```

```
upper bound of nb iterations for epilogue loop to 0
projet5b.c:20: note: LOOP VECTORIZED.
projet5b.c:17: note: vectorized 1 loops in function.
projet5b.c:20: note: Turned loop into non-loop; it never loops.
Analyzing loop at projet5b.c:28
projet5b.c:26: note: vectorized 0 loops in function.
Analyzing loop at projet5b.c:20
Vectorizing loop at projet5b.c:20
projet5b.c:20: note: === vect_do_peeling_for_loop_bound ===Setting
upper bound of nb iterations for epilogue loop to 0
projet5b.c:20: note: LOOP VECTORIZED.
Analyzing loop at projet5b.c:12
Vectorizing loop at projet5b.c:12
projet5b.c:12: note: === vect_do_peeling_for_loop_bound ===Setting
upper bound of nb iterations for epilogue loop to 3
projet5b.c:12: note: LOOP VECTORIZED.
Analyzing loop at projet5b.c:12
Vectorizing loop at projet5b.c:12
projet5b.c:12: note: === vect_do_peeling_for_loop_bound ===Setting
upper bound of nb iterations for epilogue loop to 3
projet5b.c:12: note: LOOP VECTORIZED.
Analyzing loop at projet5b.c:12
Vectorizing loop at projet5b.c:12
projet5b.c:12: note: === vect_do_peeling_for_loop_bound ===Setting
upper bound of nb iterations for epilogue loop to 3
projet5b.c:12: note: LOOP VECTORIZED.
projet5b.c:34: note: vectorized 4 loops in function.
projet5b.c:20: note: Turned loop into non-loop; it never loops.
projet5b.c:12: note: Completely unroll loop 3 times
projet5b.c:12: note: Completely unroll loop 3 times
projet5b.c:12: note: Completely unroll loop 3 times
```

Certaines boucles ont été vectorisées en indiquant au compilateur que notre processeur cible supportait le jeu d'instructions vectoriels SSE3 (support démontré par la première ligne de commande affiachant les flags du processeur).

6. Pour certaines boucles, le compilateur peut dire qu'il a généré un test dynamique (*runtime check*) pour vérifier une propriété du code durant l'exécution. Les références correspondent à des représentations internes. Utiliser l'option `–fdump-tree-vect` pour générer un fichier après l'étape de vectorisation : les symboles peuvent alors être décryptés…

CORRECTION

Extrait de la question précédente :

```
projet5b.c:20: note: create runtime check for data references *_10
and *_8
projet5b.c:20: note: create runtime check for data references *_13
and *_8
```

Il semble que le compilateur a dû générer un test qui sera évalué lors de l'exécution du code, pour les références (pointeurs) *_10, *_13 et *_8. Bien sûr, ceci ne fait pas référence à une variable du

code source, mais à une représentation interne. Pour savoir quelles variables sont concernées, il faut générer une représentation du code pendant la compilation grâce à la ligne suivante

```
$ gcc -O3 -msse3 -fdump-tree-vect -o projet5a -ftree-vectorizer-verbose=1 projet5b.c
```

En parcourant le fichier généré (`projet5b.c.112t.vect`), on trouve les lignes suivantes :

```
  _8 = c_7(D) + _6;
  _10 = a_9(D) + _6;
  _13 = b_12(D) + _6;
```

Ainsi `_8` fait référence au tableau `c`, `_10` au tableau `a` et `_13` au tableau `b`. Le compilateur a dû générer un test pour vérifier que ces tableaux ne se recouvrent pas en mémoire, seule condition qui garantit que les itérations de la boucle sont indépendantes.

II – Environnement de programmation

Cette deuxième partie met en application la seconde partie du cours sur l'environnement de compilation via l'utilisation de l'outil `make`.

1 – Règles et dépendances

Le répertoire `PROJET6` contient un fichier source similaire au `PROJET1` étudié dans l'exercice précédent. Le but est alors de créer un `Makefile` avec plusieurs règles.

1. Créer un fichier `Makefile` avec une règle `all` qui dépend d'une règle `projet6`. Cette dernière correspond à la compilation en une ligne de commande du fichier `projet6.c` vers l'exécutable `projet6`.

CORRECTION

Contenu du fichier Makefile

```
all: projet6

projet6: projet6.c
	gcc -o projet6 projet6.c
```

2. Créer un ensemble de règles pour créer l'exécutable `projet6_multi` en découpant la compilation en plusieurs étapes (fichier preprocessé, fichier assembleur, fichier objet, exécutable). Il est possible d'utiliser `gcc` pour toutes les étapes, sauf pour l'assembleur. On pourra alors utiliser la commande `as`.

CORRECTION

Contenu du fichier Makefile

```
all: projet6 projet6_multi

projet6: projet6.c
```

```
	gcc -o projet6 projet6.c

projet6_multi: projet6.o
	gcc -o projet6_multi projet6.o

projet6.o: projet6.s
	as -o projet6.o projet6.s

projet6.s: projet6.i
	gcc -S -o projet6.s projet6.i

projet6.i: projet6.c
	gcc -E -o projet6.i projet6.c
```

3. Pour vérifier la validité de ce `Makefile`, il suffit de modifier légèrement certains fichiers et relancer la commande `make` pour voir dérouler les commandes nécessaires pour reconstruire les fichiers exécutables. Pour juste changer la date de modification d'un fichier `fic`, il suffit d'utiliser la commande `touch fic`. Valider le `Makefile` ainsi.

CORRECTION

```
$ make
gcc -o projet6 projet6.c
gcc -o projet6_multi projet6.o

$ make
make: Nothing to be done for `all'.

$ touch projet6.c

$ make
gcc -o projet6 projet6.c
gcc -E -o projet6.i projet6.c
gcc -S -o projet6.s projet6.i
as -o projet6.o projet6.s
gcc -o projet6_multi projet6.o
```

2 – Règles génériques

Le répertoire PROJET7 contient plusieurs fichiers : chacun correspond à un programme.

1. Inspiré de la question précédente, créer un `Makefile` permettant de compiler ces 3 programmes, sans dupliquer 3 fois le nombre de règles dans le fichier `Makefile`. L'utilisation de règles génériques pourrait être utile…

CORRECTION

Contenu du fichier Makefile

```
all:projet7a projet7b projet7c

# Surcharge de la règle implicite existante dans l'outil make
```

```
# En ne mettant aucune action, cette règle deivent obsolète
%.o: %.c

%.o : %.s
	as -o $@ $<

%.s : %.i
	gcc -S -o $@ $<

%.i : %.c
	gcc -E $< > $@

projet7%: projet7%.o
	gcc -o $@ $<
```

Note : il a fallu désactiver la règle .c → .o de make afin qu'il utilise nos règles génériques plus complexes pour générer un fichier o à partir d'un fichier c en passant par un fichier I et s.

2. Vérifier ce `Makefile` en touchant certains fichiers.

CORRECTION

```
$ make
gcc -E projet7a.c > projet7a.i
gcc -S -o projet7a.s projet7a.i
as -o projet7a.o projet7a.s
gcc -o projet7a projet7a.o
gcc -E projet7b.c > projet7b.i
gcc -S -o projet7b.s projet7b.i
as -o projet7b.o projet7b.s
gcc -o projet7b projet7b.o
gcc -E projet7c.c > projet7c.i
gcc -S -o projet7c.s projet7c.i
as -o projet7c.o projet7c.s
gcc -o projet7c projet7c.o
rm projet7a.i projet7a.o projet7b.i projet7b.o projet7c.i projet7c.o
projet7a.s projet7b.s projet7c.s

$ make
make: Nothing to be done for `all'.

$ touch projet7b.c

$ make
gcc -E projet7b.c > projet7b.i
gcc -S -o projet7b.s projet7b.i
as -o projet7b.o projet7b.s
gcc -o projet7b projet7b.o
rm projet7b.i projet7b.o projet7b.s
```

3 – Variables

Le répertoire PROJET8 contient une copie du projet numéro 2, étudié dans le précédent exercice. Ce projet contient alors plusieurs fichiers sources et plusieurs headers

1. Créer un fichier `Makefile` permettant de générer le fichier exécutable `projet8`. Il faudra alors bien gérer les dépendances. Par exemple, modifier le fichier `include/init.h` impose alors la recompilation de `init.c` et `projet2.c` (car ces deux fichiers incluent `init.h`) et, enfin, la mise à jour de l'exécutable finale. Il est alors nécessaire de bien construire les règles. Il est également possible d'utiliser des variables pour stocker la liste des fichiers headers ou des sources C.

CORRECTION

Contenu du fichier Makefile

```
REP_HEADER=include
SRC= projet8.c affiche.c init.c kernel.c
HEADER = ./$(REP_HEADER)/affiche.h \
       ./$(REP_HEADER)/init.h \
       ./$(REP_HEADER)/kernel.h

all: projet8

projet8: $(SRC) $(HEADER)
	gcc -o $@ $(SRC) -Iinclude
```

4 - Récursivité

Le répertoire PROJET9 contient une copie du PROJET3 avec la définition de deux bibliothèques statiques.

2. Créer un fichier `Makefile` dans le répertoire `libaff` pour générer la bibliothèque statique.

CORRECTION

Contenu du fichier Makefile

```
all: libaff.a

libaff.a: affiche.o
	ar r $@ $^

affiche.o: affiche.c ../include/affiche.h
	gcc -c -o $@ $< -I../include
```

3. Faire de même dans le répertoire `libinit`.

CORRECTION

Contenu du fichier Makefile

```
all: libinit.a

libinit.a: init.o
	ar r $@ $^

init.o: init.c ../include/init.h
	gcc -c -o $@ $< -I../include
```

4. Enfin, créer un `Makefile` à la racine du projet qui appelle les fichiers `Makefile` des sous-répertoires `libaff` et `libinit`. Attention, il est nécessaire de gérer correctement les dépendances.

CORRECTION

```
$ cat Makefile
OBJ=projet9.o kernel.o

all: projet9

projet9: libs $(OBJ)
	gcc -o $@ $(OBJ) -Llibaff -laff -Llibinit -linit

projet9.o: projet9.c include/init.h include/kernel.h
include/affiche.h
	gcc -c -Iinclude -o $@ $<

kernel.o: kernel.c
	gcc -c -o $@ $<

libs:
	$(MAKE) -C libaff
	$(MAKE) -C libinit

$ make
make -C libaff
make[1]: Entering directory
`/home/bolle/CORRECTION/JOURNEE_COMPILATION/TP_COMPIL/02_ENV_COMPIL/
PROJET9/libaff'
gcc -c -o affiche.o affiche.c -I../include
ar r libaff.a affiche.o
ar: creating libaff.a
make[1]: Leaving directory
`/home/bolle/CORRECTION/JOURNEE_COMPILATION/TP_COMPIL/02_ENV_COMPIL/
PROJET9/libaff'
make -C libinit
make[1]: Entering directory
`/home/bolle/CORRECTION/JOURNEE_COMPILATION/TP_COMPIL/02_ENV_COMPIL/
PROJET9/libinit'
gcc -c -o init.o init.c -I../include
ar r libinit.a init.o
```

```
ar: creating libinit.a
make[1]: Leaving directory
`/home/bolle/CORRECTION/JOURNEE_COMPILATION/TP_COMPIL/02_ENV_COMPIL/
PROJET9/libinit'
gcc -c -Iinclude -o projet9.o projet9.c
gcc -c -o kernel.o kernel.c
gcc -o projet9 projet9.o kernel.o -Llibaff -laff -Llibinit -linit
```

III – Débogage

Cette dernière partie propose une introduction au débogage en ligne de commande grâce à l'outil GDB.

1 – Problème d'opération ?

1. Compiler le fichier source du `PROJET10` avec la commande `gcc –o projet10 projet10.c`

CORRECTION

```
$ gcc -o projet10 projet10.c
```

2. Exécuter le programme (avec un argument de type entier).

CORRECTION

```
$ ./projet10 10
Value v captured from input...
Division = 0
```

3. Essayer avec l'argument 0.

CORRECTION

```
$ ./projet10 0
Value v captured from input...
Floating point exception (core dumped)
```

4. Lancer GDB et initialiser les paramètres de lancement avec l'argument `0` (`gdb ./projet10`, et `set args 0`).

CORRECTION

```
$ gdb ./projet10
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
```

```
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./projet10...(no debugging symbols
found)...done.
(gdb) set args 0
```

5. Lancer le programme (commande `run`). A-t-on des informations sur l'endroit du problème ?

CORRECTION

```
(gdb) run
Starting program:
/home/bolle/CORRECTION/JOURNEE_COMPILATION/TP_COMPIL/03_DEBUG/PROJET
10/projet10 0
Value v captured from input...

Program received signal SIGFPE, Arithmetic exception.
0x08048564 in main ()
```

6. Recompiler le programme pour activer les symboles de debugging et refaire l'opération précédente. Quelle ligne du code source pose problème ? Afficher la valeur de `v`.

CORRECTION

```
$ gcc -g -o projet10 projet10.c

$ gdb ./projet10
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./projet10...done.
(gdb) set args 0
(gdb) r
```

```
Starting program:
/home/bolle/CORRECTION/JOURNEE_COMPILATION/TP_COMPIL/03_DEBUG/PROJET
10/projet10 0
Value v captured from input...

Program received signal SIGFPE, Arithmetic exception.
0x08048564 in main (argc=2, argv=0xbffff064) at projet10.c:18
18          t = 8 / v ;
(gdb) p v
$1 = 0
```

7. Il est alors nécessaire de suivre l'évolution de la valeur de `v` pour regarder quand elle a été initialisée à 0 : il faut alors s'arrêter dans le programme, au moment où `v` existe déjà (`v` doit être déclarée) et ensuite, il faut suivre la valeur de `v`.
    a. Ajouter un *breakpoint* dans le programme à la ligne correspondant au premier `if` du code (commande `b`).

CORRECTION

```
(gdb) l main
1     #include <stdio.h>
2     #include <stdlib.h>
3
4
5     int main(int argc, char ** argv ) {
6           int v = -1 ;
7           int t ;
8
9           if( argc != 2 ) {
10               fprintf( stderr, "Error: usage %s entier\n", argv[0]
) ;
(gdb) b 9
Breakpoint 1 at 0x804850e: file projet10.c, line 9.
```

    b. Lancer le programme

CORRECTION

```
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program:
/home/bolle/CORRECTION/JOURNEE_COMPILATION/TP_COMPIL/03_DEBUG/PROJET
10/projet10 0

Breakpoint 1, main (argc=2, argv=0xbffff064) at projet10.c:9
9           if( argc != 2 ) {
```

    c. Une fois le programme stoppé au niveau du `if`, ajouter un *watchpoint* sur `v` (commande `watch`).

CORRECTION

```
(gdb) watch v
Hardware watchpoint 2: v
```

d. Continuer le programme (commande `c`).

CORRECTION

```
(gdb) c
Continuing.
Hardware watchpoint 2: v

Old value = -1
New value = 0
main (argc=2, argv=0xbffff064) at projet10.c:16
16              printf( "Value v captured from input...\n" ) ;
```

e. Regarder quand cette valeur a changé.

CORRECTION

```
(gdb) p v
$2 = 0
(gdb) l
11              exit( 1 ) ;
12          }
13
14          v = atoi( argv[1] ) ;
15
16          printf( "Value v captured from input...\n" ) ;
17
18          t = 8 / v ;
19
20          printf( "Division = %d\n", t ) ;
```

Cette valeur a donc changé à la ligne 14 (car le changement est effectif au début de la ligne 16 : c'est l'information qui est venu du *watchpoint* de la question précédente). L'erreur bien donc de l'appel de fonction `atoi` avec l'argument `argv[1]`. Ce tableau correspond au argument en entrée du programme, l'erreur vient donc directement de l'argument 0 que nous avons mis en entrée du programme.

2 – Problème de mémoire ?

1. Compiler le fichier source dans le répertoire `PROJET11` avec les symboles de debugging.

CORRECTION

```
$ gcc -g -o projet11 projet11.c
```

2. Lancer le code avec comme argument 50. Que se passe-t-il ?

CORRECTION

```
$ ./projet11 50
Valeur v = 50
Segmentation fault (core dumped)
```

3. Naviguer dans le debuggeur pour trouver le problème…

CORRECTION

```
$ gdb ./projet11
GNU gdb (Ubuntu 7.7-0ubuntu3.1) 7.7
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show
copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./projet11...done.
(gdb) set args 50
(gdb) r
Starting program:
/home/bolle/CORRECTION/JOURNEE_COMPILATION/TP_COMPIL/03_DEBUG/PROJET
11/projet11 50
Valeur v = 50

Program received signal SIGSEGV, Segmentation fault.
0x0804859b in main (argc=2, argv=0xbffff064) at projet11.c:24
24              a[i] = i ;
(gdb) p i
$1 = 0
(gdb) p a
$2 = (int *) 0x0
```

Nous suivons la même méthode que pour le programme précédent : tout d'abord, l'application est lancée dans le débuggeur pour obtenir la ligne de code qui pose problème. Au moment du plantage, nous essayons de voir ce qu'il se passe : `i` vaut 0, mais `a` faut NULL (0x0).

```
(gdb) l main
1    #include <stdio.h>
2    #include <stdlib.h>
```

```
3
4
5     int main(int argc, char ** argv ) {
6           int v = -1 ;
7           int * a ;
8           int i ;
9
10          if( argc != 2 ) {
(gdb) b 10
Breakpoint 1 at 0x804850e: file projet11.c, line 10.
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program:
/home/bolle/CORRECTION/JOURNEE_COMPILATION/TP_COMPIL/03_DEBUG/PROJET
11/projet11 50

Breakpoint 1, main (argc=2, argv=0xbffff064) at projet11.c:10
10          if( argc != 2 ) {
(gdb) watch a
Hardware watchpoint 2: a
(gdb) c
Continuing.
Valeur v = 50
Hardware watchpoint 2: a

Old value = (int *) 0xb7fbf000
New value = (int *) 0x0
main (argc=2, argv=0xbffff064) at projet11.c:23
23          for ( i = 0 ; i < v ; i++ ) {
```

Ainsi, on examine le début de la fonction main et on met en place un *breakpoint* au niveau de la première instruction (c'est à dire la construction `if`). Grâce à cela, nous allons nous arrêter à un moment où la variable `a` a été déclarée, mais non encore définie. Ensuite, nous mettons un *watchpoint* sur `a` pour suivre son évolution. Sa valeur passe à NULL juste avant la ligne 23…

```
(gdb) l
18          printf( "Valeur v = %d\n", v ) ;
19
20
21          a = (int *)malloc( (v-100) * sizeof( int ) ) ;
22
23          for ( i = 0 ; i < v ; i++ ) {
24               a[i] = i ;
25          }
26
27
(gdb) p v
$3 = 50
(gdb) p v-100
$4 = -50
```

On examine le code source pour voir ce qu'il se passe autour de cette ligne 23. La variable `a` est effectivement initialisée avec la fonction `malloc`. On affiche les valeurs de `v` et `v-100` pour se rendre compte que la fonction `malloc` a été appelée avec un argument négatif.