DETROYAT Alexis
VINCENT Côme
BODART Maxime

MOCA
Rapport
Semaine 1

Modifications

En compilant avec l'option **`-Wall`**, nous avons obtenu le warning suivant :

```
dico.c: In function 'insertDico':
dico.c:158:9: warning: variable 'newDictionaryPrevious' set but not used
[-Wunused-but-set-variable]
  158 |   dico* newDictionaryPrevious = (dico*) malloc(sizeof(dico));
      |         ^~~~~~~~~~~~~~~~~~~~~
```

Cela signifie que la variable **`newDictionaryPrevious`** n'est jamais utilisée, bien qu'elle soit déclarée et définie. Nous avons résolu ce premier bug en supprimant toutes les lignes de code impliquant cette variable, puisqu'elle se trouve être parfaitement inutile.

Ensuite, nous avons réexécuté le programme et constaté l'erreur de segmentation. Pour la résoudre, nous avons recompilé en mode debug (**`gcc -Wall -g dico.c -lm -o dico`**) puis lancé l'exécutable dans une instance de **`gdb`** (**`gdb dico`**). En utilisant la commande **`run`**, on obtient ceci :

```
Program received signal SIGSEGV, Segmentation fault.
_IO_feof (fp=0x0) at ./libio/feof.c:35
35      ./libio/feof.c: No such file or directory.
```

Nous avons alors utilisé la commande **`backtrace`** pour afficher la pile d'appel de fonctions et essayer d'y voir plus clair :

```
(gdb) backtrace
#0  _IO_feof (fp=0x0) at ./libio/feof.c:35
#1  0x0000555555555cee in displayDico (dictionary=0x55555555a4f0) at
dico.c:251
#2  0x0000555555555ecd in main () at dico.c:286
```

Nous avons donc constaté que le bug provient d'un appel sus-jacent à **`displayDico`** dans la pile, en provenance de **`main`**. Avec la commande **`up`**, on obtient :

```
(gdb) up
#1  0x0000555555555cee in displayDico (dictionary=0x55555555a4f0) at
dico.c:251
251         if (!feof(f))
```

On affiche alors la valeur de **f** (**p** pour afficher la valeur de la variable) :

```
(gdb) p/x f
$1 = 0x0
```

Il s'agit donc d'un bug dans la fonction **`displayDico`**, lors du test de fin de fichier : le descripteur de fichier est nul, et l'appel à **`feof`** sur le pointeur NULL provoque l'envoi d'un signal **SIGSEGV**. Le descripteur de fichier est NULL dès sa définition car on ouvre le fichier résultat en mode **`rw+`** (mode lecture + écriture), ce qui signifie que le fichier n'est pas créé s'il n'existe pas. Nous avons donc trouvé deux solutions à ce problème :

- créer manuellement le fichier de nom **`DICORES`** et laisser le mode **`rw+`**
- changer le mode en **`w+`** qui permet aussi d'ouvrir le fichier en mode lecture+écriture, mais crée le fichier s'il n'existe pas (et l'écrase s'il existe) d'après nos recherches.

Dans le code fourni, c'est la première solution qui a été conservée pour le moment. Quoi qu'il en soit, nous avons ajouté une condition pour tester immédiatement après l'ouverture si le descripteur est NULL, dans **`displayDico`** comme dans le **`main`** (après ajout de la possibilité d'entrer un nom de fichier en ligne de commande).

Enfin, pour permettre de spécifier un nom de fichier en argument, nous avons utilisé les variables **`argc`** (argument counter) et **`argv`** (argument vector) du **`main`**. Nous commençons par vérifier la valeur de **`argc`** :

- si **`argc`** vaut **1** (il n'y a pas d'argument en ligne de commande), on utilise la macro **`TEXTE`** comme nom de fichier de sortie ;
- si **`argc`** vaut **2** (il y a un seul argument en ligne de commande), on utilise cet argument (**`argv[1]`**) comme nom de fichier de sortie ;
- sinon, on affiche une erreur et on termine le programme.

Modularité

Nous avons modularisé l'application (dont le code était intégralement contenu dans le fichier dico.c à l'origine) en 6 catégories :

- **dico_tools** (fichiers .c et .h) est un module de gestion de la structure de dictionnaire, dans lequel se trouvent les fonctions **`insertDico`** et **`addToDico`** ;

- **word_tools** (fichiers .c et .h) est un module de gestion des différentes structures associées aux mots, dans lequel se trouvent les fonctions **`next_word`**, **`compareWord`** et **`incWord`** ;

- **display** (fichiers .c et .h) est un module d'affichage, où se trouvent les fonctions **`displayWord`**, **`displayNodes`** et **`displayDico`** ;

- **serialization** (fichiers .c et .h) est un module dédié à la sérialisation/désérialisation des données, c'est-à-dire à leur conversion entre une structure d'arbre binaire de recherche et une structure linéaire (table de hachage), où se trouvent les fonctions **`hash`**, **`deserializeDico`** et **`serializeDico`** ;

- **structures** (fichier .h uniquement) est un module-header où se trouvent toutes les structures de données utilisées par l'application, soit **`emplacement_t`**, **`mot_data_t`**, **`mot_t`**, **`dico`** ;

- **dico** (fichier .c uniquement) est le module principal de l'application, où se trouve la fonction **`main`** (isolée).

Makefile

Nous avons écrit notre Makefile en exploitant la règle implicite **`.c.o`** ; pour cela, nous avons initialisé les variables usuelles **`CC`** et **`CFLAGS`.** Ensuite, nous définissons quelques variables supplémentaires qui permettent de gagner en lisibilité ainsi qu'en efficacité :

- **`SRCS`** et **`HEADERS`** correspondent respectivement aux listes des noms de fichiers .c et .h présents dans le répertoire courant, récupérés grâce à la méthode GNU-make **`wildcard`** ;
- **`OBJS`** correspond à l'ensemble des fichiers objets (.o) à générer à partir des fichiers sources, déterminé via une substitution de suffixe depuis **`SRCS`** ;
- **`MAIN_OBJS`** correspond aux fichiers objets associés aux exécutables à générer ; ces objets sont ici isolés de **`OBJS`** afin de pouvoir ajouter une règle dans laquelle nous plaçons tous les headers en tant que dépendances de ces fichiers objets (et ainsi forcer la recompilation si n'importe quel header est modifié) ;
- **`EXEC`** correspond classiquement à l'ensemble des fichiers exécutables à générer, obtenu directement depuis **`MAIN_OBJS`** via une autre substitution de suffixe.

Pour l'instant, nous avons prévu un petit mode debug par l'intermédiaire du bloc **`ifdef ... endif`**, qui ajoute l'option de compilation **`-g`** (compilation en mode debug, nécessaire à l'exécution de **`gdb`**) aux **`CFLAGS`** ainsi que l'option de preprocessing **`-DDEBUG`** (définition de la macro **`DEBUG`** dans les fichiers sources, donc exécution du code placé sous cette macro) aux **`CPPFLAGS`**. Ce bloc pourra être enrichi par la suite.

La directive **`.PHONY : clean`** nous permet de différencier la règle **`clean`** des autres règles du Makefile, laquelle procède, comme son nom l'indique au nettoyage du répertoire

courant, c'est-à-dire à la suppression de tous les fichiers objets ainsi que tous les exécutables.

Vient alors la règle standard **`all`**, exécutée si la commande **`make`** est invoquée sans argument, qui appelle à son tour toutes les règles associées aux fichiers exécutables à générer (dans notre cas, **`dico`** seulement). A son tour, la règle **`dico`** appelle toutes les règles attribuées aux fichiers objets, grâce à la règle implicite **`.c.o`** qui procède à la compilation des fichiers sources pour générer ces objets. Quant à la règle **`dico`** elle-même, elle dépend de l'ensemble des fichiers objets, et procède à l'édition de lien avec l'option **`-lm`** (pour intégrer la librairie **`math.h`**) pour générer l'exécutable final.

Dans un souci de propagation d'éventuelles modifications apportées uniquement à un ou plusieurs headers, nous avons également précisé manuellement toutes les dépendances des fichiers objets à leurs headers respectifs.

```
CC=gcc
CFLAGS=-Wall -Werror
SRCS=$(wildcard *.c)
HEADERS=$(wildcard *.h)
OBJS=$(SRCS:.c=.o)
MAIN_OBJS=dico.o
EXEC=$(MAIN_OBJS:.o=)

ifdef DEBUG
CFLAGS+=-g
CPPFLAGS+=-DDEBUG
endif

.PHONY : clean

all : $(EXEC)

dico_tools.o : dico_tools.h structures.h word_tools.h serialization.h
display.o : display.h structures.h
serialization.o : serialization.h structures.h word_tools.h
word_tools.o : word_tools.h structures.h

$(MAIN_OBJS): $(HEADERS)

dico : $(OBJS)
    $(CC) $(OBJS) -lm -o $@

clean :
    @echo "Nettoyage du répertoire courant"
    -rm *.o $(EXEC)
```