DETROYAT Alexis
VINCENT Côme
BODART Maxime

MOCA
Rapport
Semaine 4

Partie 1 : Tests système pour le projet

Pour tester l'exécutable dico nous avons décidé d'utiliser un script **bash** et la commande **awk** pour analyser la sortie du programme. Nous nous sommes restreint à tester si tous les mots sont comptés correctement, c'est-à-dire si le nombre d'occurrences est le bon. Comme référence nous avons dans un premier temps tenté d'écrire une fonction bash qui calculerait ce nombre d'occurrences, mais avons décidé de l'abandonner à cause de comportements inexpliqués, au profit d'un site web[1] et de commandes bash pour trier et nettoyer le résultat.

Le script final exécute, sur tous les fichiers trouvés dans le dossier d'**exemple** pour lesquels il existe un fichier de même nom dans le dossier **expected**, la commande **diff** entre la sortie nettoyée de **dico** et la sortie attendue. Le résultat est affiché dans le terminal et décrit en détail dans le dossier **auto_test/res**.

Malgré la faible portée des tests effectués à ce niveau (pas de test de la position des mots entre autre), le script nous a tout de même permis de trouver plusieurs points à corriger dans le code source de **dico**, notamment la gestion des ponctuations. Une fois ces erreurs corrigées, il reste des erreurs étonnantes que nous n'avons pas encore eu le temps de traiter, notamment des mots absents du dictionnaire (avec le texte **wells.txt**). Il nous faut également améliorer la création des résultats attendus car l'outil utilisé ne compte pas les mots composés et abrégés de la même manière que **dico**.

Une limitation de ce test est que la méthode d'analyse de la sortie standard du programme n'est pas très robuste, une modification dans l'affichage impliquerait donc le besoin de retravailler le script **awk** de ce test.

Les tests effectués avec les exemples donnés et construits donnent une couverture de code de 100% des branches et 90% des lignes de code.

Partie 2 : Evaluation de la couverture

Afin d'être en mesure d'évaluer la couverture de nos futurs tests, nous avons modifié nos Makefile et notre arborescence. Premièrement, nous avons ajouté un répertoire **auto_tests** directement sous le répertoire principal de **Projet**, dans lequel se trouvent tous les éléments relatifs aux tests et dont l'organisation est très similaire au reste de notre arborescence :

- un répertoire **headers** qui contient les fichiers **.h** associés aux tests : **AllTests.h** et **CuTest.h**

[1] https://www.browserling.com/tools/word-frequency

- un répertoire **objects** qui contient les fichiers **.o** associés aux tests

- un répertoire **sources** qui contient les fichiers **.c** associés aux tests :
  - **CuTest.c** est le premier fichier source du package **CuTest** fourni définissant les primitives de test
  - **AllTests.c** est le second fichier source du package **CuTest** fourni chargé de lancer tous les tests
  - **TestsSerialization.c** contient la suite de tests sur le fichier **serialization.c**
  - **TestsWords.c** contient la suite de tests sur le fichier **word_tools.c**
  - **testUdico.c** est le programme principal de test qui génère l'exécutable

- un (troisième …) **Makefile**, appelé par le Makefile principal si l'on utilise l'argument **make test** :

```
#################
### VARIABLES ###
#################

CC=gcc
CFLAGS=-Wall -Werror

# Directories for source files, object files, headers, executable files, and
Doxygen documentation
SRCDIR=sources
OBJDIR=objects
HEADERSDIR=headers
EXECDIR=.
LIBHEADERSDIR=../libs/headers

# Files names : sources, headers, objects, objects that correspond to an
executable file name, and executable files
SRCS=$(wildcard $(SRCDIR)/*.c)
HEADERS=$(wildcard $(HEADERSDIR)/*.h)
OBJS=$(SRCS:$(SRCDIR)/%.c=$(OBJDIR)/%.o)
MAIN_OBJS=$(OBJDIR)/testDico.o
EXEC=$(MAIN_OBJS:$(OBJDIR)/%.o=$(EXECDIR)/%)

# -fprofile-arcs -ftest-coverage to be able to run test file and gcov
OPTGCOV = -fprofile-arcs -ftest-coverage

CFLAGS= -I$(HEADERSDIR) -I../headers -Wall $(OPTGCOV) -I$(LIBHEADERSDIR)
LDFLAGS= $(OPTGCOV) -lm


#############
### RULES ###
#############
```

```
.PHONY: clean

all: ensureDirs test

# Rule to make the object files
$(OBJDIR)/%.o:$(SRCDIR)/%.c
    $(CC) $(CFLAGS) $(CPPFLAGS) -c $< -o $@

# Rule to make the executable files
$(EXEC): $(OBJS)
    $(CC) $(OBJS) ../libs/objects/serialization.o ../objects/word_tools.o -o
$(EXEC) $(LDFLAGS)

# Rule to make the tests
test:
    make clean
    make $(EXEC)
    @echo
    @echo "----- Lancement de l'exécutable des tests et analyse de
couverture -----"
    ./testDico
    gcov -b -n -f -s ../sources/ -o ../$(OBJDIR) -c word_tools.c
    gcov -b -n -f -s ../libs/sources/ -o ../libs/$(OBJDIR) -c
serialization.c

# Rule to create all needed directories (-p to disable errors if they
already exist)
ensureDirs:
    @mkdir -p $(OBJDIR)
    @mkdir -p $(EXECDIR)

# Rule to clean the workspace
clean:
    rm -rf $(OBJDIR)/*.o $(EXEC)
    rm -rf $(OBJDIR)/*.gc*
```

Essentiellement, ce Makefile compile simplement les fichiers sources relatifs aux tests en ajoutant les flags nécessaires à l'utilisation de **gcov**, puis effectue l'édition de liens en ajoutant ces mêmes flags. La seule petite subtilité étant la suivante : nous ajoutons les fichiers objets associés aux fonctions que nous testons, c'est-à-dire **word_tools.o** et **serialization.o** au moment de l'édition de lien, afin que le programme de test puisse aller chercher les définitions des fonctions testées.

Cela génère un exécutable **testDico**, que l'on lance directement et que l'on agrémente deux appels à **gcov** afin d'afficher les résultats de l'analyse de couverture pour nos deux fichiers sources testés : **word_tools.c** et **serialization.c**.

Partie 3 : Automatisation des tests

Nous avons réalisé des tests unitaires automatisés de certaines de nos fonctions, ces tests sont réalisés avec pour but de couvrir le maximum des branches et lignes des fonctions, cela permet d'être assuré que l'on n'oublie pas des cas spécifiques. Nous avons fait des tests à partir des spécifications, car bien que l'on ait couvert chaque partie du code, le résultat peut être un résultat non attendu. Pour cela on utilise les fichiers du package **CuTest** donnés en TP, où l'on a modifié le fichier **SuiteDeTests.c** en ajoutant des fonctions pour tester certaines fonctions du projet : les fonctions des fichiers **word_tools.c** et **serialization.c**.

1) Suite de tests sur les mots : TestsWords.c

compareWord :

★ **Paramètres :**
- **w1** la structure du premier mot
- **w2** la structure du second mot

★ **Résultats attendus :** comparaison de 2 mots selon les lettres par ordre ASCII ou selon la taille de leurs chaînes respectives
- **-1** si w1 < w2
- **1** si w1 > w2
- **0** si w1 = w2

★ **Tests réalisés :** (avec **CuAssertIntEquals**)
- Comparaison de 2 mots égaux et on regarde si on a pour résultat 0;
- Comparaison entre un champ NULL et un mot;
- Comparaison de 2 mots différents dont le premier caractère est différent;
- Comparaison de 2 mots différents dont seul le dernier caractère est différent;
- Comparaison de 2 mots de tailles différentes avec les mêmes caractères sur l'entièreté du mot;
- Comparaison de 2 mots dont les tailles diffèrent de 1, tels que tous les caractères sont identiques entre les deux mots, sauf le dernier caractère du mot le plus long;

★ **Défaillances :**
- **Aucune** défaillance constatée.

incWord :

★ **Paramètres :**
- **location** la liste chaînée d'emplacements où insérer le mot;
- **line** l'indice de ligne dans le texte du mot à ajouter, >= 0;
- **colonne** l'indice de colonne dans le texte du mot à ajouter, >= 0;

★ **Résultats attendus :**

- On souhaite ajouter à la fin de la liste chaînée **location** un maillon où on y renseigne la ligne et la colonne données en arguments;

★ **Tests réalisés :**
- On regarde si le pointeur du maillon créé n'est pas NULL (avec **CuAssertPtrNotNull**);
- On vérifie que les champs **line** et **colonne** du maillon correspondent bien aux valeurs données en arguments de la fonction (avec **CuAssertIntEquals**);

★ **Défaillances :**
- **Aucune** défaillance constatée.

next_word :

★ **Paramètres :**
- **f** le fichier contenant le texte d'origine;
- **line** l'indice de ligne actuel dans le texte, modifié par l'indice de ligne du mot si il est trouvé;
- **colonne** l'indice de colonne actuel dans le texte, modifié par l'indice de colonne du mot si il est trouvé;

★ **Résultats attendus :**
- **mot** correspondant au prochain mot trouvé dans **f**;
- On souhaite que les champs **line** et **colonne** soient modifiés par l'indice de ce nouveau mot **mot** trouvé;

★ **Tests réalisés :**
- On a fait des tests sur un nouveau fichier (**test_next_word.txt**) créé spécialement pour gérer le maximum de cas;
  - On a d'abord repris le fichier hugo.txt pour avoir un texte généré en français correct;
  - Puis dans ce fichier on a sauté plusieurs ligne et ajouter plusieurs séparateur au début et au milieu du fichier, pour voir si il les considérait;

- On regarde si le mot renvoyé correspond au mot souhaité (avec **CuAssertStrEquals**);
  - On regarde pour le premier mot du texte;
  - Pour les mots suivant;
  - Pour le premier mot d'une nouvelle ligne;
  - Pour le mot après un séparateur;

- On vérifie que les indices retournés correspondent bien aux indices de ligne et colonne du mot, pour chaque point cité précédemment (avec **CuAssertIntEquals**);

- On a aussi testé avec les pointeurs de **line** et **colonne** à NULL mais on les a mis en commentaires à cause d'une erreur de segmentation qui sera vue pour un autre cours;

★ **Défaillances :**

- On a eu les valeurs de **colonne** trop incrémentées;
  - Le problème venait du fait que l'on incrémenté de 1 à chaque fois que l'on voyait un séparateur et aussi incrémente de 1 pour chaque lettre du mot trouvé;

2) Suite de tests de (dé)sérialisation : TestsSerialization.c

hash :

★ **Paramètres :**

- **m** une chaîne de caractères

★ **Résultats attendus :**

- Le hashcode de la chaîne **m** (entier non signé sur 64 bits)

★ **Tests réalisés :**

- Calcul du hashcode de tous les mots dans le texte **hugo1.txt** via un calculateur en ligne (WolframAlpha) à partir de la formule donnée sur l'énoncé, où **K = 127** et **N = 16381** dans le code fourni :

$$\left(\sum_{i=0}^{len-1} mot[i].K^{len-i-1}\right) mod\ N$$

★ **Défaillances :**

- Le code fourni utilise un entier (type **int**) comme valeur de hashcode ; or un entier étant codé sur 4 octets dans la plupart des systèmes, la valeur maximale (en signé) qu'une telle variable peut contenir est :

$$2^{31} - 1 = 2\,147\,483\,647$$

- Cette valeur est très souvent atteinte, voire même largement dépassée étant donné la formule de calcul du hashcode utilisée ici

- Dans l'espoir de corriger ce défaut, nous avons remplacé le type **int** par un type standard **uint64_t** (entier non signé sur 8 octets). Malheureusement, dans certains cas de mots longs tels que **pervenches**, le calcul excède aussi la limite d'un entier 64 bits.

- En outre, dans certains cas, le calcul du hashcode produisait des résultats légèrement différents (à un ou deux digits près) des résultats donnés par les calculateurs en ligne. Cela s'expliquait par une certaine imprécision de **pow**

pour des flottants très grands. Nous avons donc remplacé l'appel à cette fonction par un appel à **powl** qui travaille sur des **long double**, et permet ainsi une meilleure précision.

serializeDico :

★ **Paramètres :**
- **dictionary** un dictionnaire (arbre binaire de recherche contenant des structures de mots **mot_t** dans ses noeuds) ;
- **table** un tableau de mots contenant des structures de données de mots **mot_data_t**.

★ **Résultats attendus :**
- La structure **table** doit être mise à jour avec l'ensemble des données contenues dans les noeuds de **dictionary** ;
- Les indices de **table** correspondent aux hashcodes des différents mots de **dictionary**, et les valeurs des cases du tableau sont les structures de mots associées (chaîne de caractères + positions des occurrences dans le texte).

★ **Tests réalisés :**
- Nous construisons notre propre dictionnaire sous forme d'ABR, noeud par noeud (voir le schéma en commentaires du fichier source) ;
- Puis nous appelons la fonction **serializeDico** sur ce dictionnaire ;
- Enfin, nous vérifions que les pointeurs contenus dans les cases de la table générée correspondent bien au contenus des nœuds du dictionnaire associés (avec **CuAssertPtrEquals**).

★ **Défaillances :**
- Pas de défaillance identifiée.

deserializeDico :

★ **Paramètres :**
- **dic** un pointeur sur une structure de dictionnaire (ABR) ;
- **elt** une structure de données associée à un mot.

★ **Résultats attendus :**
- La structure **elt** doit être insérée dans le dictionnaire **dic** en respectant les propriétés d'un arbre binaire de recherche.

★ **Tests réalisés :**
- Nous construisons une table de hachage contenant les mots du texte *"bonjour a tous, ceci est un test de deserialisation" ;*
- Nous appelons la fonction **deserializeDico** sur toutes les cases non vides de cette table de hachage ;
- Le parcours de la table étant effectué dans l'ordre croissant des indices, donc des hashcodes, nous savons dans quel ordre seront insérés les mots, donc à

quel ABR nous devons nous attendre en sortie (voir le schéma en commentaires du fichier source) ;
    - Nous vérifions que l'arbre contient effectivement les bonnes chaînes de caractères (ainsi que les bons hashcodes) dans les bons nœuds, via **CuAssertStrEquals, CuAssertIntEquals** et **CuAssertPtrEquals**.

★ **Défaillances :**
    - Pour comparer le mot donné en argument et le mot situé dans le noeud courant de l'arbre, et savoir où effectuer l'insertion, le code fourni effectuait une comparaison entre la structure de donnée contenue dans le noeud courant (**mot_data_t**) et un cast en **mot_data_t** de la structure **newLinkWord** construite par la fonction, de type **mot_t**.
    - Ceci menait à des erreurs d'insertion.
    - Nous avons corrigé le code en remplaçant **newLinkWord** par **elt** dans les comparaisons précédant les appels récursifs, afin de bien comparer deux **mot_data_t**.