TME2 : Langage C: tables de hachage

Objectif

Ce TME se décompose en deux parties: La première partie porte sur l'analyse d'un programme C existant. Dans la seconde partie vous devrez modifier ce programme pour introduire de nouvelles fonctionnalités. L'objectif est triple :

1. Il doit vous permettre de complêtez l'auto-évaluation de vos connaissances des outils de developpement C que vous avez commencée dans le précédent TME. Si vous ne savez pas répondre directement aux questions posées , vous devez trouver les réponses dans les documentations (man, web), ou auprès de vos camarades.
2. Il introduit de nouveaux outils permettant l'indentation automatique d'un programme source (outil *indent*), ou l'écriture d'une documentation (outil *man*).
3. Il présente une structure de données, la table de hachage, très utilisée dans tous les programmes où on a besoin de rechercher un objet par son nom.

Il vous offre également un modèle de programme, avec Makefile et man pour vos futurs développements.

Vous devez commencer par créer un répertoire *tme2* et y copier tous les fichiers se trouvant dans :

```
/users/enseig/encadr/cao/tme2
```

Ce répertoire contient tous les fichiers sources permettant de générer un programme qui utilise une table de hachage pour compter le nombre total de mots d'un fichier texte quelconque, ainsi que le nombre d'occurences de chaque mot.

- `Makefile ..... description du processus de construction de l'exécutable.`
- `main.c, main.h ..... programme principal source et déclarations.`
- `count.c, count.h ..... analyseur lexical et fonctions de comptage et d'affichage du résultat.`
- `dico.c, dico.h..... fonctions de gestion de la table de hachage.`
- `hash.c, hash.h..... fonction générale de calcul de l'index à partir de la clé de hachage.`
- `man1/tool.1 ..... fichier au format man`

Principe des tables de hachage

Une table de hachage est une structure de données permettant de représenter des ensembles d'éléments, où chaque élément est un couple de la forme (clé, donnée). La clé est le plus souvent une chaîne de caractères. La donnée peut être une valeur ou un pointeur sur une structure plus ou moins complexe. Le principal objectif d'une table de hachage est d'accélérer la recherche d'un élément par sa clé.

Pour représenter un ensemble de couples (clé, data) la méthode la plus simple consiste à les stocker dans une unique liste chainée. La recherche d'un élément se fait alors par un parcours de la liste, ce qui n'est pas très efficace si le nombre d'éléments est grand.

Pour accélérer la recherche, on créé un tableau de listes chainées. On définit ensuite une fonction, que l'on nomme **fonction de hachage**, qui calcule un index à partir de la clé. Tout élément doit être rangé dans la liste chainée associée à la case du tableau définie par l'index calculé par la fonction de hachage.

Dans la pratique, il n'est pas possible d'éviter les collisions: deux éléments ayant des clés différentes peuvent avoir le même index de hachage, et seront donc stockés dans la même liste chaînée. Pour que la méthode soit efficace, il faut cependant que les éléments se répartissent aussi uniformément que possible dans les différentes cases du tableau, et qu'il n'y ait qu'un petit nombre d'éléments dans chaque liste chainée. Le nombre de cases du tableau doit donc être du même ordre de grandeur que le nombre total d'éléments à stocker.

Il existe plusieurs méthodes de calcul de l'index. La fonction hashindex() contenue dans le fichier hash.c implémente celle proposée par Donald Knuth.

La propriété principale d'une table de hachage est que, si la taille de la table est du même ordre de grandeur que le nombre d'éléments, alors le temps de recherche est en O(1), c'est à dire indépendant du nombre d'éléments, même pour un million d'éléments, comme s'il s'agissait d'un simple tableau.

Etape 1 : Questions sur le code fourni

A) Le Makefile

Les premières questions portent sur le fichier Makefile

Completez la liste des dépendances pour les cibles : `statt, main.o, ...` , puis re-écrivez les commandes en utilisant les variables automatiques : `$@ $< $^`

- `$@` : désigne le fichier cible d'une règle.
- `$<` : désigne le premier fichier de la liste des fichiers source d'une règle.
- `$^` : désigne la liste des fichiers source d'une règle.

- **QA1** Pourquoi est-il préférable de regrouper la definition des commandes et paramètres au début du Makefile?
- **QA2** A quoi servent les options -p, -g, -Wall, -Werror, -ansi ? (man gcc)
- **QA3** Comment demander l'optimisation maximale du compilateur ? (man gcc)
- **QA4** L'option -p est présente dans LDFLAGS et CFLAGS, pourquoi n'est-ce pas le cas de -g ? (man gcc)
- **QA5** Que fait la regle indent ? quelle est la signification des flags utilisés par le programme indent ? (man indent)

B) Le programme main

Les questions suivantes portent sur le programme principal main.c Ce fichier contient la fonction main() et la fonction getarg() qui effectue l'analyse de la ligne de commande.

- **QB1** Expliquez à quoi sert chacun des fichiers inclus au début du fichier *main.c*
- **QB2** A quoi sert le fichier *main.h* ? A quoi servent les 2 premières lignes et la dernière du fichier *main.h* Pourquoi inclure `stdio.h` ici ?
- **QB3** Expliquez le fonctionnement de la fonction getopt() (`man 3 getopt`)

  > Vous ajouterez plus tard dans la fonction getarg() l'option -s qui demande de fournir des statistiques concernant l'utilisation de la tables de hachage.

- **QB4** A quoi sert l'appel *return* a la fin de la fonction main() ?
- **QB5** A quoi sert l'appel système *exit* à la fin de la fonction usage() ?
- **QB6** Quels sont les appels systeme utilisés dans ce fichier main.c ? Quelle precaution doit on prendre lors de leur utilisation ?
- **QB7** Où sont definies les fonctions standards de la bibliothèque C ?
- **QB8** Qu'est-ce qu'un filtre unix ? Que faut-il faire pour transformer ce programme en filtre ?

C) Le dictionnaire

Le fichier dico.c rassemble les fonctions d'accès à une table de hachage utilisée comme dictionnaire.

- **QC1** Quel est l'encombrement (en nombre d'octets) des structures créées en mémoire par la fonction dico_create(nb_item) , quand nb_item=10 ? Quelle différence y-at-il entre les appels système malloc() et calloc() ?
- **QC2** Pourquoi la structure `dico_item_t` définie dans le fichier dico.h a-t-elle un encombrement mémoire variable ? On aurait pu utiliser une structure de donnée de taille fixe en définissant le troisième champs de la structure comme un pointeur sur chaîne de caractères du type `char *KEY`. Pourquoi n'a-t-on pas utilisé cette technique ?
- **QC3** Quelle est la seule fonction capable d'introduire un nouvel item dans le dictionnaire? Quelle est la technique utilisée dans la fonction dico_get() pour créer ce type d'objet de taille variable en mémoire ?
- **QC4** Dans la fonction dico_get() comment fait-on pour tester le retour de l'appel système malloc() ? à quoi celà sert-il ? Pourquoi teste-on la valeur de l'argument *key* au début de cette fonction ? Que fait la fonction perror() ?
- **QC5** A quoi sert la structure `dico_iterator_t` définie dans le fichier dico.h ? Quelle est la signification des trois champs de cette structure ? Que font les fonctions dico_first() et dico_next() ?
- **QC6** La fonction hashindex() peut-elle provoquer une erreur de segmentation ? Comment y remèdier proprement ?

D) La fonction de comptage des mots

Le fichier count.c contient le code de la fonction count(), qui compte le nombre d'occurences des différents mots présents dans le fichier texte analysé. Il contient également la fonction auxiliaire token() qui lit le fichier texte mot par mot, et la fonction result_count() qui affiche les résultats.

- **QD1** Le mot clé *static* est utilisé de deux manières différentes dans le fichier *count.c* (à l'intérieur et à l'extérieur d'une fonction). Précisez sa signification pour chacune des deux utilisations.
- **QD2** Dans la fonction token(), pourquoi ne peut-on pas utiliser l'appel système malloc() pour allouer la mémoire correspondant au tableau *buffer* ?

- **QD3** La fonction token() doit renvoyer un nouveau "token" (mot) du fichier texte analysé, ainsi que le numéro de ligne, chaque fois qu'elle est appelée. Elle utilise les fonctions fgets() et strtok(). Que font ces deux fonctions ?
- **QD4** Pourquoi as-ton mis une étoile devant l'argument *numero* de la fonction token() ?
- **QD5** La fonction result_count() utilise-t-elle des fonctions d'accès spécifiques pour effectuer le parcours des éléments présents dans la table de hachage ? Quelles sont ces fonctions d'accès et que font-elles ? Dans quel fichier sont-elles définies? Dans quel ordre vont être affichés les élément de la table ?

Etape 2 : Modifications du programme

A) affichage des numéros de ligne

Le programme qui vous est fourni affiche, pour chaque mot présent dans le fichier texte analysé, le nombre d'occurences de ce mot. Vous devez modifier le programme de façon à ce qu'il indique en plus, pour chaque mot, les numéros de toutes les lignes où le mot est présent.

Le fichier Makefile est un fichier texte. Si on lance le programme `statt` sur le fichier Makefile, on obtient quelque chose comme :

```
$ ./statt Makefile
              dico.o : 2 occurences
                 des : 3 occurences
                   : : 2 occurences
               clean : 3 occurences
           /dev/null : 2 occurences
                  2> : 2 occurences
             count.o : 2 occurences
                   # : 3 occurences
```

Après modification, il devra afficher:

```
$ ./statt Makefile

              dico.o : 2 occurences lignes: 23  22
                 des : 3 occurences lignes: 14   8   2
                   : : 2 occurences lignes: 22  19
               clean : 3 occurences lignes: 37  34
           /dev/null : 2 occurences lignes: 37  34
                  2> : 2 occurences lignes: 37  34
             count.o : 2 occurences lignes: 20  19
                   # : 3 occurences lignes: 14   8   2
```

Pour introduire cette nouvelle fonctionnalité, il faut:

- Définir un type de liste chainée pour le stockage des numéros de ligne. Cette structure contient deux champs: un pointeur vers l'élément suivant de la liste et un entier représentant le numéro de ligne.
- Changer la structure dico_item_s afin d'ajouter un champs contenant un pointeur sur la liste chainée contenant les numéros de lignes.
- Modifier la fonction count() pour ajouter un nouveau numéro de ligne dans la liste chaînée associée à l'item chaque fois que la fonction token() renvoie un mot.
- Mofifier la fonction result_count() pour parcourir les listes chaînées contenant les numéros de ligne et les afficher.

B) statistiques sur la table de hachage

Vous donnerez également des statistiques sur l'usage de la table de hachage. Vous fabriquerez pour cela une fonction `void dico_info(dico_root_t *)` dans le fichier `dico.c` qui affichera quelque-chose comme:

```
Informations sur la table de routage
Taux de remplissage          : 79.21 %
Longueur moyenne des listes  : 1.40
Longueur maximale des listes : 4
```

C) création d'un manuel en ligne

Vous devez enfin écrire la page de manuel pour le programme *statt*. Le fichier tool.1 se trouve dans le répertoire man1, et contient une page de manuel générique contenant des commentaires pour expliquer la syntaxe. La structure de cette page est standard, c'est celle utilisée pour les commandes unix.

Renommez le fichier *tool.1* en *statt.1*, et modifiez son contenu pour documenter le programme *statt*.

Pour rendre ce manuel utilisable en ligne, ajoutez le répertoire `.` à la variable d'environnement `MANPATH`:

```
export MANPATH=.:$MANPATH
```

Il suffit de taper la commande `man statt` dans le répertoire tme2 pour afficher la documentation.

Compte-Rendu

Pour la première partie de ce TME, vous rédigerez un compte-rendu contenant les réponse aux questions posées, en respectant la numérotation.

Pour la seconde partie, une démonstration des modifications introduites dans le programme vous sera demandée au début du prochain TME.