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 un programme utilisant 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.

Description des sources et principe des tables de hachage

- `Makefile ..... description du processus de construction de l'exécutable.`
- `main.c, main.h ..... programme principal source et déclarations.`
- `count.c, count.h ..... algorithme de parcours d'un fichier texte en vue de comptage.`
- `hte.c, hte.h ..... fonction de création des tables de hachage et déclaration de toutes fonctions de gestion.`

- `dico.c, dejavu.c, namealloc.c ..` fonctions de gestion des tables pour trois types d'usage
- `man1/tool.1 ....................` fichier au format man

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, data). Le plus souvent la clé est une chaîne de caractères. La donnée peut être un nombre ou un pointeur sur une structure de données 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 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 d'une case 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 case du tableau. Le nombre de case du tableau doit donc être du même ordre de grandeur que le nombre total d'éléments à stocker. De plus, on choisit généralement un nombre premier pourle nombre de cases du tableau.

Il existe plusieurs méthodes de calcul de l'index, nous vous donnons 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.

Etape 1 : Questions sur le code fourni

a) Le Makefile

Les premières questions portent sur le fichier attachment:"Makefile"

1. Completez la liste des dépendances pour les cibles : `main.o ... namealloc.o`.
2. Réécrivez 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.
3. Pourquoi est-il préférable de regrouper la definition des commandes et paramètres au début du Makefile?
4. A quoi servent les options -p, -g, -wall, -werror, -ansi ? (man gcc)
5. Comment demander l'optimisation maximale du compilateur ? (man gcc)
6. L'option -p est présente dans LDFLAGS et CFLAGS, pourquoi n'est-ce pas le cas de -g ? (man gcc)
7. 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 attachment:main.c Ce fichier contient la fonction main() et la fonction getarg() qui effectue l'analyse de la ligne de commande.

- Expliquez à quoi sert chacun des fichiers inclus au début du fichier *main.c*
- A quoi sert le fichier *main.h* ?
- 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.

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

Questions sur le fichier de prototype associé attachment:main.h

- A quoi servent les lignes les 2 premières lignes et la dernière ?
- Pourquoi inclure `stdio.h` ici ?

c) La table de hachage générique

Le fichier attachment:hte.h contient les déclarations des fonctions de base et des types de données permettant de manipuler une table de hachage générique possédant la structure décrite ci-dessous:

- Les types `hte_item_t` et `hte_data_t` sont des structures dont le contenu n'est pas défini dans les fichiers *hte.h* ou *hte.c*. En effet, la table de hachage définie dans ces fichiers est une ressource "générique", qui peut être utilisées pour stocker différents types d'objets. Par conséquent les types `hte_item_t` et `hte_data_t` peuvent (et doivent) être définis en fonction de l'utilisation qu'on souhaite. Les 3 fichiers *dico.c*, *dejavu.c* et *namealloc.c* définissent trois utilisations différentes d'une table de hachage. Quelle est la contrainte d'usage de ces types dans le fichier *hte.c* ?
- Dans la définition des prototypes de fonctions, le nom des paramètre est-il nécessaire ? si non pourquoi les mettre ?

questions sur le fichier attachment:hte.c

- Quel est l'encombrement mémoire (en nombre d'octets) de la structure `root` créée par la fonction hte_create() , quand `nb_list==10`.
- La fonction hte_hash() peut-elle provoquer une erreur de segmentation ? Comment y remedier proprement ?
- Dans la fonction hte_create() comment fait-on pour tester le retour de l'appel système malloc() ? à quoi celà sert-il ?

d) Le dictionnaire

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

- Pourquoi la structure de donnée `hte_item_s` 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`. Pour quoi n'a-t-on pas utilisé cette technique ?
  - ♦ Quelle est la technique utilisée dans la fonction hte_add() pour créer ce type d'objet de taille

variable en mémoire ?
- La structure de donnée `hte_data_s` n'est pas définie dans le fichier *dico.c*. Pourquoi ? Où devra-t-il être défini ? quel genre d'information devra-t-elle contenir ?
- Pourquoi teste-on la valeur de l'argument *key* dans la fonction hte_get() ? A quoi sert la fonction perror() ?

e) La fonction de comptage des mots

Le fichier attachment: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.

- Le mot clé **static** est utilisé de trois manières différentes dans le fichier *count.c*. Précisez sa signification pour chacune des trois utilisations.
- Dans la fonction token(), pourquoi ne peut-on pas utiliser l'appel système malloc() pour allouer la mémoire correspondant au tableau *buffer*, en écrivant par exemple `char * buffer = malloc(1024);` ?
- 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 ?
- Pourquoi as-ton mis une étoile devant l'argument *numero* de la fonction token() ?
- Pourquoi la fonction result_count utilise-t-elle des fonctions d'accès spécifiques pour effectuer le parcours des éléments présents dans la table de hachage ? Dans quel ordre vont être affichés les élément de la table ?

f) Autres services utiles

Vous trouverez deux autres services possibles utilisant des tables de hachage dans les fichiers, attachment:dejavu.c et attachment:namealloc.c. Ces services ne sont pas utilisés dans ce TME2, mais ils vous seront utiles lors des TME futurs. Vous pourrez faire alors une édition de liens avec la bibliothèque libhte.a.

La fonction dejavu() permet de répondre à la question "ai-je déjà vu cette chaine de caractères". La fonction namealloc() est très utilisée dans la chaîne de CAO Alliance, et permet de garantir que si deux chaines de caractères sont identiques alors elles seront rangées à la même adresse.

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 :

```
$ ./statt Makefile
          namealloc.o : 2 occurences
                hte.o : 2 occurences
                true : 2 occurences
               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
         Definition : 3 occurences
           dejavu.o : 2 occurences
                 -p : 2 occurences
                 || : 2 occurences
             indent : 2 occurences
                  = : 9 occurences
              $(RM) : 2 occurences
              statt : 4 occurences
             main.o : 2 occurences
                 de : 2 occurences
           libhte.a : 3 occurences
116 mots dont 80 différents
```

Après modification, il devra afficher:

```
$ ./statt Makefile
         namealloc.o : 2 occurences lignes: 23  22
               hte.o : 2 occurences lignes: 23  22
                true : 2 occurences lignes: 37  34
              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
          Definition : 3 occurences lignes: 14   8   2
            dejavu.o : 2 occurences lignes: 23  22
                  -p : 2 occurences lignes: 10   9
                  || : 2 occurences lignes: 37  34
              indent : 2 occurences lignes: 17   8   2
                   = : 9 occurences lignes: 15  12  11  10   9   6   5   4   3
               $(RM) : 2 occurences lignes: 37  34
               statt : 4 occurences lignes: 37  34
              main.o : 2 occurences lignes: 20  19
                  de : 2 occurences lignes: 14  14
            libhte.a : 3 occurences lignes: 23  22
116 mots dont 80 différents
```

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 hte_data_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 au champs 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 hte_info(hte_root_t *)` 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 attachment: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.

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