TME2 : Langage C: étude de cas sur les tables de hachage

Objectif

Pour développer une application en C, vous devez savoir:

- Ecrire un programme C en respectant des conventions d'écriture.
- Compiler en plusieurs fichiers objet et construire une librairie.
- Ecrire un Makefile.
- Debugger en utilisant gdb ou xgdb.
- Faire des mesures de performances avec gprof.
- Ecrire un man sur l'outil.

L'objectif de ce TME est double :

1. Il doit d'une part vous permettre de complêtez l'auto-évaluation de vos connaissances des outils de developpement C que vous avez commencée dans le précédent TME, en vous posant des questions auxquelles vous devriez savoir répondre. Si ce n'est pas le cas, 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*), la constructtion d'une bibliothèque C (outil *ar*), ou l'écriture d'une documentation (outil *man*).

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

Travail demandé

- Vous devez commencer par copier sur votre compte le répertoire :

```
cp -rp /users/enseig/encadr/cao/tme2 ~/cao/tme2
```

Ce répertoire contient un programme utilisant une table de hachage. Le travail demandé comporte deux phases. Dans un premier temps vous devez analyser le code fourni, et **rédiger** des réponses aux questions portant sur ce code. Dans un deuxième temps, vous devrez modifier ce programme, pour introduire de nouvelles fonctionnalités.

Le programme fourni compte le nombre de mots d'un fichier texte et indique le nombre total de mots dans le fichier et le nombre de mots différents. Vous devrez modifier ce programme de façon à ce qu'il indique, pour chaque mot:

- le nombre d'occurences
- les numéros de toutes les lignes où il est présent

Vous donnerez également des statistiques sur l'usage des tables de hachage:

- taux de remplissage.
- moyenne du nombre de comparaisons nécessaire lors de la recherche d'un 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 déclaration.`
- `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 stocker des ensembles d'éléments, où chaque élément est un couple de la forme (clé, valeur). Le plus souvent la clé est une chaîne de caractères. La valeur peut être un nombre ou une structure de données quelconque. Le principal objectif de cette structure est d'accélérer la recherche d'un élément par sa clé.

Pour représenter un ensemble de couple (clé, valeur) la méthode la plus simple consiste à les réunir dans une liste chainée. La recherche d'un élément se fait alors par un parcours de la liste, donc de l'ensemble des éléments. Cette solution n'est évidemment pas satisfaisante si le nombre d'élément est grand.

Pour accélérer la recherche, on créé un tableau de listes chainées. Le nombre de cases de ce tableau doit être de l'ordre de grandeur du nombre maximal d'éléments que l'on souhaite gérer. Si on veut gérer 1000 éléments, il faut un tableau d'environ 1000 cases. On définit ensuite une fonction, que l'on nomme fonction de hachage, donnant un index de case à partir de la clé d'un élément. Un élément doit être rangé dans la liste chainée associée à la case définie par la fonction de hachage appliquée sur la clé de cet élément.

Pour que cette méthode soit efficace, il faut que les éléments se répartissent aussi uniformément que possible dans les différentes cases de la table. Il existe plusieurs méthodes de calcul de l'index, nous vous donnons celle proposé par Donald Knuth. Dans la pratique, il n'est pas possible d'éviter les collisions. Cela signifie que deux éléments de clés différentes peuvent avoir le même index de hachage.

En résumé, pour ranger 1000 éléments, on fabrique un tableau de 1000 listes chainées avec l'espoir que la plupart des listes ne contiendront qu'un seul élément et qu'il n'y aura qu'un tout petit nombre de listes contenant plus d'un élément.

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 présents, le temps de recherche est en O(1), c'est à dire indépendant du nombre d'éléments, (même pour un million d'éléments). Les deux principales actions sont l'ajout d'un élément (add) et la recherche d'un élément (get).

- La fonction get() prend en paramètre la clé de l'élément recherché. Si l'élément existe, elle rend la valeur associée.
- La fonction add() prend en paramètre le couple (clé, valeur). Si l'élément existe, elle change sa valeur, sinon elle créé l'élément.

Questions sur le code fourni

Le Makefile

Les premières questions portent sur le fichier attachement:"file 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. Donnez une raison à la definition des commandes et paramètres au début du Makefile
4. A quoi servent les options -p, -g, -wall, -werror, -ansi ?
5. Comment demander l'optimisation maximale du compilateur ?
6. L'option -p est présente dans LDFLAGS et CFLAGS, pourquoi n'est-ce pas le cas de -g ?
7. Que fait la regle indent ? quelle est la signification des flags utilisés par le programme indent ?

Le programme main

Les questions suivantes portent sur le programme principal main()

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

```
#ifndef _MAIN_H_
#define _MAIN_H_

#include <stdio.h>

struct mainarg_s
{
    FILE *INFILE;
    FILE *OUTPUTFILE;
    char VERBOSE;
};
extern struct mainarg_s MAINARG;

#endif
```

Fichier main.c

- A quoi sert chaque include ?
- Pourquoi a-t-on un fichier main.h ?
- Expliquez le fonctionnement de la fonction getopt (`man 3 getopt`)

> Ajoutez l'option -h qui affiche l'usage du programme et un petit texte de description du comportement (très court, c'est juste pour l'exercice).
> Vous ajouterez plus tard l'option -s qui demande les statistiques d'usage de la tables de hachage.

- A quoi sert l'appel de return a la fin de la fonction main() ?
- Pourquoi y-a-t-il exit() a la fin de la fonction usage() ?
- Qu'est ce qu'un appel systeme, en voyez-vous dans ce fichier, si oui lesquels,

citez en d'autres.

- Quelle precaution doit on prendre lors de leur utilisation ?
- Ou sont definiies les fonctions standards ?
- Qu'est-ce qu'un filtre unix ?
- Que faut-il faire pour transformer ce programme en filtre ?

```
include <stdlib.h>
#include <stdio.h>
#include <getopt.h>
#include "main.h"
#include "count.h"
#include "hte.h"

struct mainarg_s MAINARG;

void usage (char *command)
{
    printf ("\nStatistique Diverses\n");
    printf ("Usage  : %s [-v] [-o OutFile] InFile\n", command);
    printf ("-v          verbose mode\n");
    printf ("-o OutFile  output file (stdout by default)\n\n");
    exit (EXIT_FAILURE);
}

void getarg (int argc, char **argv)
{
    extern char *optarg;
    extern int optind;
    char option;

    MAINARG.OUTPUTFILE = NULL;
    while ((option = getopt (argc, argv, "vo:")) != EOF)
        switch (option)
        {
        case 'v':
            MAINARG.VERBOSE = 1;
            fprintf (stderr, "Verbose mode\n");
            break;

        case 'o':
            if (MAINARG.VERBOSE)
                fprintf (stderr, "Fichier de sortie : %s\n", optarg);

            MAINARG.OUTPUTFILE = fopen (optarg, "w");
            if (MAINARG.OUTPUTFILE == NULL)
            {
                fprintf (stderr, "%s: %s: \n", argv[0], optarg);
                perror ("fopen");
                exit (EXIT_FAILURE);
            }
            break;

        case '?':
        default:
            usage (argv[0]);
        }
    if ((optind + 1) != argc)
    {
        usage (argv[0]);
    }
    else
    {
        if (MAINARG.VERBOSE)
```

```
            fprintf (stderr, "Fichier d'entrée : %s\n", argv[optind]);

        MAINARG.INFILE = fopen (argv[optind], "r");
        if (MAINARG.INFILE == NULL)
        {
            fprintf (stderr, "%s: %s ", argv[0], argv[optind]);
            perror ("fopen");
            exit (EXIT_FAILURE);
        }
        if (MAINARG.OUTPUTFILE == NULL)
            MAINARG.OUTPUTFILE = stdout;
    }
}

int main (int argc, char **argv)
{
    getarg (argc, argv);
    count (MAINARG.INFILE);
    fclose (MAINARG.OUTPUTFILE);
    return EXIT_SUCCESS;
}
```

Les fonctions de bases de la tables de hachage

Fichier hte.h

- Les types `hte_item_t` et `hte_data_t` sont des structures dont le contenu n'est pas défini ici.

  > Leur contenu n'est pas défini non plus dans le fichier `hte.c`, en revanche il est défini dans les fichiers `dico.c`, `dejavu.c` et `namealloc.c` Quelle est alors la contrainte d'usage de ces types dans le fichier `hte.c` ? Quel est l'intérêt de cette écriture ?

- Dans la définition des prototypes de fonctions, le nom des paramètre est-il nécessaire ? si non pour les mettre ?

```
#ifndef _HTE_H_
#define _HTE_H_

#include <stdlib.h>

/*
 * Les deux structures ci-dessous ne sont pas définies ici
 * elles seront redéfinies dans chaque fichier
 */
typedef struct hte_item_s hte_item_t;
typedef struct hte_data_s hte_data_t;

/*
 * structure définissant la table de hachage
 * On y trouve le nombre de liste de liste et
 * un pointeur vers le tableau de liste
 */
typedef struct hte_root_s
{
    size_t NB_LIST;             /* nombre de liste d'ITEM du dictionnaire */
    hte_item_t **LIST;          /* pointeur sur un tableau de liste d'ITEM */
} hte_root_t;

/*
 * Calcul de l'index de hachage de Donald Knuth
```

```
 * Elle produit un nombre entier à partir d'une chaine de caractères.
 */
extern unsigned hte_hash (char *key);

/*
 * hte_create fabrique une table de hachage vide
 * parametres:
 *      nb_item = nombre d'item maximum prévu dans la table de hachage
 */
extern hte_root_t *hte_create (size_t nb_item);

/*
 * hte_add ajoute l'item (key,data)
 * parametres:
 *      root = pointeur sur la table de hachage
 *      key  = pointeur sur une chaine de caractères (la clé)
 *      data = donnée associée à key
 * comportement:
 *      La clé key passée en parametre est comparée à celles présentes dans
 *      la table. Si la clé n'est pas trouvée alors un nouvel item est créé
 *      avec le couple (key,data). La clé key est en fait duppliquée.
 *      Si la clé est trouvée, le champ DATA de l'item présent dans la
 *      table est remplacé par le paramètre data de la fonction.
 * retour:
 *      Si l'item n'a pu etre créé faute d'espace, on sort du programme.
 */
extern void hte_add (hte_root_t * root, char *key, hte_data_t * data);

/*
 * hte_get recherche item, et en cas d'absence l'ajoute
 * parametres:
 *      root = pointeur pour la table de hachage
 *      key  = pointeur sur une chaine de caractères (la clé)
 * comportement:
 *      La clé passée en parametre est comparé a celles présentes dans la
 *      table de hachage.
 * retour:
 *      Si la clé est trouvée alors la fonction retourne le champ data
 *      sinon la fonction rend NULL.
 */
extern hte_data_t *hte_get (hte_root_t * root, char *key);

/*
 * namealloc permet de garantir l'unicité des chaines de caractères
 * parametres:
 *      name = chaine de caractères
 * comportement:
 *      name est recherché dans la table,
 *      s'il est absent on ajoute une copie de name dans la table
 * retour:
 *      un pointeur sur la copie de la table
 */
char *namealloc (char *name);

/* nombre maximum de noms dans la table */
#define MAX_NAMEALLOC 1000

/*
 * dejavu permet de savoir si un nom a déjà été vu
 * parametres:
 *      name = chaine de caractères
 * comportement:
 *      name est recherché dans la table,
 *      s'il est absent on ajoute une copie de name dans la table
 * retour:
```

```
 *      rend 1 s'il était déjà présent sinon 0
 */
unsigned dejavu (char *name);

/* nombre maximum de noms dans la table */
#define MAX_DEJAVU 1000

#endif
```

Fichier hte.c

- Faîte un dessin représentatif de la valeur de la variable locale `root` de la fonction `hte_create` si `nb_list==10` juste avant l'appel return.
- La fonction `hte_hash` peut-elle provoquer une erreur de segment ? Comment y remedier proprement ?
- Dans la fonction `hte_create` comment fait-on pour tester le retour de `malloc` ? à quoi celà sert-il ?

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "hte.h"

/*
 * Calcul de l'index de hachage de Donald Knuth
 * Elle produit un nombre entier à partir d'une chaine de caractères.
 */
unsigned hte_hash (char *key)
{
    char *c = key;
    unsigned h;
    for (h = 0; *c; c++)
    {
        h += (h ^ (h >> 1)) + 314159 * (unsigned char) *c;
        while (h >= 516595003)
            h -= 516595003;
    }
    return h;
}

/*
 * Création d'un dictionnaire vide
 */
hte_root_t *hte_create (size_t nb_item)
{
    hte_root_t *root;
    unsigned i, nb_list;
    static unsigned primes[] = { /*suite croissante de nombres premiers */
        101, 149, 223, 347, 499, 727, 1163, 1697, 2503, 3989, 5839, 8543,
        12503, 20143, 29483, 43151, 63179, 92503, 101747, 148961, 218107,
        319313, 514243, 752891, 1212551, 1613873, 2599153, 3805463,
        5571523, 8157241, 11943011, -1
    };

    /* détermination de la taille réelle du dictionnaire */
    for (i = 0; primes[i] < nb_item; i++);
    nb_list = primes[i];
    if (nb_item != -1)
    {
        /* allocation et initialisation du dictionnaire */
        if ((root = malloc (sizeof (hte_root_t))))
        {
            root->NB_LIST = nb_list;
            if ((root->LIST = calloc (nb_list, sizeof (hte_item_t *))))
                return root;
        }
```

```
    }
    fprintf (stderr, "hte_create: not enough memory\n");
    exit (1);
}
```

Le service *dejavu*

Fichier dejavu.c

- Pourquoi définit on la structure `hte_item_s` ici ?
- Dans la structure `hte_item_s` le champ KEY est un tableau de taille indéfinie.

  Quel différence y aurait-il avec une définition du type `char *KEY` ?

- La variable `root_namealloc` est static. Qu'est-ce que cela veut dire ?
- Donner les autres usages du mots clé static en langage C.
- A quoi sert la fonction `perror` ?

```
#include "hte.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

/*
 * structure définissant un item du dictionnaire
 * Un item est formé d'un couple (clé,valeur) plus un pointeur
 * permettant de chainer les items ayant le meme index de hachage
 */
struct hte_item_s
{
    struct hte_item_s *NEXT;    /* pointeur sur l'item suivant */
    char KEY[];                 /* tableau flexible contenant la clé */
};

unsigned dejavu (char *key)
{
    unsigned index;
    hte_item_t *curr;
    static hte_root_t *root_namealloc;

    if (key)
    {
        /* création du dictionnaire la première fois */
        if (root_namealloc == NULL)
            root_namealloc = hte_create (MAX_NAMEALLOC);

        /* calcul de l'index pour trouver la liste ou devrait être l'item */
        index = hte_hash (key) % root_namealloc->NB_LIST;

        /* recherche et l'item dans sa liste et sortie si trouvé */
        for (curr = root_namealloc->LIST[index]; curr; curr = curr->NEXT)
            if (strcmp (curr->KEY, key) == 0)
                return 1;

        /* création d'une nouvelle entrée */
        curr = malloc (sizeof (struct hte_item_t *) + strlen (key) + 1);
        if (curr)
        {
            strcpy (curr->KEY, key);
            curr->NEXT = root_namealloc->LIST[index];
            root_namealloc->LIST[index] = curr;
            return 0;
```

```
        }
    }
    perror ("dejavu");
    exit (1);
}
```

Le service *dictionnaire*

Fichier dico.c

- Le type `hte_data_t` n'est pas défini ici. Est-ce grave ? Où devra-t-il être défini ?
- Pourquoi teste-on key ici ?

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "hte.h"

/*
 * structure définissant un item du dictionnaire
 * Un item est formé d'un couple (clé,valeur) plus un pointeur
 * permettant de chainer les items ayant le meme index de hachage
 */
struct hte_item_s
{
    struct hte_item_s *NEXT;    /* pointeur sur l'item suivant */
    hte_data_t *DATA;           /* valeur associée à l'item */
    char KEY[];                 /* tableau flexible contenant la clé */
};

/*
 * Recherche d'un item de clé key dans le dictionaire root
 */
hte_data_t *hte_get (hte_root_t * root, char *key)
{
    int index;
    hte_item_t *curr;

    if (key)
    {
        /* calcul de l'index pour trouver la liste ou devrait être l'item */
        index = hte_hash (key) % root->NB_LIST;

        /* recherche et l'item dans sa liste et sortie si trouvé */
        for (curr = root->LIST[index]; curr; curr = curr->NEXT)
            if (strcmp (key, curr->KEY) == 0)
                return curr->DATA;

        return NULL;
    }
    perror ("hte_get");
    exit (1);
}

/*
 * Recherche d'un item de clé key dans le dictionnaire root
 * avec création en cas d'absence
 * et affectation d'une nouvelle donnée data
 */
void hte_add (hte_root_t * root, char *key, hte_data_t * data)
{
    int index;
    hte_item_t *curr;
```

```
    if (key)
    {
        /* calcul de l'index pour trouver la liste ou devrait être l'item */
        index = hte_hash (key) % root->NB_LIST;

        /* recherche et l'item dans sa liste et sortie si trouvé */
        for (curr = root->LIST[index]; curr; curr = curr->NEXT)
        {
            if (strcmp (curr->KEY, key) == 0)
            {
                curr->DATA = data;
                return;
            }
        }

        /* création d'une nouvelle entrée */
        curr = malloc (sizeof (struct hte_item_t *) + sizeof (void *) + strlen (key) + 1);
        if (curr)
        {
            strcpy (curr->KEY, key);
            curr->NEXT = root->LIST[index];
            root->LIST[index] = curr;
            curr->DATA = data;
            return;
        }
    }
    perror ("hte_add");
    exit (1);
}
```