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

Objectif

Pour la majorité d'entre-vous, vous connaissez déjà le C, mais certains ne le connaissent que superficiellement. Nous devons essayer de mettre tout le monde au niveau, en vous faisant étudier un petit programme. L'objectif de ce programme est double :

1. Il doit d'une part vous permettre de faire une auto-évaluation de vos connaissances des outils de developpement C 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 vous offre un modèle de programme, avec makefile et man pour vos futurs développements.

Pour réaliser une application en C, vous devez savoir:

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

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 consiste:
    1. à répondre aux questions portant sur le code fourni. Les questions sont sur cette page. Vous rédigerez un compte rendu informatique pour vous même avec les réponses.
    2. à programmer des évolutions du programme:
- L'évaluation sera individuelle et orale au début du tme3.
- Commencez par lire le programme en entier et faîtes le tourner pour comprendre son fonctionnement.
- Répondez ensuite aux questions et faîtes les évolutions demandées.

Evolution du programme

Le programme fourni compte le nombre de mots d'un fichier texte et indique le nombre de mots présents et le nombre de mots différents. Votre programme devra indiquer pour chaque mot:

- le nombre d'occurences
- les numéros de 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 fournies

- `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`

Questions

Le processus de construction : Makefile

1. Completez la liste des dépendances lignes 24 à 28.
2. Réécrivez les commandes en utilisant les variables automatiques : `$@ $< $^`
    - `$@` : désigne la cible d'une règle.
    - `$<` : désigne le premier fichier de la liste des sources d'une règle.
    - `$^` : désigne la liste des sources 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 ?

Fichier Makefile

```
 1 # Definition des commandes
 2 CC      = gcc
 3 AR      = ar
 4 RM      = rm
 5 INDENT  = indent
 6
 7 # Definition des parametres
 8 LDFLAGS = -p
 9 CFLAGS  = -g -p -Wall -ansi -Werror
10 ARFLAGS = -r
11 IDFLAGS = -gnu  -bli0 -npsl -l90
12
```

```
13 # Definition de la liste des librairies necesaires a l'edition de lien
14 LDLIBS  = -L. -lhte
15
16 .PHONY: all clean realclean
17
18 stat : main.o count.o libhte.a
19         $(CC) $(LDFLAGS) main.o count.o -o stat $(LDLIBS)
20
21 libhte.a : hte.o dico.o dejavu.o namealloc.o
22         $(AR) $(ARFLAGS) libhte.a hte.o dico.o dejavu.o namealloc.o
23
24 main.o:
25 count.o:
26 hte.o:
27 dejavu.o:
28 namealloc.o:
29
30 all: clean stat
31
32 clean:
33         $(RM) *.o *.a *.out *~ 2> /dev/null || true
34
35 realclean: clean
36         $(RM) stat 2> /dev/null || true
37
38 indent:
39         $(INDENT) $(IDFLAGS) *.c *.h
```

le programme principal : main

- A quoi servent les lignes 1, 2 et 11 ?
- Pourquoi inclure `stdio.h` ici ?

Fichier main.h

```
 1 #ifndef _MAIN_H_
 2 #define _MAIN_H_
 3
 4 #include <stdio.h>
 5
 6 struct mainarg_s
 7 {
 8     FILE *INFILE;
 9     FILE *OUTPUTFILE;
10     char VERBOSE;
11 };
12 extern struct mainarg_s MAINARG;
13
14 #endif
```

1. A quoi sert chaque include ?
2. Pourquoi a-t-on un fichier main.h ?
3. 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.
4. A quoi sert l'appel de return a la fin de la fonction main() ?
5. Pourquoi y-a-t-il exit() a la fin de la fonction usage() ?
6. Qu'est ce qu'un appel systeme, en voyez-vous dans ce fichier, si oui lesquels, citez en d'autres,
7. Quelle precaution doit on prendre lors de leur utilisation ?
8. Ou sont definiies les fonctions standards ?

9. Qu'est-ce qu'un filtre unix ?
10. Que faut-il faire pour transformer ce programme en filtre ?

Fichier main.c

```
 1 #include <stdlib.h>
 2 #include <stdio.h>
 3 #include <getopt.h>
 4 #include "main.h"
 5 #include "count.h"
 6 #include "hte.h"
 7
 8 struct mainarg_s MAINARG;
 9
10 void usage (char *command)
11 {
12     printf ("\nStatistique Diverses\n");
13     printf ("Usage  : %s
14             [-v] [-o OutFile] InFile\n", command);
15     printf ("-v          verbose mode\n");
16     printf ("-o OutFile  output file (stdout by default)\n\n");
17     exit (EXIT_FAILURE);
18 }
19
20 void getarg (int argc, char **argv)
21 {
22     extern char *optarg;
23     extern int optind;
24     char option;
25
26     MAINARG.OUTPUTFILE = NULL;
27     while ((option = getopt (argc, argv, "vo:")) != EOF)
28         switch (option)
29         {
30         case 'v':
31             MAINARG.VERBOSE = 1;
32             fprintf (stderr, "Verbose mode\n");
33             break;
34
35         case 'o':
36             if (MAINARG.VERBOSE)
37                 fprintf (stderr, "Fichier de sortie : %s\n", optarg);
38
39             MAINARG.OUTPUTFILE = fopen (optarg, "w");
40             if (MAINARG.OUTPUTFILE == NULL)
41             {
42                 fprintf (stderr, "%s: %s: \n", argv[0], optarg);
43                 perror ("fopen");
44                 exit (EXIT_FAILURE);
45             }
46             break;
47
48         case '?':
49         default:
50             usage (argv[0]);
51         }
52     if ((optind + 1) != argc)
53     {
54         usage (argv[0]);
55     }
56     else
57     {
58         if (MAINARG.VERBOSE)
59             fprintf (stderr, "Fichier d'entrée : %s\n", argv[optind]);
60
```

```
61          MAINARG.INFILE = fopen (argv[optind], "r");
62          if (MAINARG.INFILE == NULL)
63          {
64              fprintf (stderr, "%s: %s ", argv[0], argv[optind]);
65              perror ("fopen");
66              exit (EXIT_FAILURE);
67          }
68          if (MAINARG.OUTPUTFILE == NULL)
69              MAINARG.OUTPUTFILE = stdout;
70      }
71  }
72
73  int main (int argc, char **argv)
74  {
75      getarg (argc, argv);
76      count (MAINARG.INFILE);
77      fclose (MAINARG.OUTPUTFILE);
78      return EXIT_SUCCESS;
79  }
```

Les fonctions de bases de la tables de hachage

Fichier hte.h

```
 1  #ifndef _HTE_H_
 2  #define _HTE_H_
 3
 4  #include <stdlib.h>
 5
 6
 7  /*
 8   * Les deux structures ci-dessous ne sont pas définies ici
 9   * elles seront redéfinies dans chaque fichier
10   */
11  typedef struct hte_item_s hte_item_t;
12  typedef struct hte_data_s hte_data_t;
13
14
15  /*
16   * structure définissant la table de hachage
17   * On y trouve le nombre de liste de liste et
18   * un pointeur vers le tableau de liste
19   */
20  typedef struct hte_root_s
21  {
22      size_t NB_LIST;             /* nombre de liste d'ITEM du dictionnaire */
23      hte_item_t **LIST;          /* pointeur sur un tableau de liste d'ITEM */
24  } hte_root_t;
25
26
27  /*
28   * Calcul de l'index de hachage de Donald Knuth
29   * Elle produit un nombre entier à partir d'une chaine de caractères.
30   */
31  extern unsigned hte_hash (char *key);
32
33
34  /*
35   * hte_create fabrique une table de hachage vide
36   * parametres:
37   *      nb_item = nombre d'item maximum prévu dans la table de hachage
38   */
39  extern hte_root_t *hte_create (size_t nb_item);
40
```

```
41
42 /* ------------------------------------------------------------------------ */
43
44
45 /*
46  * hte_add ajoute l'item (key,data)
47  * parametres:
48  *      root = pointeur sur la table de hachage
49  *      key  = pointeur sur une chaine de caractères (la clé)
50  *      data = donnée associée à key
51  * comportement:
52  *      La clé key passée en parametre est comparée à celles présentes dans
53  *      la table. Si la clé n'est pas trouvée alors un nouvel item est créé
54  *      avec le couple (key,data). La clé key est en fait duppliquée.
55  *      Si la clé est trouvée, le champ DATA de l'item présent dans la
56  *      table est remplacé par le paramètre data de la fonction.
57  * retour:
58  *      Si l'item n'a pu etre créé faute d'espace, on sort du programme.
59  */
60 extern void hte_add (hte_root_t * root, char *key, hte_data_t * data);
61
62
63 /*
64  * hte_get recherche item, et en cas d'absence l'ajoute
65  * parametres:
66  *      root = pointeur pour la table de hachage
67  *      key  = pointeur sur une chaine de caractères (la clé)
68  * comportement:
69  *      La clé passée en parametre est comparé a celles présentes dans la
70  *      table de hachage.
71  * retour:
72  *      Si la clé est trouvée alors la fonction retourne le champ data
73  *      sinon la fonction rend NULL.
74  */
75 extern hte_data_t *hte_get (hte_root_t * root, char *key);
76
77
78 /* ------------------------------------------------------------------------ */
79
80
81 /*
82  * namealloc permet de garantir l'unicité des chaines de caractères
83  * parametres:
84  *      name = chaine de caractères
85  * comportement:
86  *      name est recherché dans la table,
87  *      s'il est absent on ajoute une copie de name dans la table
88  * retour:
89  *      un pointeur sur la copie de la table
90  */
91 char *namealloc (char *name);
92
93 /* nombre maximum de noms dans la table */
94 #define MAX_NAMEALLOC 1000
95
96
97 /* ------------------------------------------------------------------------ */
98
99 /*
100  * dejavu permet de savoir si un nom a déjà été vu
101  * parametres:
102  *      name = chaine de caractères
103  * comportement:
104  *      name est recherché dans la table,
105  *      s'il est absent on ajoute une copie de name dans la table
106  * retour:
```

```
107  *      rend 1 s'il était déjà présent sinon 0
108  */
109 unsigned dejavu (char *name);
110
111 /* nombre maximum de noms dans la table */
112 #define MAX_DEJAVU 1000
113
114 #endif
```

Fichier hte.c

```
  1 #include <stdio.h>
  2 #include <string.h>
  3 #include <stdlib.h>
  4 #include "hte.h"
  5
  6 /*
  7  * Calcul de l'index de hachage de Donald Knuth
  8  * Elle produit un nombre entier à partir d'une chaine de caractères.
  9  */
 10 unsigned hte_hash (char *key)
 11 {
 12     char *c = key;
 13     unsigned h;
 14     for (h = 0; *c; c++)
 15     {
 16         h += (h ^ (h >> 1)) + 314159 * (unsigned char) *c;
 17         while (h >= 516595003)
 18             h -= 516595003;
 19     }
 20     return h;
 21 }
 22
 23
 24 /*
 25  * Création d'un dictionnaire vide
 26  */
 27 hte_root_t *hte_create (size_t nb_item)
 28 {
 29     hte_root_t *root;
 30     unsigned i, nb_list;
 31     static unsigned primes[] = { /*suite croissante de nombres premiers */
 32         101, 149, 223, 347, 499, 727, 1163, 1697, 2503, 3989, 5839, 8543,
 33         12503, 20143, 29483, 43151, 63179, 92503, 101747, 148961, 218107,
 34         319313, 514243, 752891, 1212551, 1613873, 2599153, 3805463,
 35         5571523, 8157241, 11943011, -1
 36     };
 37
 38     /* détermination de la taille réelle du dictionnaire */
 39     for (i = 0; primes[i] < nb_item; i++);
 40     nb_list = primes[i];
 41     if (nb_item != -1)
 42     {
 43         /* allocation et initialisation du dictionnaire */
 44         if ((root = malloc (sizeof (hte_root_t))))
 45         {
 46             root->NB_LIST = nb_list;
 47             if ((root->LIST = calloc (nb_list, sizeof (hte_item_t *))))
 48                 return root;
 49         }
 50     }
 51     fprintf (stderr, "hte_create: not enough memory\n");
 52     exit (1);
 53 }
```

Le service *dejavu*

Fichier dejavu.c

```
 1 #include "hte.h"
 2 #include <stdio.h>
 3 #include <string.h>
 4 #include <stdlib.h>
 5
 6 /*
 7  * structure définissant un item du dictionnaire
 8  * Un item est formé d'un couple (clé,valeur) plus un pointeur
 9  * permettant de chainer les items ayant le meme index de hachage
10  */
11 struct hte_item_s
12 {
13     struct hte_item_s *NEXT;    /* pointeur sur l'item suivant */
14     char KEY[];                 /* tableau flexible contenant la clé */
15 };
16
17
18 unsigned dejavu (char *key)
19 {
20     unsigned index;
21     hte_item_t *curr;
22     static hte_root_t *root_namealloc;
23
24     if (key)
25     {
26         /* création du dictionnaire la première fois */
27         if (root_namealloc == NULL)
28             root_namealloc = hte_create (MAX_NAMEALLOC);
29
30         /* calcul de l'index pour trouver la liste ou devrait être l'item */
31         index = hte_hash (key) % root_namealloc->NB_LIST;
32
33         /* recherche et l'item dans sa liste et sortie si trouvé */
34         for (curr = root_namealloc->LIST[index]; curr; curr = curr->NEXT)
35         {
36             if ((int) curr == 0x4d52)
37                 fprintf (stderr,
38                          "------------------------------------------------------------ ARG1
39             if (strcmp (curr->KEY, key) == 0)
40                 return 1;
41         }
42
43         /* création d'une nouvelle entrée */
44         curr = malloc (sizeof (struct hte_item_t *) + strlen (key) + 1);
45         if (curr)
46         {
47             if ((int) curr == 0x4d52)
48                 fprintf (stderr,
49                          "------------------------------------------------------------ ARG2
50             strcpy (curr->KEY, key);
51             curr->NEXT = root_namealloc->LIST[index];
52             if ((int) curr == 0x4d52)
53                 fprintf (stderr,
54                          "------------------------------------------------------------ ARG3
55             root_namealloc->LIST[index] = curr;
56             return 0;
57         }
58     }
59     perror ("dejavu");
60     exit (1);
```

```
61 }
```

Le service *dictionnaire*

Fichier dico.c

```
 1 #include <stdio.h>
 2 #include <string.h>
 3 #include <stdlib.h>
 4 #include "hte.h"
 5
 6
 7 /*
 8  * structure définissant un item du dictionnaire
 9  * Un item est formé d'un couple (clé,valeur) plus un pointeur
10  * permettant de chainer les items ayant le meme index de hachage
11  */
12 struct hte_item_s
13 {
14     struct hte_item_s *NEXT;    /* pointeur sur l'item suivant */
15     hte_data_t *DATA;           /* valeur associée à l'item */
16     char KEY[];                 /* tableau flexible contenant la clé */
17 };
18
19
20 /*
21  * Recherche d'un item de clé key dans le dictionaire root
22  */
23 hte_data_t *hte_get (hte_root_t * root, char *key)
24 {
25     int index;
26     hte_item_t *curr;
27
28     if (key)
29     {
30         /* calcul de l'index pour trouver la liste ou devrait être l'item */
31         index = hte_hash (key) % root->NB_LIST;
32
33         /* recherche et l'item dans sa liste et sortie si trouvé */
34         for (curr = root->LIST[index]; curr; curr = curr->NEXT)
35             if (strcmp (key, curr->KEY) == 0)
36                 return curr->DATA;
37
38         return NULL;
39     }
40     perror ("hte_get");
41     exit (1);
42 }
43
44 /*
45  * Recherche d'un item de clé key dans le dictionnaire root
46  * avec création en cas d'absence
47  * et affectation d'une nouvelle donnée data
48  */
49 void hte_add (hte_root_t * root, char *key, hte_data_t * data)
50 {
51     int index;
52     hte_item_t *curr;
53
54     if (key)
55     {
56         /* calcul de l'index pour trouver la liste ou devrait être l'item */
57         index = hte_hash (key) % root->NB_LIST;
58
```

```
59          /* recherche et l'item dans sa liste et sortie si trouvé */
60          for (curr = root->LIST[index]; curr; curr = curr->NEXT)
61          {
62              if (strcmp (curr->KEY, key) == 0)
63              {
64                  curr->DATA = data;
65                  return;
66              }
67          }
68
69          /* création d'une nouvelle entrée */
70          curr = malloc (sizeof (struct hte_item_t *) + sizeof (void *) + strlen (key) + 1);
71          if (curr)
72          {
73              strcpy (curr->KEY, key);
74              curr->NEXT = root->LIST[index];
75              root->LIST[index] = curr;
76              curr->DATA = data;
77              return;
78          }
79      }
80      perror ("hte_add");
81      exit (1);
82  }
```

Le service allocateur de nom

Fichier namealloc.c

```
 1 #include "hte.h"
 2 #include <stdio.h>
 3 #include <string.h>
 4 #include <stdlib.h>
 5
 6 /*
 7  * structure définissant un item du dictionnaire
 8  * Un item est formé d'un couple (clé,valeur) plus un pointeur
 9  * permettant de chainer les items ayant le meme index de hachage
10  */
11 struct hte_item_s
12 {
13     struct hte_item_s *NEXT;    /* pointeur sur l'item suivant */
14     char KEY[];                 /* tableau flexible contenant la clé */
15 };
16
17
18 char *namealloc (char *key)
19 {
20     int index;
21     hte_item_t *curr;
22     static hte_root_t *root_namealloc;
23
24     if (key)
25     {
26         /* création du dictionnaire la première fois */
27         if (root_namealloc == NULL)
28             root_namealloc = hte_create (MAX_NAMEALLOC);
29
30         /* calcul de l'index pour trouver la liste ou devrait être l'item */
31         index = hte_hash (key) % root_namealloc->NB_LIST;
32
33         /* recherche et l'item dans sa liste et sortie si trouvé */
34         for (curr = root_namealloc->LIST[index]; curr; curr = curr->NEXT)
35         {
```

```
36              if (strcmp (curr->KEY, key) == 0)
37                  return curr->KEY;
38          }
39
40          /* création d'une nouvelle entrée */
41          curr = malloc (sizeof (struct hte_item_t *) + strlen (key) + 1);
42          if (curr)
43          {
44              strcpy (curr->KEY, key);
45              curr->NEXT = root_namealloc->LIST[index];
46              root_namealloc->LIST[index] = curr;
47              return curr->KEY;
48          }
49      }
50      perror ("namealloc");
51      exit (1);
52  }
```