Structures de données courantes
réalisation en langage C

M Billaud

16 juillet 2021

Table des matières

I Rappels sur le langage C **7**

1 Les données en mémoire **9**
1.1 Taille des données, opérateur `sizeof()` . . . 9
1.2 Structures en mémoire . . . 9
1.3 Adresse des données, opérateur "`&`" . . . 11
1.4 Tableaux et adresses . . . 12

2 Pointeurs **15**
2.1 Déclaration des pointeurs . . . 15
2.2 Pointeurs non-typé, conversions . . . 15
2.3 Indirection, opérateur "`*`" . . . 16
2.4 Le pointeur `NULL` . . . 16
2.5 Pointeurs de structures, notation "`->`" . . . 17
2.6 Passage de paramètres . . . 17
2.6.1 Pointeur pour le passage par référence . . . 17
2.6.2 Pointeur pour éviter de copier . . . 17
2.7 Parcours de tableau, arithmétique des pointeurs . . . 18

3 Allocation dynamique **19**
3.1 Fonctions `malloc()` et `free()` . . . 19
3.2 Réallocation . . . 19

II Quelques conteneurs **23**

4 Tableaux extensibles **25**
4.1 Choix d'une API, exemple d'utilisation . . . 25
4.2 L'implémentation . . . 27
4.2.1 Données . . . 27
4.2.2 Code . . . 27
4.2.3 Stratégie de doublement de la capacité . . . 28

5 Chainages **31**

6 Ensemble de chaînes, hachage **33**
6.1 Opérations de base . . . 33
6.2 Idée générale . . . 34
6.2.1 Répartition en alvéoles . . . 34

6.2.2 Agrandissement par doublement 35
6.2.3 Doublement et redistribution 35
6.3 Détails d'implémentation 35
6.3.1 Sémantique de valeur pour les chaines 35
6.3.2 Structure des alvéoles 36
6.4 Choix de la fonction de hachage 36
6.5 Code source 36
6.5.1 Entête 36
6.5.2 Code 37

Introduction

Objectif du document

Montrer les principes de fonctionnement de divers conteneurs (tableau extensible, listes chaînées, dictionnaire, ...) en allant jusqu'aux détails d'implémentation.

Pour aller au niveau le plus bas que permet la portabilité, l'implémentation est réalisée en C.

Inconvénient : le manque de généricité de C. Mais il ne s'agit pas de présenter une bibliothèque générique : on se limitera donc à présenter des conteneurs avec des éléments d'un type spécifique (exemple : liste d'entiers).

Choix du langage C

Bonnes et mauvaises raisons de programmer en C.

Ici : pour se contraindre à expliciter les détails de réalisation des structures de données.

Première partie

Rappels sur le langage C

Chapitre 1

Les données en mémoire

Un programme C contient généralement des variables. Pendant l'exécution, chaque variable est stockée en mémoire quand elle n'est pas sont pas éliminée par le compilateur, ce qui arrive quand il détecte que la variable est inutile, ou suffisamment temporaire pour être rangée dans une registre du processeur, dans un **emplacement en mémoire** formé d'octets consécutifs.

1.1 Taille des données, opérateur `sizeof()`

Le nombre d'octets (la **taille**) dépend du type de la variable. On la détermine en appliquant l'opérateur `sizeof()` à une variable ou à un type.

Le listing 1.1 montre un programme qui fait afficher les tailles de quelques types. Le résultat (listing 1.2) dépend des choix d'implémentation du compilateur que vous utilisez, sauf pour le type le type `char` qui correspond **toujours** à un octet exactement. [1]

Par exemple, un `int` occupe *en général* 4 octets sur une machine 32 bits, et 8 octets sur une machine 64 bits.

Remarques :

1. `sizeof()` retourne un `size_t`, type qui correspond à un entier non signé assez grand pour stocker une taille. L'implémentation de `size_t` (`unsigned int`, `unsigned long`, ...) est dépendante de l'architecture.
2. Portabilité : utilisez la spécification de format `%zu` pour le type `size_t`.

1.2 Structures en mémoire

Rappel : une structure contient un ou plusieurs membres (champs) qui peuvent être de types differents. Exemple de définition d'un type et d'une variable :

```
struct Employe {
  char nom[40];
  int  age;
};
```

1. Il a été nommé `char` à une époque où le codage d'un caractère tenait toujours sur un octet (codages ANSI, EBCDIC, ...). Si c'était à refaire, ce type s'appellerait certainement `byte`.

Listing 1.1 – Affichage de tailles de quelques types

```c
/**
 * Affichage des tailles de divers types de base.
 *
 * Les tailles dépendent du compilateur,
 * sauf pour pour char (toujours 1).
 */
#include <stdio.h>

int main()
{
    printf("type\ttaille\n----\t------\n");
    printf("char\t%zu\n",   sizeof(char));
    printf("int\t%zu\n",    sizeof(int));
    printf("long\t%zu\n",   sizeof(long));
    printf("float\t%zu\n",  sizeof(float));
    printf("double\t%zu\n", sizeof(double));
    return 0;
}
```

Listing 1.2 – Exécution du programme du listing 1.1

```
type    taille
----    ------
char    1
int     4
long    8
float   4
double  8
```

Listing 1.3 – Affichage des adresses de quelques variables

```c
/**
 * Affichage d'adresses de variables.
 *
 */
#include <stdio.h>

int glob1 = 12;
float glob2 = 34;

int main()
{
    int loc1 = 33;
    float loc2 = 3.14;
    printf("var.\tadresse\n----\t------\n");
    printf("glob1\t%p\n",  (void *)  & glob1);
    printf("glob2\t%p\n",  (void *)  & glob2);
    printf("loc1\t%p\n",   (void *)  & loc1);
    printf("loc2\t%p\n",   (void *)  & loc2);
    return 0;
}
```

Listing 1.4 – Exécution du programme

```
var.    adresse
----    ------
glob1   0x5594b04a9038
glob2   0x5594b04a903c
loc1    0x7fffa1c515fc
loc2    0x7fffa1c515f8
```

```c
struct Employe cuistot  = { "Maurice", 63 };
```

Exercice : . Écrivez un programme montrant un exemple de structure dont la taille n'est pas *égale* à la somme des tailles des champs.

1.3 Adresse des données, opérateur "&"

Le programme du Listing 1.3 fait afficher les adresses de quelques variables pendant l'exécution, obtenues en leur appliquant l'opérateur "`&`" (`address-of`)[2] du langage C. Résultat sur Listing 1.4

Les adresses sont des données typées : par exemple l'adresse d'une variable de type `int` est de type `int*`. L'affichage se fait avec la spécification "`%p`" (pour *pointer*) en les convertissant en "adresses non-typées" (`void *`).

2. Il s'agit ici des *adresses virtuelles*, dans l'espace mémoire où le système a chargé le processus.

Listing 1.5 – Tableaux et adresses

```c
/**
 * Affichage d'adresses tableaux / éléments
 */
#include <stdio.h>

int main()
{
  int tab[4] = { 11, 22, 33, 44 };

  printf("var.\tadresse\n----\t------\n");
  printf("tab\t%p\n", (void *) tab);
  for (int i = 0; i < 4; i++) {
    printf("tab[%d]\t%p\n", i, (void *) &tab[i]);
  }
  return 0;
}
```

Remarques :

- Les variables globales du programme et les variables locales de la fonction `main()` sont dans des "segments" dont les adresses diffèrent considérablement : le segment de données pour les variables globales, et le segment de pile pour les autres [3].
- Sur les systèmes d'exploitation modernes, les adresses virtuelles qui s'affichent changent à chaque exécution [4].

Exercice : reprenez l'exemple de structure (1.2) dont la taille est supérieure à la somme des tailles des champs, et définissez une variable de ce type. Faites afficher l'adresse et la taille de la structure et de chacun de ses champs. Conclusion ?

1.4 Tableaux et adresses

Le programme du listing 1.5 affiche l'adresse d'un tableau et de ses éléments. Résultat sur Listing 1.6.

Remarque : en C, une variable de type tableau désigne en fait l'*adresse* de l'emplacement qui a été réservé en mémoire pour placer les éléments. Il n'y a dons pas besoin de mettre un "`&`" devant `tab` dans le `printf`.

On constate qu'à l'exécution, les éléments se suivent en mémoire, et l'adresse du tableau correspond à celle du premier élément.

3. Sur une machine qui supporte la notion de segmentation, évidemment. Ce n'est pas le cas des petits microcontrôleurs dans le domaine de l'informatique embarquée

4. C'est une mesure de sécurité pour éviter l'exploitation de "débordements de tampon" et autres erreurs de programmation. Lors du chargement d'un programme, le système d'exploitation choisit des adresses aléatoires pour placer les segments dans l'espace mémoire virtuel du processus.

Listing 1.6 – Exécution du programme

```
var.      adresse
____      _______
tab       0x7ffd7b053230
tab[0]    0x7ffd7b053230
tab[1]    0x7ffd7b053234
tab[2]    0x7ffd7b053238
tab[3]    0x7ffd7b05323c
```

Chapitre 2

Pointeurs

On appelle **pointeur** une donnée qui contient l'adresse d'une donnée en mémoire. [1].
On emploie les pointeurs pour diverses raisons, en particulier :
— le passage de paramètres,
— le parcours de tableaux,
— la manipulation des données allouées dynamiquement.

2.1 Déclaration des pointeurs

Pour déclarer un pointeur destiné à contenir des adresses d'objets de type T, on précède son nom par une étoile. Exemples :

```
int *pi;                // pointeur sur un int
struct Personne *pp;    // pointeur sur struct Personne
```

Pour déclarer un tableau de pointeurs, le nom du tableau est précédé par une étoile

```
char *noms[10];  // tableau de 10 pointeurs de caractères
```

La règle générale est qu'en C, la déclaration d'une variable ressemble à son usage (voir l'indirection ci-dessous).

2.2 Pointeurs non-typé, conversions

Les pointeurs non-typés sont déclarés avec `void *`.

```
int entier = 123;
void *adresse = &entier;          // pointeur non typé
printf("valeur=%d, adresse=%p\n",
       entier, adresse);
```

1. Ce terme est aussi (hélas) souvent employé par extension pour désigner les adresses elles-mêmes. Nous essaierons d'éviter ce regrettable manque de rigueur, source de confusions, qui permettrait d'écrire qu'un pointeur (au sens de variable) *contient* un pointeur (au sens d'adresse)...

Remarque : l'expression "`& entier`" de la seconde ligne est de type `int *`, mais il y a une **conversion implicite** entre les adresses typées et non-typées.

2.3 Indirection, opérateur "*"

L'opérateur "`*`" fournit un accès à la donnée dont l'adresse est contenue dans un pointeur **typé**. Exemple :

```
int nombre = 12;
int *p = & nombre;              // pointeur typé (entiers)

*p = 33;                        // modif. à travers p
printf("= %d\n", *p);           // accès indirect
```

Terminologie : on dit que
- la variable `nombre` **est pointée par** `p`.
- on fait une **indirection** pour, à partir d'un pointeur, accéder à la donnée qu'il pointe.
- on **déréference** le pointeur.

Remarque : dans l'exemple d'un tableau de pointeurs de caractères vu plus haut,
- `noms[2]` est le troisième [2] pointeur ;
- `*noms[2]` est le `char` désigné par ce pointeur.

Et puisque `*noms[i]` est un `char`, dans la logique de C, il n'est pas anormal que la déclaration d'un tableau de pointeurs

```
char    *noms[10];
```

ressemble fortement à l'usage qu'on a des éléments.

2.4 Le pointeur `NULL`

La constante `NULL` est une valeur conventionnelle (de type `void*` que l'on affecte à un pointeur pour indiquer qu'il **ne contient pas**, à un moment donné, l'adresse d'un objet en mémoire. Le pointeur ne pointe sur rien. [3]

Quand un pointeur contient `NULL`, tenter de le déréférencer est un **comportement indéfini**, qui provoque généralement un arrêt brutal de l'exécution :

```
int *p = NULL;
*p = 12;                // crash
```

2. le premier a l'indice 0...
3. Ne pas confondre avec un pointeur non-initialisé, qui contient une valeur aléatoire

2.5 Pointeurs de structures, notation "->"

Selon les règles de priorités d'opérateurs de C, "`*a.b`" se lit "`*(a.b)`".

La notation "pointeur->champ" facilite la désignation d'un champ d'une structure dont on a l'adresse dans un pointeur. Exemple :

```
struct Point {
  float x, y;
};
...
struct Point *p;  // p pointeur de point

p->x = 0.0;           // au lieu de  (*p).x = 0.0
p->y = 0.0;
```

2.6 Passage de paramètres

Le langage C ne connaissant que le passage de paramètres **par valeur**, on utilise des pointeurs pour simuler le "passage de référence" dans deux situations :

1. l'action que l'on veut coder modifie un objet qu'on lui indique,
2. les objets que l'on souhaite transmettre sont assez gros, et pour des raisons de performance, on veut éviter la copie inhérente à un passage par valeur.

2.6.1 Pointeur pour le passage par référence

Exemple du listing 2.1 : une action consistant à échanger les nombres contenus dans deux variables. On la traduit par une fonction à qui on passe les adresses des variables à modifier.

Listing 2.1 – Émulation d'un passage par référence

```
void echanger(int *pa, int *pb) {
  int tmp = *pa;
  *pa = *pb;
  *pb = tmp;
}

// usage
int a = 34, b = 23;
echanger( & a, & b);
```

2.6.2 Pointeur pour éviter de copier

Exemple (listing 2.2 : affichage d'une structure.

Listing 2.2 – Passage d'un structure volumineuse

```
  struct Personne {
    char nom[100];
```

```
    char prenom[100];
    ...
};

void afficher_personne(const struct Personne *p) {
    printf("nom = %s\n", p->nom);
    ...
}
```

Le mot-clé `const` annonce nos intentions. La déclaration de paramètre se lit de droite à gauche : `p` est un pointeur vers une structure `Personne` qu'on ne modifie pas.

2.7 Parcours de tableau, arithmétique des pointeurs

Une chaine de caractères est un tableau d'octets terminé par un caractère nul.

```
char test[] = "abc";  // tableau de 4 octets
```

Pour parcourir une chaine, on peut[4] utiliser un pointeur qui va désigner tour à tour chaque octet :

```
void affiche_codes(const char chaine[]) {
    char *p = chaine;
    while (*p != '\0') {
        printf("-> %d\n", *p);
        p++;
    }
}
```

Remarques

1. Un tableau déclaré en paramètre est en réalité un pointeur.
2. l'incrémentation d'un pointeur (`p++`) modifie ce pointeur pour qu'il désigne l'élément suivant[5]

4. à la place d'un indice
5. la valeur numérique du pointeur - celle qu'on voit avec `printf` - est augmentée de la taille du type pointé (ici 1, parce que c'est un `char`.

Chapitre 3

Allocation dynamique

L'**allocation dynamique de mémoire** est un ensemble de fonctionnalités mises à la disposition du programmeur d'application par la bibliothèque standard C.

Elle lui permet de gérer de l'espace mémoire supplémentaire (en plus de la pile d'exécution et du segment de données) pour y placer des données, en spécifiant le nombre d'octets voulu. Elle permet aussi de libérer un espace alloué dont on n'a plus besoin.

Attention : l'usage de l'allocation dynamique impose un soin très attentif au programmeur qui est guetté par deux dangers :

— **la fuite mémoire** si un programme alloue en boucle des zones mémoires, sans les libèrer quand il n'en n'a plus besoin. L'espace mémoire du programme s'agrandit indéfiniment, ce qui finit mal.

— la **corruption des données** si un programme utilise par erreur une zone qui a été libérée.

C'est une difficulté typique de la programmation en C.

3.1 Fonctions `malloc()` et `free()`

Nous utilisons essentiellement deux fonctions, définies dans `stdlib.h` :

— `malloc()` pour obtenir de l'espace mémoire supplémentaire,

— `free()` pour restituer (libérer) de l'espace obtenu par `malloc()`,

et occasionnellement `realloc()` qui agrandit ou rétrécit un espace qu'on a obtenu, quitte à le déménager ailleurs.

Le listing 3.1 montre l'utilisation d'un tableau de structures alloué dynamiquement.

Pour l'allocation par `malloc()`, on indique en paramètre la taille (nombre d'octets) souhaitée. La fonction retourne l'adresse (non typée) de la zone allouée, ou `NULL` en cas d'échec..

Pour libérer une zone, on fournit son adresse à la fonction `free`.

Exercice : écrire les fonctions manquantes.

3.2 Réallocation

Si on veut ajouter un employé supplémentaire, il faut agrandir le tableau. pour cela on fait un appel à `realloc()` en indiquant

— l'adresse de la zone que l'on veut redimensionner (ici `tableau`),

Listing 3.1 – Exemple : tableau de structures alloué dynamiquement

```
/**
 * Allocations et libérations. A compléter
 * Un tableau de personnes
 */
#include <stdio.h>
#include <stdlib.h>

struct Employe {
  char prenom[20];
  int  bureau;
};

struct Employe *nouveauTableau(int nb)
{
  struct Employe *t
    = malloc(nb * sizeof(struct Employe));
  if (t == NULL) {
    fprintf(stderr, "échec d'allocation");
    exit (EXIT_FAILURE);
  }
  return t;
}

int main()
{
  int nbEmployes;
  printf("Combien d'employés ? ");
  scanf("%d", & nbEmployes);

  struct Employe *tableau = nouveauTableau(nbEmployes);

  for (int i = 0; i < nb; i++) {
    lire_employe( & tableau[i]);
  }
  for (int i = 0; i < nb; i++) {
    afficher_employe( & tableau[i]);
  }

  free(tableau);                    // libération
  exit (EXIT_SUCCESS);
}
```

— la nouvelle taille

et `realloc` retournera l'adresse de la nouvelle zone :

```
nbElements += 1;
tableau = realloc(tableau,
                  nbElements * sizeof(struct Employe));
```

À savoir :

— si le premier paramètre de `realloc` est `NULL`, la fonction se comporte comme `malloc()`,

— vous l'aviez deviné : en cas d'échec, `realloc()` retourne `NULL`.

Exercice : écrivez un programme qui

— part d'un tableau vide

— fait une boucle, en demandant si on veut en ajouter d'autres

— les affiche tous à la fin.

Deuxième partie

Quelques conteneurs

Chapitre 4

Tableaux extensibles

Nous appellons *tableau extensible* une structure de données qui sert à stocker des élements, et

— comme un tableau ordinaire, permet de désigner un élément par sa position (première = 0, seconde = 1, etc.), pour le modifier ou le consulter.

— à la différence des tableaux, permet d'ajouter des éléments à la fin sans limite de taille [1].

Ici nous allons prendre l'exemple des tableaux extensibles d'entiers.

4.1 Choix d'une API, exemple d'utilisation

— Le type "tableau extensible d'entiers" se matérialise par une structure appelée `tab_int`.

— Une famille de fonctions, dont le nom est préfixé par "`ti`" représentera les actions qui agissent dessus.

— Le premier paramètre de ces fonctions sera toujours l'adresse du tableau concerné. [2]

Le programme du listing 4.1 montre l'emploi d'un tel tableau, et on voit fig. 4.2 le résultat de l'exécution.

Listing 4.1 – Utilisation d'un tableau extensible d'entiers

```
#include <stdio.h>
#include <stdlib.h>

#include "tab_int.h"

void afficher(const char *m, const struct tab_int *a);

int main()
{
  struct tab_int tableau;
  ti_init(& tableau);

   // 10, 20 ... 100
```

1. autre que les limitations de l'allocation dynamique
2. pour les deux raisons évoquées plus haut :

— c'est obligatoire pour les fonctions qui modifient le tableau

— c'est souhaitable pour les autres, pour éviter de faire des copies. Dans ce cas on mettra un `const`.

```
  for (int v = 10; v <= 100; v += 10) {
    ti_ajouter(& tableau, v);
  }

  afficher("avant", & tableau);
  ti_changer(& tableau, 3, 421);
  afficher("après", & tableau);

  ti_detruire(& tableau);

  exit (0);
}

/**
 * affiche un message et le contenu d'un tableau
 * @param m : chaine
 * @param a : adresse du tableau
 */
void afficher(const char *m,
              const struct tab_int *a)
{
  printf("%s : ", m);
  int taille = ti_taille(a);
  for (int i = 0;  i < taille; i++) {
    printf("%d ", ti_valeur(a, i));
  }
  printf("\n");
}
```

Listing 4.2 – Exécution

```
avant : 10 20 30 40 50 60 70 80 90 100
après : 10 20 30 421 50 60 70 80 90 100
```

4.2 L'implémentation

4.2.1 Données

Un tableau extensible est représenté par
— un **tableau** alloué dynamiquement, pouvant accueillir un certain nombre d'éléments (sa **capacité**),
— un entier indiquant le nombre d'éléments utilisés, au début du tableau (sa **taille**)

Listing 4.3 – Fichier `tab_int.h`

```
#ifndef TAB_INT_H
#define TAB_INT_H

struct tab_int {
  int  taille;
  int  capacite;
  int *elements;
};

void ti_init     (       struct tab_int *a);
void ti_ajouter  (       struct tab_int *a, int valeur);
void ti_detruire(        struct tab_int *a);
int  ti_taille   (const struct tab_int *a);
int  ti_valeur   (const struct tab_int *a, int indice);
void ti_changer  (       struct tab_int *a, int indice,
                                            int valeur);
#endif
```

4.2.2 Code

Le code comporte quelques choix d'implémentation
— la capacité initiale, lorsqu'on initialise un tableau extensible (ici, 4 éléments)
— la stratégie d'agrandissement en cas de débordement. Ici on double : l'ajout du 5ieme élément réalloue le tableau avec une capacité de 8, et l'ajout du 8ieme passe la capacité à 16. Cette stratégie est justifiée section 4.2.3.

Listing 4.4 – Code du module `tab_int.h`

```
#include <stdlib.h>
#include "tab_int.h"

#define CAPACITE_MINIMALE 4

void ti_init (struct tab_int *a)
{
  a->taille = 0;
  a->capacite = CAPACITE_MINIMALE;
  // NOTE : on devrait vérifier le résultat de malloc
  a->elements = malloc(a->capacite * sizeof(int));
}

void ti_ajouter(struct tab_int *a, int valeur)
{
```

```
  // si plein, agrandir
  if (a->taille == a->capacite) {
    a->capacite *= 2;
    // NOTE : on devrait vérifier le résultat de realloc
    a->elements = realloc(a->elements,
                          a->capacite * sizeof(int));
  }
  // ajout à la fin
  a->elements[a->taille] = valeur;
  a-> taille += 1;
}

void ti_detruire(struct tab_int *a)
{
  a->taille = 0;
  a->capacite = 0;
  free(a->elements);
  a -> elements = NULL;
}

int ti_taille(const struct tab_int *a)
{
  return a->taille;
}

// les indices doivent être entre 0  et a->taille - 1

int ti_valeur(const struct tab_int *a, int indice)
{
  return a->elements[indice];
}

void ti_changer(struct tab_int *a, int indice,
                int valeur)
{
  a->elements[indice] = valeur;
}
```

4.2.3 Stratégie de doublement de la capacité

Lorsque le tableau est plein, on le réalloue avec une capacité supérieure.

La stratégie de doublement de cette capacité est, contrairement à ce que suggère l'intuition, très efficace en terme de nombre de copies : au cours du remplissage, chaque élément a été copié **au plus une fois** en moyenne.

Imaginons qu'à un moment le vector ait grandi jusqu'à 500 éléments. Comme le tableau grandit en doublant de taille, sa capacité est la première puissance de 2 supérieure à 500, soit 512.

Le tableau sera agrandi (et réalloué) en ajoutant le 513ieme, sa capacité passera à 1024 élements, et pour cela il faudra réallouer ce qui provoquera la copie des 512 éléments existants. Cout : 512, si on prend comme unité la copie d'un élément.

Mais pour arriver à 513, il avait fallu copier 256 éléments. Et pour arriver à 257, en copier 128.

Si on fait le total, si on en est au 513-ième élément ajouté (et jusqu'au 1024-ième) on a fait en tout $256 + 128 + 64 + \ldots$ copies d'éléments, ce qui est plus petit que 512.

Dans le pire des cas (ajout du 513 ième), le coût moyen d'ajout d'un élément est inférieur à 512/513 : il y a donc eu **moins d'une copie par élément**.

Exercice : évaluez la stratégie consistant à augmenter d'un la capacité à chaque ajout.

Chapitre 5

Chainages

Chapitre 6

Ensemble de chaînes, hachage

Dans cette partie, nous montrons comment représenter **efficacement** un ensemble[1] de chaînes de caractères en utilisant une **fonction de hachage**.

6.1 Opérations de base

Les opérations de base sur cet ensemble :
— l'initialiser,
— y ajouter un élément (si il n'y est pas déjà),
— savoir combien il y a d'éléments dans l'ensemble,
— tester si un élément est présent,
— enlever un élément,
— libérer les ressources utilisées.

Sur le listing 6.1 figure un exemple d'utilisation où l'on ajoute une suite de mots (éventuellement en plusieurs exemplaires) et on fait afficher la taille (qui doit être 10).

Le programme affiche également (listing 6.2) le contenu interne de l'ensemble de chaînes, ce qui nous facilitera les explications.

Listing 6.1 – Utilisation d'un ensemble de chaînes

```
#include <stdio.h>
#include <stdlib.h>

#include "ens_chaines.h"

int main()
{
    struct ens_chaines ensemble;
    ec_init(& ensemble);

    char *mots[] = {
        "un", "deux", "trois", "un",
        "quatre", "deux", "cinq", "six",
        "sept", "trois", "huit", "neuf",
        "dix", "trois", "sept",
```

1. fini, donné par extension

```
        NULL
    };
    for (int i = 0; mots[i] != NULL; i++) {
        ec_ajouter(& ensemble, mots[i]);
    }

    printf("-> taille %d (attendu = 10)\n",
            ec_taille(& ensemble));

    ec_dump(& ensemble);
    ec_liberer(& ensemble);

    return EXIT_SUCCESS;
}
```

Listing 6.2 – Exécution

```
-> taille 10 (attendu = 10)
0 ->
1 ->    "trois" (10282497)
2 ->    "quatre" (170727922)
3 ->    "un" (2099)
4 ->    "six" (35140)
5 ->    "dix" (30805)
6 ->    "deux" (522598)
7 ->
8 ->
9 ->
10 ->   "huit" (546666)
11 ->   "cinq" (518715)
12 ->   "sept" (596204)
13 ->
14 ->   "neuf" (571710)
15 ->
```

Nous ne présenterons que quelques opérations, les autres sont laissées en exercice.

6.2 Idée générale

6.2.1 Répartition en alvéoles

- les chaînes de caractères qui font partie de l'ensemble sont réparties dans des "alvéoles".
- les alvéoles forment un tableau, ce qui permet un accès rapide par indice.
- le numéro de l'alvéole dans laquelle se trouve (ou devrait se trouver) une chaîne de caractères est calculé à partir du contenu de cette chaîne, par ce qu'on appelle une **fonction de hachage**.
- plus précisement, le numéro d'alvéole s'obtient comme reste (opération modulo) de la division de la valeur du hachage par le nombre d'alvéoles.

Intérêt : La répartition en alvéoles permet de diviser le nombre de comparaisons nécessaires pour tester la présence d'une chaîne : on ne regarde que celles présentes dans son alvéole.

Dans l'idéal, la fonction de hachage serait parfaite, et conduirait à une alvéole où ne se trouve qu'une chaîne.

En pratique, il va y avoir quand même plusieurs chaînes dans certaines alvéoles. On va donc
- prévoir qu'une alvéole contient une `liste` de chaînes,
- avoir un grand nombre d'alvéoles de façon à avoir statistiquement peu de chaînes par alvéole.

6.2.2 Agrandissement par doublement

La stratégie choisie est de doubler le nombre d'alvéoles quand le nombre de chaînes présentes dans l'ensemble atteint certain seuil (3/4 du nombre d'alvéoles). Les chaines sont alors redistribuées entre les alvéoles.

Le respect de ce seuil garantit qu'il a au maximum 0.75% chaines par alvéole. Il y aura donc peu d'alvéoles avec plus d'une chaine.

Comme pour les tableaux extensibles, la stratégie de doublement fait qu'en moyenne chaque alvéole est copiée au plus une fois.

6.2.3 Doublement et redistribution

Le doublement a une autre propriété intéressante. Quand on redistribue les chaines d'une alvéole,
- soit elles restent dans la même alvéole,
- soit elles vont dans une alvéole "jumelle" qui vient d'être ajoutée.

Exemple : pour la chaîne `"dix"`, la fonction de hachage vaut 30805.
- Si il y a 4 alvéoles, elle se trouve dans l'alvéole $30805 \% 4 = 1$.
- En passant à 8 alvéoles, elle va dans la nouvelle alvéole $30805 \% 8 = 5 = 4 + 1$.
- En passant à 16, elle reste en $30805 \% 16 = 5$.
- En passant à 32, elle va en $30805 \% 32 = 21 = 16 + 5$.
- etc.

Ceci nous autorise à redistribuer les chaines en traitant les anciennes alvéoles une par une : on est sûr de ne pas avoir à déplacer chaque chaine plus d'une fois.

6.3 Détails d'implémentation

6.3.1 Sémantique de valeur pour les chaines

Lorsqu'on appelle la fonction qui sert à ajouter une chaine, ce qu'on veut c'est ajouter le contenu de la chaine. Pour cela on ne peut pas se contenter de stocker l'adresse de la chaine reçue, il faut en faire une copie.

Ci-dessous une erreur classique de programmation en C : si on fait ensuite afficher le tableau, on s'aperçoit qu'il ne contient pas ce qu'on pense y avoir mis :

```
char *joueurs[10];
char nom[100];
for (int i=0; i < 10; i++) {
  printf("donnez un nom :");
  scanf("%s", nom);
```

```
    joueurs[i] = nom;          // une erreur de débutant
}
```

puisqu'on a stocké 10 fois l'adresse de la même variable locale `nom`...

Lors de l'ajout d'une chaine, on stocke donc en réalité *une copie* obtenue par `strdup()`. Cette copie est une ressource appartenant à l'ensemble, et sera libérée quand

— on retire une chaine de l'ensemble

— on libère l'ensemble

6.3.2 Structure des alvéoles

Les alvéoles, qui en principe ne contiendront que peu d'éléments (moins de 0.75 en moyenne), sont représentées ici par des listes chainées non ordonnées.

6.4 Choix de la fonction de hachage

Une fonction de hachage retourne un nombre non signé, parce qu'elle sert à calculer un indice (entier positif ou nul) comme reste d'une division (opérateur modulo).

Ce modulo (par une puissance de 2) fait qu'on utilise comme indice les bits de poids faible de la valeur retournée. Il est important que ces bits soient, autant que possible, dépendants de tous les caractères de la chaine.

Un contre-exemple pour illustrer cette notion. Si le hachage était calculé ainsi

```
unsigned int hash = 0;
    for (const char *c = chaine; *c != '\0'; c++) {
        hash = 16 * hash + *c;                  // mauvais
    }
```

à cause du décalage produit par la multiplication par 16, les 4 bits de droite ne dépendraient que du dernier caractère de la chaine ; les 8 bits de droite des deux derniers, etc. Les chaines `"sept"` et `"huit"` se retrouveraient toujours dans la même alvéole, pour les ensembles qui ont moins de 256 alvéoles. Idem pour `"six"` et `"dix"`. Et à partir d'une certaine taille, les premiers octets de la chaine seront sans influence sur le résultat (ils seront perdus dans le débordement).

La multiplication par 17 (16 + 1) garantit que chaque octet de la chaine a une influence sur les bits de poids faible du résultat de la fonction de hachage.

6.5 Code source

6.5.1 Entête

Listing 6.3 – Ensemble de chaines : entêtes

```
// ens_chaines.h

#ifndef ENS_CHAINE_H
#define  ENS_CHAINE_H

struct ens_alveole;             // prédéclaration

struct ens_chaines {
```

```
    int nb_alveoles;
    int nb_elements;
    struct ens_alveole *alveoles;
};


void ec_init   (struct ens_chaines *e);
void ec_ajouter(struct ens_chaines *e,
                const char *chaine);

int ec_taille  (const struct ens_chaines *e);
void ec_dump   (const struct ens_chaines *e);

void ec_liberer(struct ens_chaines *e);

#endif
```

6.5.2 Code

Listing 6.4 – Ensemble de chaines : implémentation

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

#include "ens_chaines.h"


struct ens_cellule {
    char * chaine;
    struct ens_cellule *suivant;
};

struct ens_alveole {
    struct ens_cellule *premier;
};


#define NOMBRE_MIN_ALVEOLES 4

void ec_init(struct ens_chaines *e)
{
    e->nb_elements = 0;
    e->nb_alveoles = NOMBRE_MIN_ALVEOLES;
    e->alveoles = malloc(e->nb_alveoles
                         * sizeof (struct ens_alveole));
}

static unsigned int ec_hash(const char * chaine)
{
    unsigned int hash = 0;
    for (const char *c = chaine; *c != '\0'; c++) {
        hash = 17 * hash + *c;
    }
    return hash;
}
```

```c
static void ec_doubler_nb_alveoles(struct ens_chaines *e)
{
    int na = e->nb_alveoles; // avant agrandissement
    e->nb_alveoles *= 2;

    int taille =
        e->nb_alveoles * sizeof (struct ens_alveole);
    e->alveoles = realloc(e->alveoles, taille);


    // initialisation de nouvelles alvéoles
    for (int i = na; i < e->nb_alveoles; i++) {
        e->alveoles[i].premier = NULL;
    }

    // reclassement des éléments des anciennes alvéoles
    for (int i = 0; i < na; i++) {
        struct ens_cellule *premier
            = e->alveoles[i].premier;
        e->alveoles[i].premier = NULL;

        while (premier != NULL) {
            struct ens_cellule *c = premier;
            premier = premier->suivant;
            int num_alveole
                = ec_hash(c->chaine) % (e->nb_alveoles);
            struct ens_alveole *a
                = &( e->alveoles[num_alveole] );
            c->suivant = a->premier;
            a->premier = c;
        }
    }
}

void ec_ajouter(struct ens_chaines *e, const char *chaine)
{
    int num_alveole = ec_hash(chaine) % (e->nb_alveoles);
    struct ens_alveole * a = &(e->alveoles[num_alveole]);

    // sortie si déjà present
    for (struct ens_cellule *c = a->premier;
         c != NULL;
         c = c->suivant) {
        if (strcmp(c->chaine, chaine) == 0) {
            return;
        }
    }

    // Ajout nouvelle cellule avec copie de chaine
    struct ens_cellule *nc
        = malloc(sizeof (struct ens_cellule));
    nc->chaine = strdup(chaine);
    nc->suivant = a->premier;
    a->premier = nc;
    e->nb_elements += 1;

    // besoin d'agrandir ?
    if (e->nb_elements >= (3 * e->nb_alveoles) / 4) {
        ec_doubler_nb_alveoles(e);
    }
}
```

```
void ec_liberer(struct ens_chaines *e)
{
    for (int i = 0; i < e->nb_alveoles; i++) {
        struct ens_cellule *premier
            = e->alveoles[i].premier;
        while (premier != NULL) {
            struct ens_cellule *c = premier;
            premier = premier->suivant;
            free(c->chaine);
            free(c);
        }
    }
    free(e->alveoles);
    // par précaution
    e->nb_alveoles = 0;
    e->nb_elements = 0;
    e->alveoles = NULL;
}

int ec_taille(const struct ens_chaines *e)
{
    return e->nb_elements;
}

void ec_dump(const struct ens_chaines *e)
{
    for (int i = 0; i < e->nb_alveoles; i++) {
        printf("%d -> ", i);
        for (struct ens_cellule *c = e->alveoles[i].premier;
                c != NULL; c = c->suivant) {
            printf("\t\"%s\" (%u)",
                    c->chaine,
                    ec_hash(c->chaine));
        }
        printf("\n");
    }
}
```