Lecture de fichiers en C

Florian Bourse

Le but de ce TP est d'implémenter des fonctions basiques sur les fichiers textes grâce à un code source en C séparé en plusieurs fichiers.

L'objectif est un programme qui lit un fichier qui lui est donné en argument et affiche son mirroir (tous les caractères sont affichés mais en commençant par le dernier et en terminant par le premier). Le programme doit lire l'entrée standard si aucun fichier n'est passé en paramètre.

Ouvrir et parcourir un fichier

Commençons avec un fichier vierge `parse.c` et ajoutons les bibliothèques nécessaires, ainsi que la fonction main et ses arguments dont nous aurons besoin.

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

int main(int argc, char *argv[]){

  return EXIT_SUCCESS;
}
```

Un fichier doit être ouvert avant de pouvoir être lu, et doit être fermé lorsque la lecture est terminée. Pour cela, on va utiliser les fonctions :

`fopen :` prend en argument deux chaînes de caractères. La première désigne le nom d'un fichier et la seconde désigne les permissions utilisés pour ouvrir ce fichier : `"r"` désigne une ouverture du fichier pour la lecture uniquement. Cette fonction renvoie un descripteur de fichier, de type `FILE *`.

`fclose :` prend en argument un descripteur de fichier et ferme le fichier.

Exemple d'utilisation :

```
FILE *f = fopen("80jours.txt","r");
/* Commentaire à remplacer par la lecture du fichier */
fclose(f);
```

On peut à présent lire les caractères du fichier un par un avec la fonction suivante :

`fscanf :` prend en argument un descripteur de fichier, une chaîne de formattage, et éventuellement des pointeurs, selon la chaîne de formattage. Lit dans un fichier une chaîne de caractères qui correspond au format passé en entrée. Par exemple `"%c"` est une chaîne de formattage qui désigne 1 caractère. Le caractère remplacé par `"%c"` est recopié dans l'emplacement mémoire désigné par le pointeur donné en argument. La valeur de retour de `fscanf` est `EOF` si la lecture à échouée car le fichier est arrivé à sa fin.

`scanf :` prend en argument uniquement la chaîne de formattage et les pointeurs et lit dans le flux `stdin` à la place.

Exemple d'utilisation :

```
char x;
fscanf(f, "%c", &x);
// La valeur de x est maintenant le premier caractère de f
fscanf(f, "%c", &x);
// La valeur de x a changé et est maintenant le second caractère de f
```

Stocker le fichier

Pour pouvoir afficher le fichier dans le sens inverse, il nous faut le stocker en mémoire. La taille n'étant pas connue à l'avance, on ne peut pas le stocker dans un tableau de taille statique.
La solution la plus simple à mettre en place, et qui utilise au mieux la mémoire, consiste à lire deux fois le fichier : une première fois pour calculer sa taille, puis on alloue la mémoire et on le relit pour remplir la chaîne de caractère. On peut pour cela utiliser les fonctions suivantes :

`sizeof :` si on lui donne en argument un type, renvoie le nombre d'octets occupés par une valeur de ce type.

`malloc :` prend en argument un entier n et renvoie un pointeur vers un emplacement mémoire de n octets contigus. Si l'allocation de la mémoire échoue, renvoie la valeur particulière `NULL`.

`free :` prend en argument un pointeur p et libère l'emplacement mémoire vers lequel pointe p. Il faut que cet emplacement ait été alloué par un appel à `malloc`. On ne peut pas appeler `free` plusieurs fois sur le même emplacement mémoire.

Exemple d'utilisation :

```
char *str = malloc(10 * sizeof(char));
str[4] = 'a';
free(str);
```

Traitement du fichier

Nous sommes à présent capables de lire le contenu d'un fichier et de le stocker dans une chaîne de caractère dont on connaît la tialle. On peut utiliser cet outil pour répondre à différent problèmes. Par exemple :

— Compter le nombre d'occurences d'un caractère.
— Déterminer si le parenthésage d'un fichier est cohérent.
— Vérifier qu'aucune ligne ne fasse plus de 80 caractères.
— Déterminer si un fichier est un palindrome.

Redimensionner le tableau

Parfois, la solution précédente ne s'applique pas (e.g., si on veut stocker le contenu d'un flux comme `stdin`). Une autre solution consiste à commencer avec un tableau d'une certaine taille que l'on retient, et lorsque le tableau devient trop petit (si on cherche à écrire dans une case qui dépasse la dernière case du tableau), on créer un tableau plus grand, et on recopie les éléments du premier tableau dans le second. On continue ensuite avec le second tableau, et on libère la mémoire occupée par le premier. On peut être amenés à redimensionner plusieurs fois le tableau.

Ranger le code : fichiers séparés

À présent, il serait souhaitable de bien ranger notre code pour pouvoir le réutiliser plus tard. Par exemple, les tableaux redimensionnables que l'on a implémenté pourront servir dans d'autres contextes. Rangeons les dans un autre fichier : `string.c`. On pourra inclure tout son contenu dans le fichier `parse.c` en ajoutant dans son entête

```
#include "parse.c"
```

Ranger le code : structures

Les tableaux redimensionnables vont toujours avec leur nombre d'élément et la capacité maximale du tableau. Pour éviter de toujours utiliser ces 3 paramètres et avoir des codes à rallonge pour les interfaces des fonctions, ou encore pour pouvoir renvoyer un triplet contenant ces éléments sans avoir à passer de pointeur en paramètre, on peut créer un type structuré. Le code suivant déclare un nouveau type `struct string_s`, puis lui donne un alias `string_t` pour y accéder de manière plus concise :

```
struct string_s {
  char *data;
  int len;
  int max;
};
typedef struct string_s string_t;
```

Un élément `id` de type `string_t` possède 3 champs : `id.data` qui est un pointeur vers un caractères, `id.len` qui est un entier, et `id.max` qui est un entier également. Ils peuvent être lus ou modifiés, et lors de la déclaration d'une variable d'un type structuré, on peut donner des valeurs aux champs, dans l'ordre. Par exemple :

```
string_t str = {NULL, 0, 0};
str.max = 10;
str.data = malloc(10 * sizeof(char));
str.data[1] = '\0';
```