Langage C
Fichiers binaires

1 Introduction

Un fichier binaire peut être vu comme une suite d'octets, mais contrairement à un fichier texte, ces octets ne sont plus des codes ascii de caractères : ils peuvent représenter des données quelconques.
Les fichiers binaires s'ouvrent et se ferment comme les fichiers textes, toujours à l'aide des fonctions `fopen` et `fclose` dont les paramètres ont alors le même sens. Les manipulations de la position courante s'effectuent toujours avec `fseek` et `ftell`, et de la même manière.

1.1 Fichiers binaires vs. fichiers textes

Considérons par exemple une variable de type `int` :

```
1 int x;
```

Supposons que dans la suite du programme, `x` prenne une certaine valeur que nous souhaitions sauvegarder dans un fichier.
Si nous décidons de sauvegarder la valeur dans un fichier texte, il faudra d'abord transformer cette valeur en une suite de caractères (par exemple ␣␣13␣␣), puis écrire ces caractères un à un dans le fichier. Inversement, la lecture d'une valeur pour `x` depuis un fichier texte nécessite la transformation d'une suite de caractères en un `int`. Il s'agit de ce qu'on appelle des lectures et écritures formatées.
Si par contre nous décidons de nous servir de fichiers binaires, l'ecriture de la valeur de `x` dans un fichier ne nécessite aucune transformation : il suffira de transférer `sizeof(int)` octets de la mémoire vive où réside `x` vers le fichier. Inversement, si l'on souhaite lire une valeur pour `x` depuis un fichier binaire, il suffira de transférer `sizeof(int)` octets depuis le fichier vers la mémoire vive où réside `x`.

1.2 Non-portabilité des fichiers binaires

Les fichiers binaires ne sont pas portables. Ceci est évident lorsque le `sizeof` d'une donnée diffère entre deux machines. Même en cas d'égalite de ce `sizeof`, rien ne garantit qu'un fichier binaire écrit par l'une des machines soit correctement lisible par l'autre. Par exemple, sur la plupart de machines contemporaines, `sizeof(int) == 4`. Mais l'*ordre* dans lequel les octets en mémoire représentent un `int` dépend du « boutisme » du processeur [1].
Ces problèmes ne sont pas les seuls rencontrés dans la portabilité des fichiers binaires. Une garantie plutôt probable est qu'un fichier binaire écrit sur une machine sera lisible sur la *même* machine... et encore : il vaut mieux que le programme ayant écrit le fichier et le programme relisant le fichier aient été compilés avec le même compilateur – ou idéalement, que ce soit le même programme qui effectue ces deux opérations.

1. *c.f.* l'article sur wikipedia `https://fr.wikipedia.org/wiki/Boutisme`.

Si vous vous demandez comment des données binaires, par exemple des `int`, sont transmises via internet entre deux machines distantes, la réponse est simple :

- Les `int` ne sont jamais envoyés de manière directe puisque la taille en octets d'un `int` peut différer d'une machine à l'autre.
- Le choix du boutisme est fixé par le protocole de communication, indépendamment des boutismes respectifs des deux machines.
- Les `int` émis sont d'abord convertis en un type dont la taille est par convention fixe d'une machine à l'autre : par exemple `uint32_t`, dont la taille sera toujours 4 octets (32 bits)

2 Lecture et écriture de données binaires

Deux fonctions permettent la lecture et l'écriture de données binaires dans un flot :

```
size_t
fread(void *restrict ptr, size_t len, size_t nitems,
    FILE *restrict stream);

size_t
fwrite(const void *restrict ptr, size_t len, size_t nitems,
    FILE *restrict stream);
```

Chaque fonction suppose que `ptr` est l'adresse d'un vecteur.
La fonction `fwrite` suppose que ce vecteur contient `nitems` objets, chacun de taille `len` octets. Elle écrit séquentiellement ces objets dans le flot `stream`.
La fonction `fread` lit successivement depuis le flot `stream` `nitems` objets, chacun de taille `len` octets, et les stocke séquentiellement dans le vecteur.
Les deux fonctions retournent le nombre d'objets lus/écrits.
A noter que `fread` ne fait pas de distinction entre une erreur de lecture et l'atteinte de la fin du flot. Pour distinguer ces deux cas on peut utiliser les fonctions :

```
int ferror(FILE *flot)
void clearerr(FILE *flot)
int feof(FILE *flot)
```

La fonction `ferror` renvoie une valeur non nulle dès qu'un appel de `fread` ou `fwrite` échoue à cause d'une erreur. L'indicateur d'erreur reste levé tant qu'il n'a pas été réinitialisé par un appel de `clearerr`.
La fonction `feof` renvoie une valeur non nulle dès que la fin du flot est atteinte. La encore, même si l'on se repositionne dans le fichier, l'indicateur de fin de flot reste levé tant qu'il n'a pas été réinitialisé par un appel de `clearerr`.

Inutilité de l'écriture de pointeurs. Même s'il est techniquement possible d'écrire des valeurs de pointeurs dans un fichier binaire, cela ne présente aucun intérêt. Par exemple, cela n'a pas vraiment de sens de sauvegarder dans un fichier une variable de type

```
typedef struct{
  char *nom;
```

```
  char *prenom;
  int age;
} personne;
```

Les champs `nom` et `prenom` sont des pointeurs vers des strings, mais ce qu'il est évidemment nécessaire de sauvegarder ici, ce sont les strings eux-mêmes, pas leurs adresses.

2.1 Exemples

Sauvegarde d'un vecteur d'`int`.

```
int tab2file(const char *nom, size_t len, int *tab){
  FILE *flot = fopen(nom, "w");
  if(flot == NULL)
    return -1;

  if( fwrite( tab, sizeof(int), len, flot) = 0 ){
    fclose(flot);
    return -1;
  }

  fclose(flot);
  return 0;
}
```

Modification d'un vecteur sauvegardé. Considérons un vecteur d'`int` sauvegardé à l'aide de la fonction de l'exemple précédent. On souhaite modifier cette sauvegarde en ajoutant `n` à son élément de position `i` dans le vecteur.

```
int updateFile(const char *nom, int n, size_t i){
  FILE *flot = fopen(nom, "r+");
  if( flot == NULL )
    return -1;

  /* se positionner sur le ieme int */
  if( fseek(flot, i*sizeof(int), SEEK_SET ) < 0)
    goto err;

  int u ;

  size_t r = fread( &u, sizeof(int) , 1, flot);
  if(r == 0)
    goto err;

  u += n;
  /* revenir sur le ieme int */
  if( fseek(flot, -sizeof(int), SEEK_CUR ) < 0)
    goto err;

  /* sauvegarder la nouvelle valeur */
```

```
22   if( fwrite( &u , sizeof(int), 1, flot ) == 0 )
23     goto err;
24
25   fclose(flot);
26   return 0;
27
28 err:
29   fclose(flot);
30   return -1;
31 }
```

Ecriture d'une structure

```
1 typedef struct{
2   double x;
3   double y;
4 } point;
5 ...
6 point a = { .x = -33.4, .y = 21.98 };
7 fwrite( &a, sizeof(a), 1, flot );
```