Chapitre 5
Fichiers

Plan du chapitre 5

1. Introduction

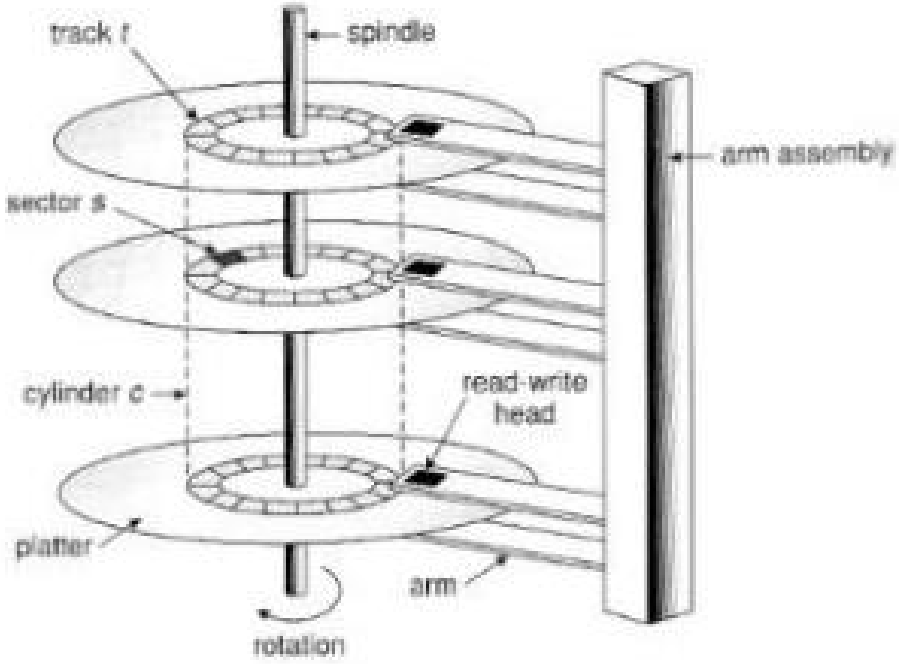
Les fichiers informatiques

- Définition [https://fr.wikipedia.org/wiki/Fichier_informatique]
  - « Un fichier informatique est … un ensemble de données numériques réunies sous un même nom, enregistrées sur un support de stockage permanent … tel qu'un disque dur, un CD-ROM, une mémoire flash ou une bande magnétique, et manipulées comme une unité. … [Leur] manipulation … est un des services classiques offerts par les systèmes d'exploitation. »

- Généralisation [https://en.wikipedia.org/wiki/Everything_is_a_file]
  - en C (pour tous les OS), `stdin`, `stdout` et `stderr` sont définis dans `<stdio.h>` également comme des fichiers (ou flux)

Façons de manipuler les fichiers

- Les fichiers sont en fait une abstraction fournie par le système d'exploitation
  (réalité : plateaux, pistes, secteurs/blocs,
  ou mémoire flash si SSD)
- C étant proche de l'OS, il offre dans sa librairie standard de nombreuses fonctions de manipulation des fichiers
- Deux grands types d'accès
  - **séquentiel** versus **direct** (*random access*)
- Une structure de données principale
  - **flux (*streams*)**



2. Les fichiers comme flux (streams) : ouverture

Les fichiers comme flux (*stream*)

- Comment identifier le fichier sur lequel on travaille ?
- `<stdio.h>` fournit une structure complexe décrivant un fichier, dont on manipulera toujours un pointeur vers une instance
  - on passe ce pointeur pour identifier le fichier lorsqu'on appelle des fonctions
  - on dit que ce pointeur pointe vers un flux (*stream*) ou un fichier
- Les flux fournissent une interface unifiée de haut niveau par-dessus les descripteurs de fichiers plus élémentaires (chemin, date, créateur, droits, etc.)
  - `stdin`, `stdout` et `stderr` sont des flux prédéfinis

```c
#include <stdio.h>
FILE* fp = NULL;
```

```c
typedef struct _iobuf {
   char* _ptr;
   int    _cnt;
   char* _base;
   int    _flag;
   int    _file;
   int    _charbuf;
   int    _bufsiz;
   char* _tmpfname;
} FILE;
```

Comment ouvrir un fichier ?

```
FILE* fopen(const char* filename, const char* mode);
```

- ouvre le fichier (externe) dont le nom est `filename`
- associe à ce fichier un pointeur `FILE*` qui est retourné si l'ouverture a réussi
- si l'ouverture échoue, retourne `NULL` et écrit un code d'erreur dans la variable globale **`errno`**
  - A noter aussi :
    - `int ferror(FILE* stream)` : livre une valeur différente de 0 si une erreur d'E/S est survenue (c'est-à-dire si l'indicateur d'erreur associé au flux est activé)
    - `void clearerr(FILE* stream)` : remet à zéro l'indicateur d'erreur associé au flux

Traitement des erreurs d'ouverture

- La variable globale (unique) **errno**, dans <errno.h>, conserve le code (int) de la dernière erreur d'E/S survenue
- On dispose aussi de quoi afficher les messages associés aux codes d'erreur

```
#include <stdio.h>
void perror(const char* str);
```

  - affiche la chaîne str (choisie par le programmeur) puis le message de la dernière erreur stockée dans errno

```
#include <string.h>
char* strerror(int errnum);
```

  - retourne la chaîne correspondant à un numéro d'erreur donné

Exemple

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

int main(void) {
  FILE* fichier;
  fichier = fopen("fichier.txt", "r");
  if (fichier == NULL) {
     perror("Error");
     return EXIT_FAILURE;
  }
  fclose(fichier);
  return EXIT_SUCCESS;
}
```

Error : No such file or directory

A noter l'ajout automatique du ':'

Ouverture d'un fichier : modes

```
FILE* fopen(const char* filename, const char* mode);
```

- 2e paramètre : comment on souhaite accéder au fichier

- **"r"** : en **lecture** seulement; le pointeur de flux est positionné au début du fichier, qui doit exister
- **"w"** : en **écriture** seulement; le pointeur de flux est positionné au début du fichier; si le fichier n'existe pas, il est créé; sinon, il est écrasé (son ancien contenu est perdu)
- **"a"** : seulement pour **ajouter** du contenu à la fin; le pointeur de flux est positionné à la fin du fichier; le fichier est créé s'il n'existe pas

Ouverture d'un fichier : modes

- **"r+"** : **en lecture et en écriture**; le fichier doit exister; le pointeur de flux est positionné au début
  - après des lectures / écritures, penser à repositionner le pointeur de flux avec `fseek` (voir accès direct, slide 31)
- **"w+"** : **en lecture et en écriture**, comme r+; si le fichier existe, son contenu est écrasé; s'il n'existe pas, il est créé; le pointeur de flux est positionné au début
- **"a+"** : **en lecture et ajout**; le fichier est créé s'il n'existe pas; pour les lectures, le pointeur de flux est positionné au début; mais les écritures se font toujours à la fin
- **"t"** ou **"b"** : indiquent un fichier **texte** ou bien **binaire**; on les ajoute aux 6 modes précédents (p.ex. rb+ ou r+b)
  N.B. "t" est rarement utilisé en pratique

Fermeture d'un fichier

```
int fclose(FILE* stream);
```

- ferme le fichier, vide le tampon associé au flux, et libère les ressources allouées à ce flux
  - l'appeler quand l'accès à un fichier n'est plus nécessaire ou lorsqu'un autre mode d'accès est souhaité
- en cas de succès, retourne 0
- sinon retourne la valeur `EOF` définie dans `<stdio.h>` et écrit le code de l'erreur dans `errno`
- un autre accès au flux (y compris un autre appel de `fclose`) produit des résultats indéfinis (UB)

3. Fichiers texte vs fichiers binaires

Que signifie le « type » d'un fichier ?

- Tous les fichiers sont stockés comme des suites d'octets
  - pas de différence au niveau du stockage sur disque
- **Le type d'un fichier indique la façon de l'interpréter**
  - comment on va lire/écrire le fichier
  - quelles fonctions seront disponibles pour lire/écrire
- Différence principale en C : **fichiers texte vs binaires**
  - **texte** : octets lus comme des caractères, dont certains ont un sens particulier, p.ex. les fins de ligne (CR/LF)
  - **binaire** : lus octet par octet sans interprétation
- Exemple : écrire `int n = 123;` dans un fichier binaire (01111011) ou texte (00110001 00110010 00110011)

Indication du type de fichier en C

- Par défaut, C ouvre les fichiers comme des fichiers texte
  - chaque octet sera interprété comme un caractère
  - pas adéquat pour les informations numériques
- On peut indiquer à C comment on souhaite lire/écrire le fichier
  - format binaire (i.e. suite d'octets lus comme des nombres) : "**b**"
  - après le mode, après ou avant le + : "rb", "wb", "ab",
    ou bien "r+b", "w+b", "a+b", ou "rb+", "wb+", "ab+"
  - indication explicite éventuelle du mode texte : "**t**"
- Considérations sur la vitesse : (2) plus rapide que (1)
  1. E/S sur fichier texte : C doit convertir entre des valeurs en mémoire et des suites de caractères
  2. E/S sur fichier binaire : copie directe vers/depuis la mémoire

Fichiers binaires

- Fichiers lus comme un flux d'octets (nombres), sans aucune interprétation prédéfinie
- **Pour pouvoir utiliser l'information qu'ils contiennent, on doit savoir comment celle-ci est structurée**
  - la structure repose sur des conventions / standards
  - exemples : images (JPEG, PNG), vidéos (MPEG, MP4), sons (WAV, MP3), documents (DOC, ODT)
  - le type est indiqué par des extensions à la fin du nom du fichier (Windows) ou grâce au type **MIME** (*Multipurpose Internet Mail Extensions*) qui apparaît au début d'un fichier (Mac OS, Linux)

Exemples : WAVE et JPEG

endian	File offset (bytes)	field name	Field Size (bytes)
big	0	ChunkID	4
little	4	ChunkSize	4
big	8	Format	4
big	12	Subchunk1ID	4
little	16	Subchunk1Size	4
little	20	AudioFormat	2
little	22	NumChannels	2
little	24	SampleRate	4
little	28	ByteRate	4
little	32	BlockAlign	2
little	34	BitsPerSample	2
big	36	Subchunk2ID	4
little	40	Subchunk2Size	4
little	44	data	Subchunk2Size

The "RIFF" chunk descriptor

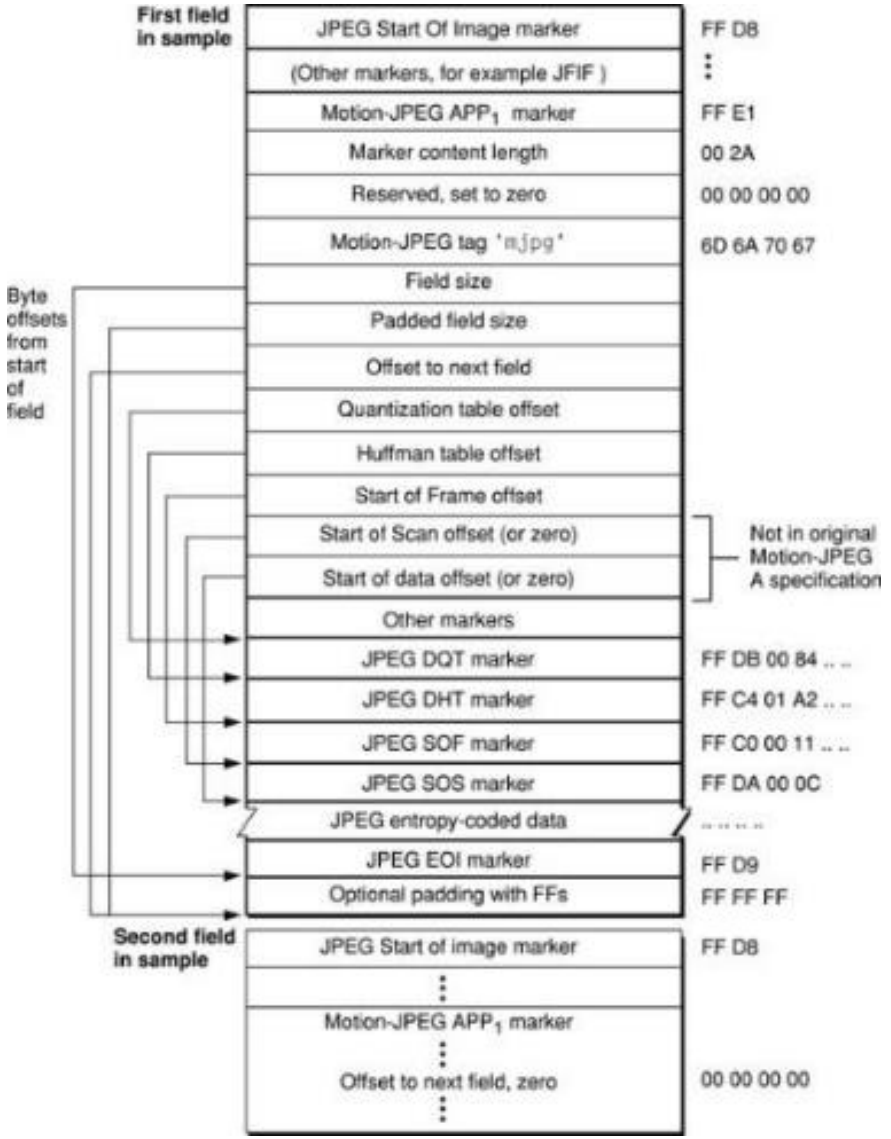
The Format of concern here is "WAVE", which requires two sub-chunks: "fmt " and "data"

The "fmt " sub-chunk

describes the format of the sound information in the data sub-chunk

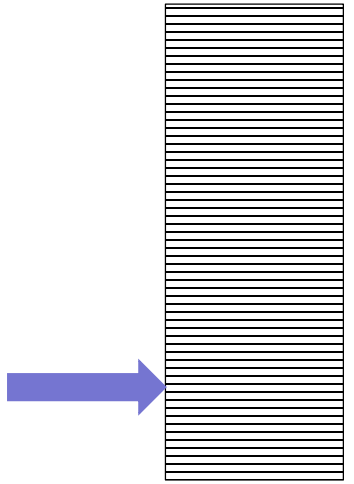
The "data" sub-chunk

Indicates the size of the sound information and contains the raw sound data



Attention aux distinctions suivantes

1. **Mode d'ouverture du fichier** : binaire ou texte
2. **Accès aux données** : séquentiel ou direct
3. **Opérations** : formatées, binaires, mixtes
   - **formatées** : `fprintf, fscanf, fputs, fgets,` plutôt destinées à des fichiers texte, mais appliquées parfois sur des flux binaires
   - **binaires** : `fwrite, fread`
   - **mixtes** : `fputc, fgetc` (et les macros `putc`, `getc`)
   - peuvent être utilisées pour des accès directs ou séquentiels, car ces accès ne diffèrent que par les fonctions sur le pointeur de fichier



4. Lecture et écriture dans les fichiers texte

Lecture des fichiers texte

```
int   fgetc(FILE* stream);
char* fgets(char* s, int size, FILE* stream);
int   fscanf(FILE* stream, const char* format, ...);
```

- fonctions analogues à celles déjà utilisées pour `stdin`
- `fgetc` lit un caractère dans le flux `stream`, et retourne sa valeur comme un `int`, ou bien `EOF` dans les cas suivants
  - fin de fichier, position hors fichier, erreur matérielle, flux incorrect
  - ❖ tester le cas `EOF` avant de convertir le `int` en `char` !
- `fscanf` est analogue à `scanf` : lit des caractères dans `stream` et affecte les chaînes aux adresses fournies
  - retourne le nombre de chaînes lues, ou `EOF` comme `fgetc`

Lecture des fichiers texte

```
char* fgets(char* s, int size, FILE* stream);
```

- lit des caractères depuis le flux `stream` et les stocke dans la chaîne `s` (note : `stream` peut être également stdin)
- s'arrête lorsque `size-1` caractères ont été lus, ou sur un retour à la ligne, ou lorsque la fin de fichier a été rencontrée
  - si arrêt sur un retour à la ligne, il est inclus dans `s`
  - un '\0' est automatiquement ajouté à la fin de `s`

```
int feof(FILE* stream);
```

- si l'indicateur de fin de fichier a été renseigné par une E/S précédente, retourne une valeur non-nulle, sinon zéro
- pour que l'indicateur soit renseigné, il faut tenter de lire le flux
- indicateur remis à zéro par `fseek`, `clearerr`, `rewind`, etc.

Écriture dans les fichiers texte

```
int fputc(int c, FILE* stream);

int fputs(const char* s, FILE* stream);

int fprintf(FILE* stream, const char* format, ...);
```

- `fputc` écrit un caractère (`int c` converti en `char`), et retourne `EOF` si impossible (p.ex. manque de place)
- `fputs` écrit une chaîne `s` sur le flux, sans le '\0' final (ni '\n'), résultat ≥ 0 si ok, ou `EOF` si erreur (même ex.)
- `fprintf` fonctionne comme `printf`, résultat < 0 si erreur
  (note : penser à ajouter les fins de ligne)

5. Lecture et écriture dans les fichiers binaires

Lecture de fichiers binaires : fread

```
size_t fread(void* ptr, size_t size, size_t count, FILE* stream);
```

- lit un nombre d'éléments égal à `count`, chacun de taille `size` (en octets) depuis le fichier `stream` et les stocke à l'adresse `ptr` (zone mémoire)
- renvoie le nombre d'éléments lus avec succès, ou 0 si size ou count vaut 0
- ne distingue pas entre une fin de fichier et une (vraie) erreur de lecture, donc on doit utiliser `feof` et/ou `ferror` pour déterminer ce qui s'est passé

Ecriture de fichiers binaires : fwrite

```
size_t fwrite(const void* ptr, size_t size, size_t count, FILE* stream);
```

- écrit `count` éléments de taille `size` (en octets) depuis la zone mémoire commençant à `ptr`, vers le flux `stream`
- renvoie le nombre d'éléments écrits avec succès, ou 0 si size ou count vaut 0

Fonctions « mixtes » : fgetc, fputc

- La lecture et l'écriture de caractères (octets) s'appliquent aussi bien aux fichiers binaires qu'aux fichiers texte
- Exemple simplifié : recopie brute d'un fichier (sans traitement d'erreurs)

```c
#include <stdio.h>
#include <stdlib.h>
int main(void) {
  int caractere;
  FILE *source, *dest;
  source = fopen("fichier1.dat", "rb");
  dest   = fopen("fichier2.dat", "wb");
  while ((caractere = fgetc(source)) != EOF) {
     fputc(caractere, dest);
  }
  fclose(source);
  fclose(dest);
  return EXIT_SUCCESS;
}
```

6. Accès direct

Accès direct

- Le langage C permet un traitement en accès direct
  - par opposition à l'accès *séquentiel*
  - concerne le positionnement du pointeur de flux
  - s'applique aux fichiers texte ou binaires
  - indépendant des modes d'ouverture ou fermeture

- Exemple

```
void rewind(FILE* stream);
```

  - permet de (re)positionner le pointeur de fichier au début

Accès direct : positionnement

```
int fseek(FILE* stream, long offset, int origin);
```

- positionne à un endroit du fichier le pointeur de fichier :
  la position (en octets) est égale à `origin + offset`
  - `origin` peut prendre l'une des valeurs prédéfinies :
    SEEK_SET, **SEEK_CUR**, **SEEK_END**
    c'est-à-dire : début du fichier, position courante, ou fin du fichier
  - `offset` peut ou doit parfois être négatif

```
long ftell(FILE* stream);
```

- fournit la position actuelle du pointeur de fichier (en octets par rapport au début), ou -1 si erreur

Exemple d'utilisation de `fseek`

- Calculer le nombre d'enregistrements dans un fichier
- Dans cet exemple, 1 enregistrement = 1 entier (`int`)
- Méthode :
  - se positionner à la fin du fichier
  - demander la position (octets)
  - diviser par la taille d'un enregistrement
  - se repositionner au début du fichier si nécessaire

```
fseek(fichier, 0, SEEK_END);
dernier = ftell(fichier) / sizeof(int);
fseek(fichier, 0, SEEK_SET);
```

- **N.B**. On dispose aussi de `fgetpos` et `fsetpos` (peu utilisées)

Vider les tampons de lecture ou écriture

```
int fflush(FILE* stream);
```

- pour un **flux de sortie**, force l'écriture des données stockées dans le tampon (vide le buffer)
- pour un **flux d'entrée**, le comportement dépend de l'implémentation. Dans certaines implémentations revient à vider le tampon de lecture.
- avec l'argument NULL, vide tous les tampons d'écriture
- retourne 0 si succès, sinon EOF et modifie `errno`

7. Autres fonctions sur les fichiers

Manipulation de fichiers

```
int remove(const char* name);
```

- efface un fichier du disque, qui ne doit pas être ouvert lors de l'appel

```
int rename(const char* old, const char* new);
```

- renomme un fichier; si le nouveau nom correspond à un fichier existant, le résultat dépend de l'implémentation

- `remove` et `rename` retournent 0 en cas de succès
- `<dirent.h>` permet l'accès aux répertoires
  (avec `DIR*`, puis `opendir, readdir, closedir, rewinddir, telldir, seekdir`)

Fichiers temporaires gérés par C

```
FILE* tmpfile(void);
```

- crée un fichier temporaire en mode « wb+ », qui sera automatiquement détruit en fin de programme

```
char* tmpnam(char* name);
```

- génère un nouveau nom de fichier valide, sans le créer
- le nom n'est celui d'aucun fichier existant
  - si `name` n'est pas NULL, le nom est fourni dans `name`, sinon comme résultat de la fonction

```
char* name = tmpnam(NULL);
printf("%s", name);
/* affiche p.ex. \sgv8.
```

8. Résumé

Résumé

Type de fichier

- Texte
- Binaire

Accès

- Séquentiel (tester avec `feof`)
- Direct (`fseek, ftell, rewind, feof`)

Lecture, écriture

- Formatée (`fprintf, fscanf, fputs, fgets`)
- Binaire (`fread, fwrite`)
- Mixte (`fputc, fgetc, putc, getc`)

Opérations

- Ouverture (fopen), fermeture (fclose)
  - modes (r, w, a, +, t, b)
  - erreurs (ferror, perror, strerror, clearerr, errno)
  - vidage flux (fflush)
- Déplacement, etc. (`rename, remove, tmpfile, tmpnam`)

L'accès aux fichiers dans le « style C » reste valable en C++.

Toutefois, C++ définit aussi les classes *ofstream* et *ifstream* dans <fstream>.