Chapitre 4

Les fichiers

Les données que manipule un programme sont placées en mémoire centrale RAM (Random Access Memory). Ce type de mémoire est caractérisée par la volatilité des données c.à.d elles disparaissent à la fin de l' exécution. D'autre part, la mémoire centrale RAM de l'ordinateur a une capacité finie, et un programme ne pourra pas mémoriser des données d'une taille supérieure à celle de la mémoire centrale.

La solution consiste à utiliser une mémoire plus volumineuse qui permet de stocker les données sans trop se soucier de la capacité, et en même temps elle ne soufre pas de la volatilité des données. Donc la solution est d'utiliser les fichiers qui utilise les mémoires secondaires comme support de stockage.

Donc le rôle principale des fichiers est de conserver de l'information sur des supports externes, en particulier des disques. Mais, dans bien des systèmes, comme Unix par exemple, les fichiers ne se limitent pas à cette fonction, ils sont utilisés également dans les opérations d'entrée et de sortie standard (i.e. le clavier et l'écran de l'ordinateur), les périphériques, des moyens de communication entre processus ou réseau, etc.

4.1 Les fichiers textes

Un fichier texte contient du texte ASCII. Lorsqu'un fichier texte contient des nombres, ces nombres sont codés sous forme de texte à l'aide des caractères '1', '2', etc. Dans ce format, chaque chiffre prend 1 octet en mémoire. On peut visualiser le contenu d'un fichier texte avec un éditeur de texte.

4.1.1 Déclaration d'un fichier

Les fichiers séquentiels modélisent la notion de suite d'éléments telle que l'on ne peut accéder à un élément qu'après avoir accédé à tous ceux qui le précèdent. Lors du traitement d'un fichier séquentiel, à un moment donné, un seul composant du fichier est accessible, celui qui correspond à la position courante du fichier.

La syntaxe de la déclaration d'un fichier est comme suit : Cette déclaration

Syntaxe
<id_fichier_ram> : fichier <type> ;

définit la variable f comme un fichier dont tous les éléments sont de même type *type*, ça peut être n'importe quel type à l'exception du type fichier. Les fichiers de fichiers ne sont donc pas autorisés.

En C, les fichiers sont mis en œuvre par le support d'exécution, c.à.d la bibliothèque standard *stdlib.h* qui possède de nombreuses fonctions qui permettront leur manipulation de façon séquentielle.

En C, un fichier est vu comme une suite linéaire d'octets sans structure particulière. Le langage ne prend donc pas en charge le type des éléments d'un fichier. Ce sera donc au programmeur de le faire.

Un programme C manipule un fichier par l'intermédiaire d'un descripteur accessible par un pointeur. Le descripteur de fichier est de type FILE, défini dans le fichier *stdio.h* qu'il faut donc inclure.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 FILE * <id_file>;
```

4.1.2 Manipulation des fichier

Les fichiers séquentiels se manipulent par le bais des fonctions suivantes.

Ouverture d'un fichier

Avant d'utiliser un fichier il faut impérativement l'ouvrir. Une fonction prédéfinie OuvrirFichier permet de le faire. Pour ouvrir un fichier, il faut définir son mode d'ouverture qui peut être :

- **Lecture** : Il faut noter que plusieurs lectures peut se faire en même temps c'est à dire plusieurs programmes peuvent utiliser un fichier en lecture en même temps.

- **Écriture** : Quand on ouvre un fichier en écriture un seul programme est autorisé à le faire. L'écriture qu'on va opérer va écraser le contenu précédant.
- **Ajout** : C'est ouverture en écriture, la différence avec le mode d'ouverture en écriture est qu'on garde le contenu du fichier et on rajoute le nouveau contenu à la fin du fichier.

Le mode d'ouverture est représenté par une constante qui est déterminée dans la commande d'ouverture du ficher. Sa syntaxe est comme suit :

Syntaxe

$id\_fichier\_ram \leftarrow OuvrirFichier(< cheminFichier >, ModeOuverture)$

En C, la fonction $fopen$ permet d'ouvrir un fichier en lecture ou en écriture et renvoie le pointeur du déscripteur de fichier ouvert. La fonction $fopen$ admet deux chaînes de caractères en paramètre, le nom du fichier (y compris le chemin pour y accéder) ainsi que le mode d'ouverture :

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

FILE * <id_file>=fopen(<chemin_file>,<mode_ouverture>);
```

Où mode ouverture peut prendre les valeurs suivantes :

- "r" : lecture
- "w" : écriture
- "a" : rajout
- "r+" : mode lecture-écriture. Le fichier est prêt pour lire et écrire au début du fichier. Le fichier n'est pas écrasé. Lorsqu'on écrit une donnée, celle-ci remplace la donnée qui se trouvait éventuellement à cet emplacement sur le disque dans le fichier.
- "w+" : mode lecture-écriture. Le fichier est écrasé.
- "a+" : mode lecture-écriture. Le fichier n'est pas écrasé mais est prêt pour écrire à la suite des données existantes.

Si l'ouverture n'a pas pu se faire (e.g. fichier inexistant) la fonction renvoie la valeur $NULL$.

Le fragment de code suivant ouvre le fichier de nom nom_fich en lecture. Si l'ouverture a réussi le fichier sera accessible par le pointeur fd, sinon le programme s'achève avec un message et un code d'erreur.

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


FILE *fd;
```

```
if ((fd = fopen("nom_fich", "r")) == NULL) {
printf("Erreur ouverture de fichier");
return 1;
}
/* le fichier "nom_fich" est ouvert en lecture
* et fd pointe sur son descripteur de fichier
*/
```

Fermeture d'un fichier

Après avoir ouvert un fichier, il doit etre refermé afin de liberer les ressources que le système avait réservé lors de l'ouverture (par exemple le pointeur de descripteur associé au fichier en C). Elle est realisé par la fonction :

Syntaxe
FermerFichier (id_fichier_ram) ;

La syntaxe en C est comme suit :

```
fclose(descripteur);
```

Remarque 6 *Il préférable de fermer le fichier juste après avoir fini de l'utiliser puisque ça peut provoquer des erreurs lors de l'exécution surtout avec le mode d'ouverture écriture.*

Lecture et écriture par octet

La lecture et l'écriture se fait octet par octet à l'aide des fonctions. En algorithmique avec Lire et Ecrire. En C, avec les fonctions **fgetc** et **fputc** respectivement. La première renvoie le prochain octet à lire, et la seconde écrit l'octet qu'elle possède en paramètre.

Syntaxe
Lire (id_fichier,variable) ;
Ecrire(id_fichier,donner) ;

La syntaxe en C est comme suit :

```
char fgetc(descripteur);
void fputc(donner,descripteur);
```

La fin de fichier est représentée par :

- En algorithmique : la fonction FDF(fichier) qui revoie vrai ou faux

- En C : la constante EOF, définie dans le fichier stdio.h à inclure.

Algorithme Lecture$_{ap}artir_{f}ichier$

Var :
fd : Fichier de char ;
c : char ;
s : chaine ;
Début
fd=OuvrirFichier("fichier.txt","Lecture") ;
SI (fd == NULL) **alors**
STOP() ;
Faire
Lire(fd,c) ;
Ecrire(c) ;
Tans que (! FDF(fd)) ;
Fermer(fd) ;
s← "J'ecris sur le fichier avec Ecrire" ;
fd=OuvrirFichier("fichier.txt","Ecriture") ;
SI (fd == NULL) **alors**
STOP() ;
Ecrire(fd,s) ; **Fin.**

```c
#include <stdio.h>
#include <string.h>
FILE *fd;
int c;
fd=fopen("fichier.txt","r");
if(fd==NULL)
return;
while ((c=fgetc(fd)) != EOF) {
printf("%c",c);
}
/* on a lu tout le fichier caractere par caractere jusqu a tomber
    sur le caractere EOF qui marque la fin du fichier */
fclose(fd);

char * text="J'ecris sur le fichier avec fputc";
fd=fopen("fichier1.txt","w");
if(fd==NULL)
return;

int i;
for(i=0;i<strlen(text);i++)
fputc(fd,text[i]);
```

```
22
23 fclose(fd);
24 }
```

```
$> je suis un message lu avec fgetc
```

Le contenu du fichier "fichier1.txt" :

```
$> J'ecris sur le fichier avec fputc
```

Lecture et écriture formatées

On vu dans la section 4.1.2 que la lecture et l'écriture se fait octet par octet. Donc dans le cas d'un mot, chaque lettre est lu et écrite une à une.

En C, on dispose aussi des fonctions **fscanf** et **fprintf** qui nous permet de lire et écrire en format formaté, par exemple, pour un mot on va l'écrire directement non pas lettre par lettre. Donc elles permettent la lecture ou l'écriture d'objets de type élémentaire après conversion sous forme d'une suite caractères de la valeur à lire ou à écrire. Le premier paramètre de chaque fonction est un pointeur sur le descripteur de fichier ouvert. Pour le reste, c'est similaire aux fonctions scanf et printf déjà vu.

```
1 FLIE *fp;
2 int n=10;
3 if ((fp=fopen("fichier.txt", "w")) == NULL)
4 {
5 printf("Permission refus e ou r pertoire inexistant");
6 exit(1);
7 }
8 /*   criture   du nombre n */
9 fprintf(fd, "%d",n);
```

```
1 #include <stdio.h>
2 main(){
3     FILE *fp;
4     char buff[255];//creating char array to store data of file
5     fp = fopen("fichier.txt", "r");
6     while(fscanf(fp, "%s", buff)!=EOF){
7     printf("%s ", buff );
8     }
9     fclose(fp);
10 }
```

```
$> 10
```

Remarque 7 *Les fichiers d'entrée standard, de sortie standard et d'erreur standard sont prédéfinis, et accessibles par l'intermédiaire des variables de*

type pointeur sur FILE, stdin, stdout et stderr. Ces trois variables sont définies dans le fichier stdio.h.
vous pouvez remplacer printf par fprintf comme suis :

```
1 fprintf(stdout,"j'utilise fprintf    la palace de printf");
```

4.2 Les fichiers binaire

Un fichier binaire contient du code binaire. On ne peut pas visualiser son contenu avec un éditeur de texte. Lorsqu'une variable est écrite dans un fichier binaire, on écrit directement la valeur exacte de la variable, telle qu'elle est codée en binaire en mémoire.

Cette manière de stocker les données est plus précise et plus compacte pour coder des nombres.

4.2.1 Lecture à partir d'un fichier binaire

Pour lire dans un fichier binaire, les éléments de ce fichiers peuvent être vues comme des éléments d'un tableau. Chaque élément du tableau est appelé un bloc. Chaque bloc possède une taille en octets.

Exemple 7 *Soit un fichier binaire dont le contenu est présenté sur la Figure 4.1*

- *B1 correspond à un bloc de 4 octets qui peut être un entier ou un réel*
- *B2 correspond à un bloc de 8 octets qui peut être un double*
- *B3 correspond à un bloc de 1 octet qui peut être un char*

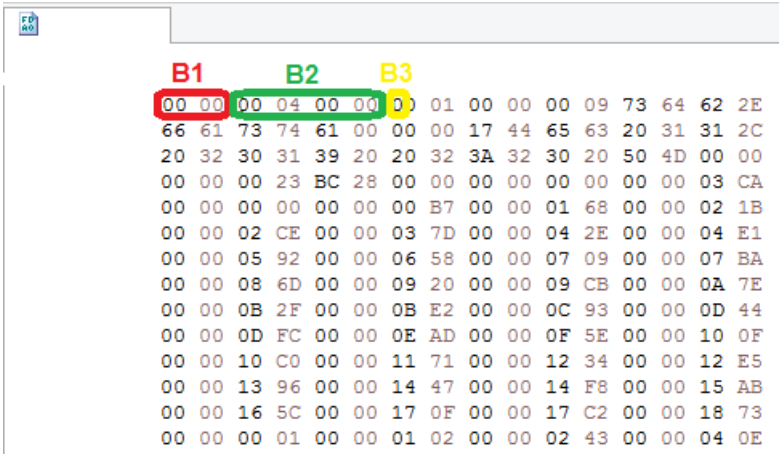

FIGURE 4.1 – Contenu d'un fichier en binaire

La fonction ***sizeof*** donne la taille de chaque type. Par exemple :

- sizeof(char) = 1,
- sizeof(float) = 4

On utilisera de préférence la fonction sizeof plutôt qu'une constante comme 1 ou 4 car cela augmente la lisibilité du programme et le programme ne dépend pas du compilateur ou du système.

Pour lire on utilise la fonction **fread(void * buffer, size_t blocSize, size_t blocCount, FILE * stream)**. Les paramètres de cette fonction sont comme suit :

- **void * buffer :** Le premier paramètre de cette fonction est l'adresse de la variable qui contient la valeur à écrire, ou à qui sera affectée la valeur lue.
- **size_t blocSize :** Le deuxième paramètre est le nombre d'octets nécessaires à la représentation du type de l'élément.
- **size_t blocCount :** Le troisième paramètre est le nombre d'éléments à écrire.
- **FILE * stream :** Le dernier paramètre est le pointeur sur le descripteur de fichier.

Exemple 8 *Pour lire une variable entière à partir d'un fichier représenté dans la Figure 4.1. Le bloc que nous allons lire correspond au bloc B1. Donc il suffit pour cela de mettre l'adresse x de la variable et de mettre le nombre de blocs égal à 1 ; les données sont alors transférées dans la variable.*

```
1 int x;
2 FILE *fp;
3 if ((fp=fopen(nomFichier, "r")) == NULL)
4 {
5 printf("Erreur :");
6 exit(1);
7 }
8 fread(&x sizeof(int), 1, fp); /* on lit le nombre d'  lments    */
9 printf("X=%d",x);
```

```
$> X=0
```

4.2.2 Écriture dans un fichier binaire

Pour écrire dans un fichier binaire, on utilise la fonction ***fwrite*** qui transfère des données de la mémoire centrale vers un fichier binaire. Comme la fonction fread, la fonction fwrite prend en paramètre le tableau, la taille de

chaque bloc, le nombre de blocs à écrire et le pointeur de fichier. La taille physique du tableau doit être au moins égale au nombre de blocs écrits, pour éviter une erreur mémoire. La fonction fwrite transfère les données du tableau vers le fichier binaire.

La fonction fwrite retourne le nombre d'éléments effectivement écrits. Si ce nombre est inférieur au nombre effectivement demandé, il s'est produit une erreur d'écriture (fichier non ouvert, disque plein...).

```
FLIE *fp;
int n=10;
if ((fp=fopen(nomFichier, "w")) == NULL)
{
printf("Permission refus e ou r pertoire inexistant");
return 1;
}
/*   criture   du nombre n */
fwrite(&n, sizeof(int), 1, fp);
```

Les paramètres sont similaires aux paramètres de la fonction fread.

Remarque 8

- *Le langage C ne gère pas le type des éléments d'un fichier. Il ne propose pas d'équivalent à notre déclaration algorithmique f : fichier de T. Ce sera au programmeur de gérer le type des éléments à l'aide des fonctions.*
- *On teste habituellement la fin de fichier en vérifiant si le nombre d'éléments effectivement lus est supérieur à 0 ou pas. La fonction fread renvoie le nombre d'éléments effectivement lus. Si la fin de fichier est atteinte prématurément, il sera inférieur à celui spécifié dans l'appel de la fonction*

4.2.3 Se positionner dans un fichier binaire

À chaque instant, un pointeur de fichier ouvert se trouve à une position courante, c'est-à-dire que le pointeur de fichier est prêt pour lire ou écrire à un certain emplacement dans le fichier.

Chaque appel à fread ou fwrite fait avancer la position courante du nombre d'octets lus ou écrits. La fonction **fseek** permet de se positionner dans un fichier, en modifiant la position courante pour pouvoir lire ou écrire à l'endroit souhaité. Lorsqu'on écrit sur un emplacement, la donnée qui existait éventuellement à cet emplacement est effacée et remplacée par la donnée écrite.

int **fseek**(FILE *fp, long offset, int origine) ;

La fonction modifie la position du pointeur fichier fp d'un nombre d'octets égal à offset à partir de l'origine.

L'origine peut être :
SEEK_SET : on se positionne par rapport au début du fichier ;
SEEK_END : on se positionne par rapport à la fin du fichier ;
SEEK_CUR : on se positionne par rapport à la position courante actuelle (position avant l'appel de fseek).

- **Exemple 9**

Si on veux retourner au début du fichier :

```
1 fseek(fich, 0, SEEK_SET);
```

Si on veux aller à 4 octets de la fin du fichier

```
1 fseek(fich, -4, SEEK_END);
```