Langage C – TD 4

1 Les fichiers

Jusqu'à présent, dans nos programmes, lorsque nous avions des informations à transmettre à l'utilisateur, nous les affichions sur la console. Il peut parfois s'avérer utile de les sauvegarder directement dans un fichier. De plus, la plupart des applications nécessitent des données en entrée (par exemple, lire une image ou un fichier explicitant des paramètres). Le C met donc à disposition un ensemble d'outils permettant de gérer les fichiers.

Les déclarations de types et de fonctions relatives à la gestion des fichiers se trouvent dans le fichier "`stdio.h`" (pour *STanDard Input–Output*).

Le type de base lorsqu'on travaille avec des fichiers est "`FILE*`". Il sert à décrire un fichier. Lorsque vous voulez accéder à un fichier, il faut l'associer à une variable de type `FILE*` que vous utiliserez comme une clé lors de l'appel aux fonctions touchant aux fichiers.

Mode opératoire :

1. Déclarer au système d'exploitation qu'on va utiliser un fichier donné (*ouvrir* le fichier) ;
2. Lire ou écrire dans le fichier ou déplacer le curseur sur le fichier ;
3. Déclarer au système d'exploitation que l'on n'utilise plus le fichier (*fermer* le fichier).

1.1 Ouverture et fermeture

Il existe trois modes d'accès à un fichier :

– Lecture : il sera possible de lire le fichier mais pas de le modifier. Le curseur est placé au début du fichier. Si le fichier n'existe pas, un fichier vide *n'est pas* créé.
– Écriture : il sera possible d'écrire dans le fichier mais pas de le lire. Le curseur est placé au début du fichier. Si le fichier n'existe pas, il est créé, s'il existait déjà, il est écrasé.
– Modification : il sera possible de lire et modifier le fichier. Le curseur est placé en fin de fichier. Si le fichier n'existe pas, il est créé.

On précise le mode d'accès lors de l'ouverture du fichier.

La fonction à utiliser pour ouvrir un fichier est la suivante :

```
FILE* fopen(char *nom_fichier, char *mode_acces);
```

Le paramètre "`mode_acces`" peut prendre 3 valeurs différentes :

`mode_acces`	Mode d'accès
`"r"`	ouverture en lecture seulement
`"w"`	ouverture en écriture seulement
`"a"`	ouverture en modification

`fopen` renvoie `NULL` si une erreur est survenue.

Pour fermer un fichier, une fonction simple existe :

```
void fclose(FILE *fichier);
```

Il est nécessaire d'appeler `fclose` pour tous les fichiers que vous avez ouvert ! Le cas échéant, il peut parfois s'avérer *nécessaire* de redémarrer l'ordinateur pour pouvoir à nouveau accéder aux fichiers que vous n'avez pas fermé !

Exemple :

```
{
        FILE *f;
        f = fopen("toto.txt", "r");
        if (f == NULL)    /* équivalent à << if (!f) >> */
        {
                printf("Erreur lors de l'ouverture de toto.txt!\n");
        }
        else
        {
                ...
                fclose(f);
        }
}
```

Trois variables supplémentaires sont déclarées dans `stdio.h` :

```
FILE *stdin, *stdout, *stderr;
```

Ce sont les entrées et sorties standard.

Fichier	Description
`stdin`	c'est entrée standard, par défaut : le clavier
`stdout`	c'est la sortie standard, par défaut : la console
`stderr`	c'est la sortie d'erreurs standard, par défaut : la console

Il est intéressant d'utiliser `stderr` pour écrire les erreurs car il existe des moyens de rediriger ce fichier.

Exemples (sous Unix) :

La commande "`toto >sorties.txt`" exécute le programme `toto` et enregistre toutes les sorties dans le fichier `sorties.txt`. Ces dernières ne sont donc pas affichées sur la console.

La commande "`toto 2>erreurs.txt`" exécute le programme `toto` et enregistre toutes les sorties d'erreurs dans le fichier `erreurs.txt`. Ces dernières ne sont donc pas affichées sur la console.

La commande "`toto <entrees.txt >/dev/null 2>erreurs.txt`" exécute le programme `toto`. Au lieu de lire les entrées sur le clavier, elles sont récupérées dans le fichier `entrees.txt` (par exemple, `scanf` ne lit plus ce que l'utilisateur écrit au clavier mais ce qui est déjà écrit dans le fichier `entrees.txt`). Les sorties sont jetées dans un puits sans fond et les sorties d'erreurs sont sauvegardées dans le fichier `erreurs.txt`.

1.2 Opérations de positionnement

Le curseur indique la position à laquelle aura lieu la prochaine lecture ou écriture dans le fichier. Si le curseur vaut 0, on est au début du fichier, s'il vaut 4, on se trouve au 4ème octet du fichier.

Ce curseur ne vous est pas accessible directement et il faut faire appel à des fonctions prédéfinies pour le consulter et le modifier.

Pour consulter le curseur sur un fichier, utilisez :

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

Pour déplacer le curseur, on utilise :

```
int fseek(FILE *fichier, long decalage, int origine);
```

Lorsque `fseek` renvoie 0, c'est que l'opération a réussi. Le cas échéant (par exemple si on essaie de déplacer le curseur hors des limites du fichier), la valeur renvoyée est non nulle (mais indéterminée).

Le paramètre `decalage` indique la distance par rapport à l'`origine` à laquelle on veut placer le curseur. Le décalage peut être négatif. L'origine est notée en utilisant les 3 constantes suivante :

origine	Position dans le fichier
SEEK_SET	début du fichier
SEEK_END	fin du fichier
SEEK_CUR	fin de la dernière lecture ou écriture

Exemples :
"`fseek(f, 0, SEEK_SET);`" déplace le curseur en début de fichier.
"`fseek(f, -4, SEEK_END);`" déplace le curseur à 4 octets avant la fin du fichier.
"`fseek(f, 1, SEEK_CUR);`" avance le curseur d'un octet.

Comment trouver la taille d'un fichier :

```
{
	FILE *f;
	f = fopen("toto.txt", "r");
	if (f == NULL)    /* équivalent à << if (!f) >> */
	{
		printf("Erreur lors de l'ouverture de toto.txt!\n");
	}
	else
	{
		int fin;
		fseek(f, 0, SEEK_END);
		fin = ftell(f);
		printf("Le fichier toto.txt a une longueur de %i\n", fin);
		fseek(f, 0, SEEK_SET);
		...
		fclose(f);
	}
}
```

1.3 Opérations de lecture/écriture

Il existe deux façons de gérer les fichiers :
- Mode binaire : toutes les données sont écrites telles quelles dans le fichier. Il est illisible par un être humain. Avant de programmer les accès à un tel fichier, il est nécessaire de définir formellement sa structure.
- Mode texte : les données sont écrites en tant que texte dans le fichier. Ce dernier est donc lisible par un être humain qui peut aussi le modifier. Il est possible d'y incorporer des mots-clés qui peuvent le rendre plus facile à relire, surtout s'il est susceptible d'être modifié à la main.

1.3.1 Accès binaire

Il est possible d'écrire la valeur d'une variable dans un fichier, ou bien l'ensemble des valeurs des membres d'un tableau ou d'une variable structurée. Pour cela, on utilise `fwrite` :

```
int fwrite(void *adresse, int taille, int nombre, FILE *fichier);
```

Paramètre	Description
adresse	Pointeur sur le premier élément à sauvegarder. Son type, `void*`, indique qu'il s'agit d'un pointeur pouvant pointer sur n'importe quel type de variable.
taille	Taille d'un élément à sauvegarder
nombre	Nombre d'éléments à sauvegarder
fichier	Descripteur du fichier

La valeur retournée est le nombre d'éléments entièrement sauvegardés. Cette valeur est différente de `nombre` si une erreur est survenue.

Pour effectuer l'opération inverse, on utilise une fonction similaire :

```
int fread(void *adresse, int taille, int nombre, FILE *fichier);
```

Vous devez vous assurer que `adresse` pointe sur une zone de mémoire correctement allouée et suffisamment grande pour contenir `nombre` éléments.

Exemple :

```
        struct machin tab[10];
        FILE *f;
        ...
        f = fopen("test.bin", "w");
        fwrite(tab, sizeof(struct machin), 10, f);
        fclose(f);
```

1.3.2 Accès en mode texte

La lecture et l'écriture de fichiers en mode texte fonctionne comme pour la console :

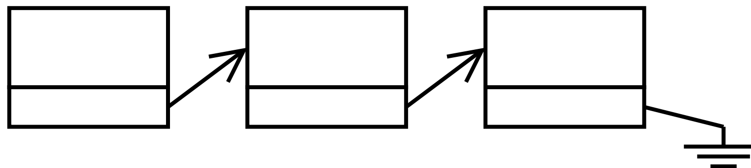
```
int fprintf(FILE *fichier, char *format, ...);
int fscanf(FILE *fichier, char *format, ...);
```

En fait, `printf` et `scanf` peuvent être considérées comme des cas particuliers de `fprintf` et `fscanf` :
- la commande "`printf("Coucou!\n");`" est équivalente à "`fprintf(stdout, "Coucou!\n");`"
- et la commande "`scanf("%i", &var);`" est équivalente à "`fscanf(stdin, "%i", &var);`".

2 Les listes chaînées

Nous l'avons vu, le C permet de définir des tableaux de taille fixe déclarés statiquement (tableaux normaux) ou dynamiquement (pointeurs + `malloc` et `free`). Lorsque le nombre d'éléments que l'on veut stocker est inconnu ou variable, il est alors nécessaire d'utiliser des types de données plus complexes.

Nous allons stocker nos données dans une structure chaînée, c'est à dire que chaque élément contiendra un lien vers le suivant. Le dernier élément contiendra un lien vers un élément vide.



```
typedef struct liste Liste;

struct liste
{
        int donnees;
        Liste *suivant;
};
```

Pour créer une liste :

```
        Liste *l;
        l = (Liste*)malloc(sizeof(Liste));
        l->suivant = NULL;
        l->donnees = ...
```

Pour ajouter un élément au début de la liste `l` :

```
        Liste *tmpl;
        tmpl = (Liste*)malloc(sizeof(Liste));
        tmpl->suivant = l;
        tmpl->donnees = ...
        l = tmpl;
```

Pour ajouter un élément en fin de la liste `l` :

```
        Liste *tmpl, *i;
        tmpl = (Liste*)malloc(sizeof(Liste));
        tmpl->suivant = NULL;
        tmpl->donnees = ...
        for (i = l; i->suivant != NULL; i = i->suivant);
        i->suivant = tmpl;
```

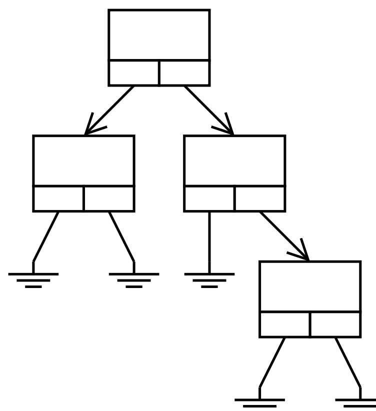
Toutes ces opérations peuvent être réalisées dans des fonctions afin de simplifier le code et pour éviter de recopier les mêmes lignes *n* fois, ce qui peut être source d'erreurs.

Un bon exercice est d'écrire un module (fichiers .c + .h) de gestion des listes chaînées avec toutes les fonctions utiles (créer une liste, effacer la totalité d'une liste, ajouter ou ôter un élément en début ou en fin de liste, etc...).

Parfois il peut s'avérer nécessaire de chaîner doublement une liste. Il suffit alors d'ajouter un pointeur vers l'élément précédent dans la structure. Le concept n'est pas plus compliqué mais il faut prendre quelques précautions supplémentaires lors de l'ajout ou de la suppression d'éléments.

3 Les arbres

Selon un principe identique, on peut définir plusieurs éléments suivants pour un élément. On obtient alors unes structure d'arbre. Un arbre dont les éléments ont deux successeurs (deux branches) est appelé arbre binaire.



```
typedef struct arbre2 Arbre2;

struct arbre2
{
        int donnees;
        Arbre2 *suivant[2];
};
```

Les arbres sont utilisés pour représenter des structures hiérarchiques, pour stocker de grandes quantités de données (en effet, la structure bi-dimentionnelle de l'arbre réduit la distance entre les noeuds) ou, plus simplement, pour effectuer des tris.