CCI - Principes des langages de programmation
TD 7 : Gestion de fichiers, Compilation séparée et Makefile

François Yvon[*], Thomas Tang[†]

24 octobre 2007

Première partie
Pour bien assimiler ...

Nous allons maintenant voir comment gérer un fichier texte dans un programme C. La manipulation de fichiers binaires (images, sons ...) requiert d'autres notions et sont au-delà de notre propos. Avant de commencer, il est important de noter qu'il existe deux façons de manipuler des fichiers en C :

- la première utilise ce qu'on appelle des descripteurs de fichier ou " file descriptors " qui sont des entiers correspondant aux fichiers ouverts par le programme. On accède au fichier par lecture ou écriture de blocs (groupe d'octets de taille définie par le programmeur). La difficulté notable vient du fait que c'est au programmeur de bien préparer et gérer ses blocs pour être sûr d'avoir bien géré tout le fichier.
- la seconde méthode utilise ce qu'on appelle des flux avec tampons ou " stream ". Ces flux ont le type FILE * et toutes les opérations se font via des tampons qui doivent être vider. Il existe 3 flux déjà ouverts dans tout programme :
  - stdin (standard input) : unité d'entrée (par défaut, le clavier) ;
  - stdout (standard output) : unité de sortie (par défaut, l'écran) ;
  - stderr (standard error) : unité d'affichage des messages d'erreur (par défaut, l'écran).

Nous ne nous intéresserons qu'à la seconde méthode dans cette partie d'assimilation, la partie approfondissement traitera la première méthode.

1 Gérer un fichier

1.1 Ouvrir et fermer un fichier

Dans la suite, on suppose que vous avez créé un fichier nommé " exemple.txt " dans votre éditeur de texte favori. Ce fichier contiendra deux lignes de texte simple, sans accent et se terminera par une nouvelle ligne (3 lignes en tout). La fonction *fopen* ouvre le fichier dont le chemin et le mode lui sont indiqués. Elle retourne un pointeur qui nous sert à accéder au fichier. En cas d'erreur, le pointeur *NULL* est retourné.

```
FILE * fopen(char * chemin, char * mode)
```

Le mode est à choisir parmi ceux du tableau suivant :

"r"	ouverture d'un fichier texte en lecture
"w"	ouverture d'un fichier texte en écriture
"a"	ouverture d'un fichier texte en écriture à la fin du fichier
"r+"	ouverture d'un fichier texte en lecture/écriture
"w+"	ouverture d'un fichier texte en lecture/écriture
"a+"	ouverture d'un fichier texte en lecture/écriture à la fin du fichier

Ces modes d'accès ont pour particularités :

[*] yvon@limsi.fr
[†] tang@cgm.cnrs-gif.fr

– Si le mode contient la lettre r, le fichier doit exister.
– Si le mode contient la lettre w, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, son ancien contenu sera perdu.
– Si le mode contient la lettre a, le fichier peut ne pas exister. Dans ce cas, il sera créé. Si le fichier existe déjà, les nouvelles données seront ajoutées à la fin du fichier précédent.

Un fichier ouvert dans un programme doit être fermé lorsqu'il n'est plus utilisé. Pour ce faire il existe une fonction pour fermer ce fichier. Elle retourne 0 si tout s'est bien passé.

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

Exercice 1 *Ecrire une fonction qui ouvre et ferme votre fichier exemple.txt.*

1.2 Lire le contenu d'un fichier

On peut lire le contenu d'un fichier ligne par ligne ou caractère par caractère.

```
char * fgets(char * str, int taille, FILE * flux)
int fgetc(FILE *flux)
```

La fonction fgets lit une chaîne en s'arrêtant au caractère "fin de ligne" ou à taille-1 caractères lus. Le résultat se trouve dans la zone pointée par str. Le pointeur str est retourné ou NULL s'il y a eu erreur de lecture du fichier. La fonction fgetc lit le premier caractère disponible dans le flux *flux*. A la fin du fichier, fgetc renvoie le caractère spécial EOF (" End Of File ").

Exercice 2 *Lire un fichier*

– *Ecrire un programme qui lit et affiche votre fichier ligne par ligne à l'aide de fgets.*
– *Ecrire un programme qui lit et affiche votre fichier caractère par caractère à l'aide de fgetc.*

1.3 Ecrire dans un fichier

De même, on peut écrire dans un fichier ligne par ligne ou caractère par caractère.

```
char fputs(char *s, FILE *flux)
int fputc(char c, FILE *flux)
```

La fonction fputs écrit la chaîne dans le fichier sans ajouter de caractère de "fin de ligne", et retourne le dernier caractère écrit ou EOF en cas d'erreur. La fonction fputc écrit *c* dans le flux de données et retourne l'entier correspondant au caractère lu ou EOF en cas d'erreur.

Exercice 3 *Ecrire un programme qui écrit plusieurs lignes dans un fichier (en écrasant son contenu) à l'aide de fputs, puis de fputc. Dans un deuxième temps, on ajoutera ces lignes à la fin du fichier.*

1.4 Se positionner dans un fichier

Les différentes fonctions d'entrées-sorties permettent d'accéder à un fichier en mode séquentiel : les données du fichier sont lues ou écrites les unes à la suite des autres. Il est également possible d'accéder à un fichier en mode direct, c'est-à-dire que l'on peut se positionner à n'importe quel endroit du fichier. La fonction fseek permet de se positionner à un endroit précis.

```
int fseek(FILE *flux, long deplacement, int origine)
```

La variable *deplacement* détermine la nouvelle position dans le fichier. Il s'agit d'un déplacement relatif par rapport à l'origine ; il est compté en nombre d'octets. La variable *origine* peut prendre trois valeurs :
– SEEK_SET (égale à 0) : début du fichier
– SEEK_CUR (égale à 1) : position courante
– SEEK_END (égale à 2) : fin du fichier

La fonction ftell retourne la position courante dans le fichier (en nombre d'octets depuis l'origine).

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

Exercice 4 *A l'aide des fonctions précédentes, se placer au début du fichier, puis à la fin du fichier. Ne pas oublier de vérifier.*

2 Compilation séparée

Une vraie application complète nécessite souvent un peu d'organisation lorsque la taille du code commence à grandir. Pour simplifier cette organisation et mettre un peu d'ordre dans le développement, on utilise souvent la notion de module séparé, aussi appelé unité de compilation.

Architecture en module

On va maintenant voir comment découper un programme en module. Un module est composé de deux parties, représentées en C par deux fichiers en général de même nom avec les extensions " .h " (interface) et " .c " (implémentation). L'interface décrit les fonctionnalités du module (types, structures, constantes, fonctions ...) tandis que l'implémentation fournit le code correspondant. Supposons que vous vouliez utiliser des listes chaînées d'entiers dans votre programme vous pouvez séparer la définition de ces listes chaînées dans un module que nous appellerons listes. Voici un exemple d'interface et d'implémentation pour ce module :

```
/* liste.h */

#include <stdio.h>

struct _liste {
    int elem ;
    struct _liste * suivant ;
} ; typedef struct _liste * liste ;

int tete(liste) ;
liste reste(liste) ;
liste ajoute(liste,int) ;
liste liste_vide() ;
int est_vide(liste) ;

/* liste.c */

#include <stdio.h>
#include "liste.h"

int tete(liste l) {
    return (l -> elem) ;
}

liste reste(liste l) {
    return (l -> suivant) ;
}

liste ajoute(liste l, int e){
    liste tmp ;
    tmp = malloc(sizeof(struct _liste)) ;
    tmp -> elem = e ;
    tmp -> suivant = l ;
    return tmp ;
}
```

```
liste liste_vide() {
    return NULL ;
}

int est_vide(liste l) {
    return (l == NULL) ;
}
```

On dispose maintenant d'un module liste qui contient la définition des listes ainsi que quelques fonctions pour leur manipulation. On notera que le fait de déclarer des fonctions de manipulation qui permettent d'effectuer toutes les opérations sur les listes permet après d'utiliser ces listes sans en connaître les détails d'implémentation. Seule l'interface et notamment les déclarations de fonction sont nécessaire à l'utilisation du module. Voici un exemple d'utilisation :

```
/* test_liste.c */ /* utilisation liste */

#include <stdio.h>
#include "listes.h"

void affiche_liste(liste l){
   if (est_vide(l)) {
        printf("FIN\n") ;
    } else {
printf("%d ", tete(l)) ;
affiche_liste(reste(l)) ;
    }
}

int main() {
    liste l ;
     l = liste_vide() ;
     l = ajoute(l,1) ;
    l = ajoute(l,2) ;
    l = ajoute(l,3) ;
     affiche_liste(l) ;
     exit(0) ;
}
```

Maintenant, nous désirons compiler ces différents fichiers. La première chose à faire consiste à compiler notre module en objet (extension " .o ") : gcc -c liste.c.

3 Compilation automatisée : Makefile

Lorsque le nombre de modules augmente la compilation de chacun devient fastidieuse, les lignes sont longues à taper et souvent sources d'erreur. De plus, il n'est pas forcément nécessaire de tout recompiler à chaque fois. Pour automatiser tout cela, il existe l'utilitaire make et son fichier makefile. make permet d'effectuer un certain de tâches qui engendrent des fichiers. Qui plus est, à l'aide de règles précises, on ne régénère que les fichiers nécessaires.

Syntaxe

```
cible: fichier1 fichier2 ...
    [commande après tabulation]
```

La commande (à taper sous la console !) *make* engendre le fichier correspondant à la cible si les fichiers sources fichier1, fichier2 ... sont plus récents que la cible. On peut utiliser plusieurs commandes en les mettant sur

différentes lignes.
ATTENTION : les lignes contenant des commandes commencent obligatoirement par une tabulation et non des espaces.

Voici un exemple de fichier makefile pour générer le programme test_liste que vous sauverez dans le même répertoire que les fichiers liste.c, liste.h et test_liste.c.

```
1 # Makefile
2 test_liste: test_liste.c liste.o
3    gcc -o test_listes listes.o test_listes.c
4
```

Remarquez que la dernière ligne du fichier est vide.

Commencez par effacer les résultats de la compilation précédente (rm liste.o test_liste) et tapez la commande make. Vous devriez obtenir à nouveau un fichier test_liste après avoir vu les différentes commandes exécutées. Si vous tapez make de nouveau, rien ne se passe de plus et la console vous informe que test_liste est à jour. Vous remarquerez que l'on n'a pas défini de règles pour compiler liste.o et qu'il a su le faire tout seul. En effet, make sait traiter un certains nombres de types de fichier tout seul. On va maintenant rajouter une règle qui nous évitera d'effacer nous même les fichiers produits par la compilation à chaque fois et éviter ainsi les problèmes de compilation.

```
4 clean:
5   rm -f *.o test_listes
6
```

Cette règle ne produit pas de fichier, elle sera donc lancée à chaque fois qu'elle est appelée. Heureusement, lorsque vous tapez make, clean n'est pas appelée systématiquement, pour l'appeler il faut l'indiquer à make comme ceci : make clean. Si aucun but n'est précisé, make exécute la première règle, sauf si on a redéfini la cible spéciale all.

Deuxième partie

Pour approfondir ...

Fonctions utilisant la méthode de gestion des fichiers avec descripteurs :

```
int open(char * chemin, int flags) La fonction open ouvre le fichier
dont le nom et le chemin sont indiqués par chemin avec les flags
(options) suivantes :
- O_RDONLY : Ouverture en lecture seule
- O_WRONLY : Ouverture en écriture seule
- O_RDWR : Ouverture en lecture/écriture
- O_APPEND : Ecriture à la fin
- O_CREAT : Crée le fichier s'il n'existe pas
- O_TRUNC : Tronque le fichier à la taille 0 (écrasement)
- O_EXCL : Provoque une erreur si le fichier existe lors d'une création.

Pour combiner plusieurs options, on utilise l'opérateur " ou " bit à
bit  " | ". Par exemple,  pour ouvrir un fichier en écriture seule
en le créant si besoin on utilisera comme flag : O_WRONLY | O_CREAT.

int close(int fd)
Ferme le fichier ouvert avec open.

int read(int fd, char * buffer, unsigned int nb_octets) Lit
nb_octets dans le fichier désigné par fd et ouvert avec open.
```

```
Renvoie le nombre d'octets effectivement lus. Le résultat est écrit
dans buffer qui doit être initialisé avec une taille suffisante
(attention il faut rajouter le caractère " \0 " à la main.)
```

Exercice 5 *Ecrire un programme qui lit votre fichier (exemple.txt) à l'aide de la fonction read. Vous devrez utiliser une boucle qui lit dans le fichier tant que read ne renvoie pas 0 octets lus (attention à fournir à read un pointeur décalé par rapport au nombre d'octets déjà lus et à rajouter le caractère.) Vous afficherez le résultat de la lecture.*