

TP 4 : Accès aux fichiers

Utilisation de gdb

1 Rappel des principales notions liées à l'accès au système de fichiers

Diverses fonctions de la librairie C permettent d'accéder aux fichiers. Ces fonctions d'accès aux fichiers ont besoin de diverses informations :

- le mode d'accès (lecture seule, écriture seule...);
- la position courante de la "tête de lecture" virtuelle;
- l'adresse d'une mémoire-tampon permettant de ne pas accéder systématiquement au matériel, ce qui serait lent;
- ...

Ces informations sont rassemblées dans une structure dont le type, `FILE`, est défini dans `stdio.h`. Un objet de type `FILE *` est appelé *flot de données* (en anglais, *stream*).

Avant de lire ou d'écrire dans un fichier, un programme doit demander l'initialisation d'une telle structure de données par la commande `fopen()`. Cette fonction prend comme arguments le nom du fichier et le mode d'accès, négocie avec le système d'exploitation et initialise un flot de données, qui sera ensuite utilisé lors de l'écriture ou de la lecture. Ces lectures/écritures seront réalisées via diverses fonctions que vous verrez dans la suite du TP. La taille minimale d'un accès par ces fonctions est l'octet (8 bits).

Après les traitements, on annule la liaison entre le fichier et le flot de données grâce à la fonction `fclose()`. Pour utiliser toutes les fonctions d'entrées-sorties, vous devez inclure `<stdio.h>`. Dans la suite du TP, pour avoir des détails sur le fonctionnement de ces fonctions, n'hésitez pas à utiliser la commande `man` (dans un terminal). Exemple : `man fopen`.

Les accès à l'entrée (resp. la sortie) standard peuvent se faire de la même manière qu'à un fichier, via le *stream* `stdin` (resp. `stdout`).

2 Exercice : copie d'un fichier caractère par caractère

Pour cet exercice, n'utilisez que les fonctions suivantes d'accès aux fichiers :

- `FILE *fopen(const char *pathname, const char *mode);`
- `int fclose(FILE *stream);`
- `int fgetc(FILE *stream);` qui permet de lire le prochain caractère dans un flot de données.
- `int fputc(int c, FILE *stream);` qui permet d'écrire le prochain caractère dans un flot de données.
- `int feof(FILE *stream)` qui retourne un entier différent de zéro si la fin de flot de données a été atteinte.

Vous verrez dans les pages de manuel des fonctions ci-dessus, des références au symbole `EOF`. C'est juste un entier défini dans la librairie `stdio`. Il est utilisé pour indiquer que la fin d'un fichier a été atteinte.

QUESTION 1 ► Écrivez un programme qui lit le contenu d'un fichier texte, caractère par caractère, et affiche à l'écran les caractères lus en utilisant la fonction `printf()`. Il existe plusieurs manières de repérer qu'on a atteint la fin d'un fichier. La fonction `feof()` renvoie un entier différent de 0 quand le caractère `EOF` (End Of File) a été lu, mais il ne faut pas afficher ce caractère. On peut également tester l'égalité d'un caractère lu par `fgetc()` avec cette constante `EOF` : quand c'est le cas c'est que la fin de fichier a été atteinte, et là encore il ne faut pas l'afficher.

QUESTION 2 ► Pour écrire (resp. lire) sur la sortie standard (resp. entrée standard), vous pouvez utiliser le symbole `stdout` (resp. `stdin`), de type `FILE*`. Modifier le programme de la Question 1 pour que l’affichage à l’écran ne se fasse plus avec la fonction `printf()`, mais avec la fonction `fputc()` pour écrire sur le fichier de sortie standard.

QUESTION 3 ► Modifiez maintenant votre programme pour qu’il n’effectue pas seulement l’affichage du fichier sur la sortie standard mais en copie le contenu dans un nouveau fichier dont le nom est indiqué dans une variable.

QUESTION 4 ► Modifiez le programme que vous venez d’écrire pour que le nom des fichiers ne soient plus indiqués par des variables mais passé en argument de la ligne de commande (et récupéré via la variable `argv` de votre fonction `main()`).

3 Exercice : positionnement 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 ; elle a pour prototype :

`int fseek(FILE *stream, long offset, int whence);`
La variable `offset` détermine la nouvelle position dans le fichier. Il s’agit d’un déplacement, en nombre d’octets, relatif à `whence` qui peut prendre trois valeurs :

- `SEEK_SET` : début du fichier ;
- `SEEK_CUR` : position courante ;
- `SEEK_END` : fin du fichier.

La fonction `int rewind(FILE *stream);` permet de se positionner au début du fichier. Elle est équivalente à `fseek(flot, 0, SEEK SET);`.

La fonction `long ftell(FILE *flot);` retourne la position courante dans le fichier (en nombre d’octets depuis l’origine).

QUESTION 5 ► Écrivez une fonction qui remplit un flot avec 20 lignes constituées chacune de 40 caractères ‘_’ et qui repositionne l’indicateur de positionnement au début du flot.

QUESTION 6 ► Écrivez un programme qui, après avoir créé le fichier évoqué à la question précédente, répète en boucle en boucle tant que l’utilisateur n’a pas appuyé sur la lettre ‘q’ :

1. affiche le contenu du fichier en entier ;
2. attend un caractère, appelons-le *k*, sur l’entrée standard ;
3. attend un nombre *offset* sur l’entrée standard ;
4. remplace, dans votre fichier, le caractère situé en position *offset* par le caractère *k*. Implémentez cette sous-partie dans une fonction en dehors de `main()`.

Pour convertir une chaîne de caractère représentant un nombre (par exemple “42”) en un type entier, voyez la fonction `long atol(const char *nptr);`.

Attention : le comportement de la fonction `scanf` est facétieux, quand vous enchaînez les `scanf`, les entrées sont enregistrées dans un buffer d’entrée-sortie. Concrètement cela signifie que si vous faites deux `scanf("%d", &x)` à la suite et qu’au clavier vous entrez par exemple `12\n23\n` (c’est à dire que vous taper “12” suivi de “entrée”, puis “23” suivi de “entrée”) le second `scanf` va prendre “\n” comme entrée suivant “12”. Pour éviter cela vous pouvez soit écrire “ `%d", &x)` avec un espace devant le format) ou bien rajouter un `getc()` “inutile” après le premier `scanf` qui va consommer le “\n”.

QUESTION 7 ► Si vous n’y avez pas pris garde, vous pouvez remplacer des retour à la ligne, ce qui casse le schéma. Prenez cela en compte en décalant l’indicateur de position dans ce cas là. N’oubliez pas de fermer le flot et de gérer les erreurs (nombre trop grand rentré par l’utilisateur). Testez votre programme en dessinant un carré avec ‘X’ (c’est-à-dire en remplaçant tous les caractères du bord par des ‘X’).

QUESTION 8 ► Faisons une petite référence au TP précédent. Votre programme précédent, si vous l’avez implémenté naïvement comme je l’attendais, accède au système de fichier, donc au disque dur, à chaque

demande de l'utilisateur. Modifiez-le pour que le contenu du fichier soit, au début du programme, copié dans une zone mémoire allouée dynamiquement (en supposant que vous ne connaissez pas la taille du fichier). Puis, chaque modification de caractères est faite dans ce *buffer* et ce n'est que lorsque l'utilisateur demandera à quitter le programme que le contenu du programme sera sauvegardé dans un nouveau fichier.

4 Utilisation du debugger gdb

Les TP ne permettent pas de se rendre compte de l'ampleur d'un véritable travail de programmation. Normalement les programmes sont beaucoup plus longs et complexes que de simples questions dont la réponse fait quelques lignes. L'utilisation d'un debugger permet d'évaluer un programme pas à pas et d'observer la valeur des variables pour comprendre comment il fonctionne concrètement et repérer les endroits où les erreurs se produisent. Il est possible de le faire "à la main" en ajoutant des `printf` mais cette méthode artisanale demande de recompiler le programme pour chaque modification et n'est pas très interactive. Les debuggers permettent de rendre ce processus plus professionnel. Le but du reste du TP est de vous montrer comment fonctionne l'outil `gdb`.

4.1 Présentation rapide de l'outil

Pour présenter l'outil `gdb`, nous allons nous servir d'un exemple de programme erroné. Le fichier `factorielle.c` est censé implanter la fonction factorielle.

QUESTION 9 ► Compilez et testez ce programme sur diverses valeurs. Que constatez vous ?

Nous allons maintenant présenter les principales caractéristiques de l'outil `gdb` pour déboguer ce programme. Suivez les instructions ci-dessous :

1. Compilez le programme avec l'option `-g`. Cela permet au compilateur de récupérer des informations utiles pour le débogage (où sont enregistrées les variables etc.) :

```
\$ gcc -g factorielle.c i -o factorielle
```

2. Lancer l'outil `gdb` sur le programme considéré :

```
\$ gdb factorielle
```

3. Définir un "break point" dans le programme considéré. La syntaxe est `break line_number`

```
break 10
```

4. Il faut maintenant exécuter le programme dans `gdb`. La syntaxe est `run [args]`

```
run
```

Le programme `factorielle` n'utilise pas d'arguments sur la ligne de commande mais il est possible de les donner à `i gdb` à la suite de la commande `run`. L'effet de cette commande est de lancer le programme : il s'exécute jusqu'à arriver au premier `breakpoint`.

5. Imprimer la valeur d'une variable dans `gdb`. La syntaxe est `print [variable]`.

```
print i
print j
print num
```

`gdb` va alors imprimer la valeur actuelle de la variable considérée.

QUESTION 10 ► Quel est le problème du programme fourni ? Comment peut on le régler ? Retestez le nouveau programme.

Pour trouver l'erreur restante on peut relancer le débogage comme précédemment jusqu'à l'étape 6. On peut alors continuer le programme en avançant un peu plus dans l'exécution du programme. Il y a trois commandes différentes sous `gdb` pour ce faire :

1. `continue` : avance l'exécution jusqu'au prochain `break point`.
2. `next` : exécute la ligne suivante du programme.

3. `step` : même sémantique que `next` mais rentre dans le code des fonctions pour exécuter la fonction ligne par ligne. Typiquement si la prochaine ligne du programme à exécuter est `x=f(z)` ; `next` fera l'affectation de `x` avec la bonne valeur ; par contre `step` va "rentrer" dans le code de `f` et n'exécutera que la première ligne de ce code.

A noter que ces commandes ont des versions raccourcies : `c` pour `continue`, `n` pour `next` et `s` pour `step`. Pour une documentation plus complète sur l'outil `gdb` vous pouvez consulter les manuels suivants :

- Le manuel en ligne : tapez `\$ man gdb` dans un terminal.
- La page officielle de la documentation `gdb` : <https://sourceware.org/gdb/documentation/>
- Page résumant les commandes `gdb` : <https://gist.github.com/rkubik/b96c23bd8ed58333de37f2b8cd052c30>

Enfin si votre programme est réparti sur plusieurs fichiers il faut utiliser la syntaxe suivante pour mettre des `break` points dans des fichiers autres que celui qui contient la fonction `main` : `break nom_du_fichier.c:x` qui a pour effet de mettre un `break` point à la ligne `x` du fichier `nom_du_fichier.c`.

QUESTION 11 ► Reprenez votre programme manipulant les listes chaînées du TP3 (à partir de la section 3 Définition récursive de types). Utilisez `gdb` pour :

1. Créer une liste chaînée de 5 éléments.
2. D'utiliser les commandes `gdb` pour parcourir cette liste "à la main" en utilisant les `print` qui vont bien. Le but est de parvenir à faire afficher la valeur des 5 cellules par des `print` en récupérant les adresses des cellules composants la liste.

QUESTION 12 ► Le code suivant est censé implanter un algorithme de tri. Il est bien entendu incorrect. Intégrez ce code dans votre programme et corrigez le en utilisant l'outil `gdb` pour voir où ça coince dans un premier temps et ensuite comment corriger cela.

```
struct Cell{
    int data;
    struct Cell *next;};

typedef struct Cell Cell;

Cell *quicksort(Cell *head)
{
    Cell *inf_p=NULL,*sup_p=NULL,*parcour=NULL;
    Cell *pptit, *pgrand;
    int pivot;

    if ( head = NULL) return NULL; // liste de taille 0

    pivot = head->data;
    parcour = head ;
    while (parcour){
        parcour = parcour ->next;
        if (parcour -> data < pivot)
            ajg(&inf_p,parcour -> data);
        else
            ajg(&sup_p,parcour -> data);
    }

    pptit = quicksort(inf_p);
    pgrand = quicksort(sup_p);
    ajg(&pgrand,pivot);
    concat(&pptit,pgrand); /*concat(&a,b) rajoute à a tous les éléments de b */

    return(pptit);          /*en faisant pointer le next de la dernière cellule de a sur b */
}
```