TP 4 : Utilisation de gdb
Accès aux fichiers

1 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 beaucoups 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 le endroits où les erreurs se produisent. Il est possible de le faire "à la main" en ajoutant des `printf` mais cette méthode artisinale demande de recompiler le programme pour chaque modification et n'est pas très interactive. Les debuggers permettent de rendre ce processus plus professionel. Le but du reste du TP est de vous montrer comment fonctionne l'outil `gdb`.

1.1 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 1 ► Compilez et testez ce programme sur diverses valeurs. Que constatez vous ?
Nous allons maintenant présenter les principales caractéristiques de l'outil `gdb` pour débugger 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 débugage (où sont enregistrées les variables etc.) :

```
\$ gcc -g factorielle.c  -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 éxécuter 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 `gbd` à la suite de la commande run. L'effet de cette commande est de lancer le programme : il s'éxécute 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 2 ► 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 debuggage comme précédement jusqu'à l'étape 6. On peut alors continuer le programme en avançant un peu plus dans l'éxécution 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 racourcies : `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 : tappez `$ 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`.
Il faut compiler les différents fichiers avec les options `-c -g` puis faire l'édition de liens comme d'habitude.

QUESTION 3 ► 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 4 ► 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.

```
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 = parcours ->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 */
}
```

2 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`).
Il est à noter que la philosophie UNIX est que "tout est fichier". Ainsi que vous fassiez de la programmation réseaux (écriture dans des sockets), produisiez du son, écriviez ou recevez des données à partir de la sortie standard etc. tout peut être vu comme l'écriture ou la lecture dans un fichier qui représente la suite des informations considérées. Cela permet d'homogénéiser de nombreux traitements en offrant un point de vue unique pour le programmeur.

3 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 retourn 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 5 ► 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 manière de repérer qu'on a atteint la fin d'un fichier. La fonction `feof()` renvoie un entier different 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 6 ► 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 7 ► 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 8 ► 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()`).

4 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 9 ► 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 10 ► 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 entréz par exemple `12\n23\n` (c'est à dire que vous tapper "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 11 ► 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 12 ► 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.

5 Exercice : générateur de langue de bois

Vous allez implémenter un générateur de langue de bois. Pour accéder au système de fichier, utilisez les fonctions suivantes :

- `FILE *fopen(const char *pathname, const char *mode);`
- `int fclose(FILE *stream);`
- `int feof(FILE *stream);`
- `char *fgets(char *s, int size, FILE *stream);`
- `int fputs(const char *s, FILE *stream);`

QUESTION 13 ► Lisez le fonctionnement de la fonction `char *fgets(char *s, int size, FILE *stream);` grâce à `man`. Quels sont les 3 cas qui conduisent cette fonction à arrêter de lire dans le fichier et à vous retourner un résultat ?

QUESTION 14 ► Votre générateur de langue de bois va implémenter le tableau 1. Sur Moodle vous trouverez le texte de chaque colonne dans des fichiers séparés (par exemple, le texte de la colonne 1 dans un fichier `colonne1.txt`), à raison de 1 ligne par case. Écrivez maintenant un programme qui parcourt tour à tour ces 4 fichiers et, pour chacun, copie une ligne du fichier prise au hasard sur la sortie standard. Pour prendre une phrase au hasard, utilisez la fonction random() ; voyez ci-dessous pour un petit exemple de tirage de 10 chiffres parmi O, 1, 2, 3.

```
#include <stdlib.h>
#include <time.h>
...
srandom(time(NULL)); // initialisation de la graine d'aléa
for (i=0 ; i < 10 ; i++) {
  long tirage = random() % 4;
  printf("Chiffre au hasard : %ld\n", tirage);
}
```

QUESTION 15 ► Adaptez maintenant votre programme pour qu'il génère un petit discours de 5 phrases dans un fichier appelé `discours.txt`. N'oubliez pas d'appeler `fclose()` quand vous n'avez plus besoin d'accéder à un fichier.

6 Exercice : contrôle d'accès aux fichiers

Vous devez vous rappeler que, sous Unix, à chaque fichier sont associées des permissions d'accès. Il est possible de connaître ces permissions grâce à la fonction `int fd, stat(const char *pathname, struct stat *statbuf);` qui remplit la structure `statbuf` avec plein d'informations associées au fichier dont le descripteur de fichier est donné en paramètre. Ce qui nous donne l'occasion de clarifier un point prêtant parfois à confusion :

- pour l'instant, nous avons manipulé les fichiers via des *flot*, de type `FILE *` pointant vers
- un numéro unique est attribué par l'OS à tout fichier ouvert, appelé, descripteur de fichier. Certains fonctions de la librairie C demandent en paraètre cet identifiant plutôt qu'un flot. Pour connaître ce descripteur de fichier, utilisez la fonction `fileno()`.

Les détails de la structure `struct stat` sont donnés dans la page de manuel de `fstat()` (`man stat` en ligne de commande pour la consulter).

QUESTION 16 ► Écrivez un programme qui affiche le plus clairement possible (avec des phrases) si le propriétaire du fichier peut lire un fichier donné, s'il peut écrire dedans, s'il a les droits d'exécution. Cela vous demande d'interprêter correctement le champ `st_mode` de la structure `fstat`

1	2	3	4
Mesdames, messieurs,	la conjoncture actuelle	doit s'intégrer à la finalisation globale	d'un processus allant vers plus d'égalité.
Je reste fondamentalement persuadé que	la situation d'exclusion que certains d'entre vous connaissent	oblige à la prise en compte encore plus effective	d'un avenir s'orientant vers plus de progrès et plus de justice.
Dès lors, sachez que je me battrai pour faire admettre que	l'acuité des problèmes de la vie quotidienne	interpelle le citoyen que je suis et nous oblige tous à aller de l'avant dans la voie	d'une restructuration dans laquelle chacun pourra enfin retrouver sa dignité.
Par ailleurs, c'est en toute connaissance de cause que je peux affirmer aujourd'hui que	la volonté farouche de sortir notre pays de la crise	a pour conséquence obligatoire l'urgente nécessité	d'une valorisation sans concession de nos caractères spécifiques.
Je tiens à vous dire ici ma détermination sans faille pour clamer haut et fort que	l'effort prioritaire en faveur du statut précaire des exclus	conforte mon désir incontestable d'aller dans le sens	d'un plan correspondant véritablement aux exigences légitimes de chacun.
J'ai depuis longtemps (ai-je besoin de vous le rappeler ?), défendu l'idée que	le particularisme dû à notre histoire unique	doit nous amener au choix réellement impératif	de solutions rapides correspondant aux grands axes sociaux prioritaires.
Et c'est en toute conscience que je déclare avec conviction que	l'aspiration plus que légitime de chacun au progrès social	doit prendre en compte les préoccupations de la population de base dans l'élaboration	d'un programme plus humain, plus fraternel et plus juste.
Et ce n'est certainement pas vous, mes chers compatriotes, qui me contredirez si je vous dis que	la nécessité de répondre à votre inquiétude journalière, que vous soyez jeunes ou âgés,	entraîne une mission somme toute des plus exaltantes pour moi :	l'élaboration d'un projet porteur de véritables espoirs, notamment pour les plus démunis

Tableau 1 - Commencez avec n'importe quelle case de la colonne 1 puis n'importe laquelle en colonne 2, puis colonne 3 puis 4.

7 Exercice de révisions de un peu de tout : une bibliothèque d'accès bit-à-bit

La bibliothèque standard n'offre pas de moyen de lire ou d'écrire un seul bit à la fois dans un fichier. Le but de cet exercice est d'écrire une bibliothèque permettant d'effectuer des entrées/sorties bit-à-bit dans un fichier. Une telle bibliothèque pourrait se révéler utile pour implémenter des algorithmes de compression par exemple. Le problème qui se pose lors de la réalisation de fonctions d'accès bit-à-bit aux fichiers est que nous devons les réaliser en nous servant des fonctions standard pour lesquelles la taille minimale d'un accès est l'octet (8 bits).
Une réalisation possible pour les fonctions d'écriture de cette bibliothèque consisterait à maintenir un ensemble de bits en attente d'écriture (dans une variable, qui nous servirait ainsi de mémoire tampon) et, dès que ces bits sont au nombre de huit, de réaliser l'écriture de l'octet correspondant à l'aide d'une des fonctions de la bibliothèque standard. Pour les fonctions de lecture, nous pouvons utiliser la même idée en maintenant un octet lu à l'avance dans le fichier duquel nous extrayons les bits les uns après les autres au fur et à mesure des demandes.
Vous aurez besoin des fonctions suivantes pour lire/écrire dans/depuis le fichier :

- `size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);`
- `size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);`

QUESTION 17 ► Vous allez implémenter les fonctions de cette librairie dans un fichier `bblib.c`. Mettez à jour votre Makefile de manière à ce qu'il compile de manière séparée les fichiers `main.c` et `bblib.c`.

QUESTION 18 ► Choisissez une structure de données permettant de maintenir toutes les informations qui vous seront utiles pour accéder à un fichier un bit à la fois. Définissez un type associé que nous appellerons BFILE dans un fichier `bblib.h` (et incluez ce fichier dans `bblib.c`.

QUESTION 19 ► Réalisez l'implémentation correspondant à la spécification suivante :

```
/*
   bstart
   description : démarre un accès bit à bit au fichier dont le descripteur est
                 passé en paramètre (le fichier doit avoir été préalablement
                 ouvert).
                 Pour conserver la cohérence des données, aucun accès non bit à
                 bit au même descripteur ne doit être fait jusqu'au prochain
                 bstop.
   paramètres : descripteur du fichier, le mode dans lequel le descripteur est
                ouvert.
   valeur de retour : pointeur vers une structure BFILE allouée par bstart
                      ou NULL si une erreur est survenue
   effets de bord : aucun (outre l'allocation)
*/
BFILE *bstart(FILE *fichier);

/*
   bstop
   description : termine un accès bit à bit à un fichier préalablement démarré
                 par bstart (termine les E/S en attente et libère la structure
                 BFILE).
   paramètres : pointeur vers une structure BFILE renvoyée par bstart
   valeur de retour : 0 si aucune erreur n'est survenue
   effets de bord : écrit potentiellement dans le fichier
*/
int bstop(BFILE *fichier);

/*
   bitread
   description : lit un bit dans un fichier sur lequel un accès bit à bit
                 a été préalablement démarré a l'aide de bstart.
   paramètres : pointeur vers une structure BFILE renvoyée par bstart
   valeur de retour : bit lu ou -1 si une erreur s'est produite ou si la
                      fin de fichier a été atteinte
   effets de bord : la valeur de retour dépend du contenu du fichier
                    lit potentiellement dans le fichier
*/
char bitread(BFILE *fichier);
```

```
/*
   bitwrite
   description : écrit un bit dans un fichier sur lequel un accès bit à bit
                 a été préalablement démarré à l'aide de bstart.
   paramètres : pointeur vers une structure BFILE renvoyée par bstart, bit à
                écrire
   valeur de retour : -1 si une erreur s'est produite
   effets de bord : écrit potentiellement dans le fichier
*/
int bitwrite(BFILE *fichier, char bit);

/*
   beof
   description : retourne 1 si un accès en lecture préalable a échoué pour
                 cause d'atteinte de la fin d'une séquence de bits (fin de
                 fichier ou fin de séquence codée dans le fichier), 0 sinon.
                 L'accès bit à bit doit avoir été préalablement démarré à
                 l'aide de bstart.
   paramètres : pointeur vers une structure BFILE renvoyée par bstart
   valeur de retour : 1 ou 0
   effets de bord : aucun.
*/
int beof(BFILE *fichier);
```