UNIVERSITÉ DE LORRAINE | IUT nancy Charlemagne

DUT Informatique
Aspects système, les signaux

1 Tables de multiplication

Faire un programme permettant de réviser les tables de multiplication. Le programme comportera une boucle infinie posant des questions à ce sujet. Les opérandes devront être choisies au hasard. Pour sortir du programme, l'utilisateur devra faire un `CTRL-C`, qui provoquera l'affichage du score obtenu. Le programme devra également s'arrêter si l'utilisateur met plus de 5 secondes à répondre à une question. L'usage des variables globales est recommandé pour cet exercice et on conseille d'utiliser `sigaction()` et de compiler le programme avec le support de la norme `ansi`[1].

On donne les informations suivantes :

- Pour la gestion du hasard, il est recommandé d'utiliser la fonction `rand()`, initialisée grâce à une instruction comme `srand(time(NULL))` (permettant de positionner la « graine » initiale du générateur de nombres aléatoires). Regarder la documentation disponible sur votre système (`man`) pour avoir des précisions sur l'utilisation de ces fonctions.
- Définir une fonction `sortie()`, qui sera appelée lorsque la sortie du programme est sollicitée. Elle affichera le score obtenu et mettra fin au programme.
- Pour cet exercice, il est préférable d'utiliser la fonction `sigaction()` plutôt que la fonction `signal()`.
- (*Utilisateurs avancés, pas nécessaire*) On rappelle que l'appel à une fonction lors du traitement d'un signal est réalisé de telle manière que le programme continue son exécution normalement au retour de cette fonction. Afin de pouvoir revenir à un endroit différent (typiquement si cet endroit correspondait à un `scanf()`), il est possible d'utiliser les fonctions :
  - `setjmp()` : permet de sauvegarder un endroit arbitraire du programme (pas forcément le début d'une fonction) et tout le contexte qui va avec,
  - `longjmp()` : permet de retourner à un endroit précédemment sauvegardé.

  Là encore, les pages de manuel peuvent vous donner des informations complémentaires à ce sujet.

Questions subsidiaires :

- modifier le programme pour qu'il attribue des points en fonction du temps de réponse de l'utilisateur (plus il est rapide, plus il marque de points). La valeur de retour de la fonction `alarm()` peut être utilisée à cette fin…
- quelle approche pourrait-on suivre si on souhaitait définir une alarme plus précise que celle fournie par la fonction `alarm()` ?

2 Clone de `ls`

On demande d'écrire un clone de la commande Unix `ls` affichant, pour chaque entrée, le nom du fichier et son numéro d'inode. Deux type de données sont particulièrement importants pour réaliser cette commande (qui dans sa forme simple tient en une dizaine de lignes…) :

- le type `DIR`, décrivant un répertoire,
- la structure `struct dirent`, décrivant une entrée de répertoire.

[1] Voir le cours, en particulier la section relative à `sigaction()`, pour plus de détails sur les options à mettre en œuvre pour réussir à compiler.

Le header `dirent.h` permet de bénéficier de la définition de ces types de données dans un programme C. Trois fonctions seront principalement utilisées :

— `opendir()` : permettant d'ouvrir un répertoire dont le nom est passé en paramètre,
— `readdir()` : permettant de récupérer la prochaine entrée (*i.e.* le prochain fichier) d'un répertoire ouvert,
— `closedir()` : permettant de fermer un répertoire précédemment ouvert avec `opendir()` (et ainsi de libérer toutes les ressources allouées lors de son utilisation).

Pour chacune de ces fonctions, faire un `man` pour connaître le prototype et les modalités d'utilisation. Le programme peut ensuite être perfectionné : introduire un tri ascendant en utilisant la fonction `qsort()`.

3 TP noté à rendre

On désire extraire toutes les villes dont le code postal est partagé entre plusieurs villes. On dispose pour cela d'un fichier texte contenant toutes les villes et les codes postaux associés (disponible sur l'ENT). L'objectif est de définir un programme C permettant de répondre à ce besoin.

Pour cela, on demande de suivre le schéma suivant :

1. Définir une structure de données permettant de représenter en mémoire les données contenues dans le fichier. Cette structure peut être une simple liste chaînée où chaque élément contient un code postal et la ville associée. Elle peut également être une structure plus complexe, permettant de gérer une liste de villes pour chaque code postal existant. Dans tous les cas, les noms de ville doivent être stockés de manière dynamique afin de minimiser l'espace mémoire utilisé pour stocker les données.
2. Définir une fonction permettant de lire le fichier et de le stocker dans la structure définie. On indique que la manipulation de fichiers texte en C se fait à partir des fonctions :
   — `fopen()` : pour ouvrir un fichier (typiquement, un appel du genre `fopen("codes.txt", "r")`),
   — `fscanf()` : pour faire des lectures formatées depuis un fichier (ou `fgets()` couplée avec `sscanf()`),
   — `fclose()` : pour fermer un fichier (nécessaire une fois que le fichier exploité n'est plus utilisé).
3. Afficher tous les noms de ville partageant un même code postal. À cette fin, pour chaque code postal concerné, on affichera, dans l'ordre, le code postal concerné et ensuite la liste des villes concernées (une ville par ligne).
4. Libérer toute la mémoire dynamique utilisée afin que le programme se termine sans présenter aucune fuite de mémoire lors d'un rapport effectué par la commande `valgrind`.

Le travail doit être déposé sur l'ENT. L'archive doit comprendre un module dédié à la structure de données utilisée, un module correspondant au programme pricipal, un Makefile, le tout dans une archive ZIP portant le nom du ou des participants (2 maximum).