Université Joseph Fourier
GRENOBLE
UFR IM²AG

INF123 - Partiel
Année 2014-2015

Durée : 1h30.
Tout document interdit à l'exception du mémo shell non annoté. Calculatrices interdites.
Pour chaque question, une partie des points peut tenir compte de la présentation.
Le barème est indicatif.

Toute réponse même incomplète sera valorisée à partir du moment où elle réalise au moins une partie de ce qui est demandé.
Les questions sont relativement indépendantes, et dans tous les cas vous pouvez utiliser les fonctions demandées dans les questions précédentes même si vous n'avez pas réussi à les écrire.

1 Programmer une commande `cut` simplifiée (14 points)

Le but de cette section est de programmer en **langage C** un exécutable réalisant la même tâche que la commande `cut` qui permet, entre autres, d'extraire une colonne d'un fichier ASCII. Par exemple, supposons qu'un fichier `cacahuetes.txt` ait le contenu suivant :

```
Buffalo:Bill:10
Super:Man:25
Miou:Miou:28
King:Kong:78
```

Ce fichier contient 3 colonnes séparées par le caractère `':'`. On veut alors récupérer le contenu de la deuxième colonne :

```
Bill
Man
Miou
Kong
```

Nous n'exposerons pas ici la syntaxe complète de `cut`, qui permet de faire beaucoup d'autres choses, mais nous nous contenterons d'une syntaxe simplifiée où notre exécutable prend en paramètres 3 arguments :

1. le *caractère* séparateur
2. le numéro de colonne (un *entier strictement positif*!)
3. le nom du fichier

Ainsi, si notre exécutable s'appelle `couper`, la ligne de commande
`./couper : 2 cacahuetes.txt`
donnera le résultat voulu dans notre exemple.

1.1 Lecture d'une ligne (2 points)

Pour commencer, nous aurons besoin d'une fonction `readline` avec le prototype suivant :

```
int readline(FILE* f,char ch[]);
```

Cette fonction lit une ligne dans le fichier pointé par le descripteur `f`. C'est-à-dire qu'elle lit et stocke dans la chaîne de caractères `ch` les caractères lus dans le fichier jusqu'à rencontrer un caractère de saut de ligne ou de fin de fichier.

Cette fonction renvoie 0 si elle a rencontré un saut de ligne, et 1 si la fin de fichier est atteinte.

Question 1 : Donnez le code de la fonction `readline`.

1.2 Conversion chaîne vers entier (2 points)

Nous utiliserons également une fonction `versEntier` avec le prototype suivant :

```
int versEntier(char ch[],int *nb);
```

Cette fonction teste si la chaîne `ch` représente un entier positif ou nul et le cas échéant, stocke cet entier dans la variable dont l'adresse est dans `nb`. Ainsi, `versEntier("12A",&x)` renvoie 0. Mais, `versEntier("124",&x)` renvoie 1 et, suite à cet appel, `x` contient 124.

Question 2 : Donnez le code de la fonction `versEntier`.
(Votre code ne doit pas *faire appel à une fonction de bibliothèque.)*

1.3 Extraire une colonne d'une ligne (4 points)

Enfin, pour le découpage des lignes, nous procédons en deux temps. Tout d'abord, considérons la fonction `trouve_colonne` qui a le prototype suivant :

```
int trouve_colonne(char ligne[],char d,int i);
```

Cette fonction renvoie l'indice de la `ligne` passée en paramètre (un tableau de caractères) où se trouve le premier caractère de la $\texttt{i}^{\text{ème}}$ colonne, en supposant que les colonnes sont séparées par le caractère `d`. En cas d'erreur la fonction renvoie -1.

Par exemple :
- `trouve_colonne("Buffalo:Bill:10",':',2)` renvoie la valeur 8 ;
- `trouve_colonne("Buffalo:Bill:10",':',4)` renvoie -1 ;
- `trouve_colonne("toto::titi",':',2)` renvoie 5.

Question 3 : Donnez le code de la fonction `trouve_colonne`.

Ensuite, la fonction `extraction_colonne` a le prototype suivant :

```
int extraction_colonne(char ligne[],char col[],char d,int i);
```

Cette fonction copie dans la chaîne de caractères `col` le contenu de la $\texttt{i}^{\text{ème}}$ colonne de la `ligne` passée en paramètre, en supposant que les colonnes sont séparées par le caractère `d`. Elle renvoie 1 en cas de succès, 0 sinon.

Par exemple :
- `extraction_colonne("Buffalo:Bill:10",col,':',2)` renvoie 1 et suite à cet appel, `col` doit contenir la chaîne "Bill" ;
- `extraction_colonne("Buffalo:Bill:10",col,':',4)` renvoie 0 (et le contenu de `col` n'a pas d'importance, on pourra y placer une chaîne vide par exemple) ;
- `extraction_colonne("toto::titi",col,':',2)` renvoie 1 et `col` doit contenir une chaîne vide.

Question 4 : Donnez le code de la fonction `extraction_colonne`. (Utilisez la fonction précédente !)

1.4 Programme principal (4 points)

Question 5 : En utilisant les fonctions précédentes, donnez le code de la fonction `main` du programme `couper`. En particulier, vous devrez vérifier que

1. le nombre d'arguments est correct,
2. le premier paramètre est un caractère,
3. le second paramètre est un entier strictement positif, et que
4. le troisième paramètre est un fichier accessible en lecture.

Votre programme devra bien entendu afficher également le contenu de la colonne demandée, lorsque cela est possible. Si on rencontre une ligne qui ne comporte pas suffisamment de colonnes, on sautera simplement une ligne à cet endroit.

Par commodité, vous pouvez supposer que toutes les chaînes lues ont une taille inférieure à une constante `MAXTAILLE` que vous aurez définie.

1.5 Makefile (2 points)

Supposons que l'ensemble des fonctions précédentes sont réparties dans des fichiers sources comme suit :
- `conversion.h` contient la déclaration de la fonction `versEntier`,
- `meschaines.h` contient les déclarations des fonctions `trouve_colonne` et `extraction_colonne`,
- `mesio.h` contient la déclaration de la fonction `readline`,
- `conversion.c` contient le code de la fonction `versEntier`,
- `meschaines.c` contient le code des fonctions `trouve_colonne` et `extraction_colonne`,
- `mesio.c` contient le code de la fonction `readline`, et
- `couper.c` contient le programme principal `main`.

Voici le résultat de la commande `grep \#include\ \" *.c` dans le répertoire contenant tous les fichiers sources :

```
conversion.c:#include "conversion.h"
couper.c:#include "mesio.h"
couper.c:#include "meschaines.h"
couper.c:#include "conversion.h"
meschaines.c:#include "meschaines.h"
mesio.c:#include "mesio.h"
```

Question 6. Dessinez le graphe de dépendance puis écrivez un `Makefile` qui permet de construire l'exécutable `couper` à partir de ces fichiers sources.

2 Script de détection de plagiat (7 points)

Dans cette section, vous devez écrire un ensemble de scripts en shell qui permettra à vos enseignants de détecter des programmes rendus par les étudiants qui seraient des plagiats de programmes rendus par d'autres étudiants [1]. Pour cela, nous allons, dans un premier temps, tenter de comparer le degré de similitude entre deux fichiers, et, dans un second temps, appliquer cette comparaison à des projets étudiants constitués de plusieurs fichiers. Pour toutes les questions, sauf mention contraire, vous n'avez pas à traiter les cas d'erreur d'utilisation des scripts demandés.

2.1 Comparer deux fichiers (5 points)

Comparer le degré de similitude entre deux fichiers est un exercice difficile, pour lequel des compagnies comme Google dépensent, chaque année, des millions. Nous nous contenterons ici de comparaisons simples, qui nous donneront une approximation de la réponse dont nous nous contenterons.

Nous commencerons par comparer le nombre de lignes de nos deux fichiers pour avoir une idée de leur similitude. La commande `wc` avec l'option `-l` affiche le nombre de lignes lues sur l'entrée standard. Par exemple :

```
wc -l < toto.txt
```

affichera

```
18
```

si le fichier `toto.txt` contient 18 lignes.

Question 1 : Écrivez un script qui prend deux noms de fichiers en arguments et qui affiche à l'écran :
- un message d'erreur si l'un des deux fichiers n'est pas accessible en lecture
- un message indiquant que les deux fichiers sont similaires s'ils ont le même nombre de lignes
- rien sinon

1. D'ailleurs, pour info, il est bon de savoir que l'université Joseph Fourier est dotée d'un logiciel de détection de plagiat. Pour plus de détails voir : `https://tice.ujf-grenoble.fr/outils-et-pedagogie/logiciel-anti-plagiat`

Le nombre de lignes n'est pas un indicateur très fiable pour déterminer si deux fichiers se ressemblent. Nous allons donc supposer que nous disposons d'une commande nommée `similarites` [2] qui, étant donnés deux noms de fichiers passés en arguments, pour chaque bloc de lignes communes (contenu dans chacun des deux fichiers) affiche, sur une ligne, le nombre de lignes qu'il contient. Par exemple :

```
similarites toto.txt tata.txt
```

Affichera

```
13
5
8
```

si les fichiers `toto.txt` et `tata.txt` contiennent 3 blocs de respectivement 13, 5 et 8 lignes en commun.

Question 2 : Écrivez un script qui prend deux noms de fichiers en arguments et qui affiche à l'écran :

- un message indiquant que le premier fichier est plagié sur le second si 90% ou plus de ses lignes sont présentes dans le second fichier
- rien sinon

2.2 Comparer des projets (2 points)

Ici, nous supposons qu'un projet étudiant est un ensemble de fichiers sources contenus dans un (unique) répertoire.

Question 3 : Écrivez un script qui prend deux noms de répertoires en arguments et qui pour tout couple (x, y) de noms de fichiers tel que x est un fichier contenu dans le premier répertoire et y un fichier contenu dans le second, appelle le script de la question 2 avec x et y en arguments.

Vous pouvez considérer que le script écrit à la question précédente s'appelle `plagiat.sh`.

2. Pour info, cette commande peut être implémentée de la manière suivante :

```
similarites () {
    diff --unchanged-group-format="%dN%c'\012'" --changed-group-format="" $1 $2
}
```