Exercices sur l'allocation dynamique et les entrées/sorties : Commandes utilitaires sur flux de caractères

On cherche à programmer une commande `textutil` qui se comporte comme une version modifiée et simplifiée de quelque commandes utilitaires sur flux de caractères : `nl`, `rev`, `tac`, `fgrep`, `paste`, `cut`, selon l'option indiquée en argument.

La commande `textutil`

Son usage est le suivant :

```
$ ./textutil -help
Usage:
./textutil -h|-help
./textutil -nl START STEP skip|* [FILE...]
./textutil -rev [FILE...]
./textutil -tac [FILE...]
./textutil -fgrep WORD [FILE...]
./textutil -paste SEP [FILE...]
./textutil -cut SEP NUM [FILE...]
```

Il peut y avoir un nombre arbitraire de noms de fichiers `FILE` en fin de commande que l'on ouvre tous avec `fopen()`. Lorsqu'un nom de fichier est un *dash* `"-"`, on fait en sorte que `fopen()` ouvre l'entrée standard via `"/dev/fd/0"`. Également, lorsqu'il n'y a aucun fichier `FILE` en fin de commande, on lit l'entrée standard via `stdin`.

La commande `textutil -nl START STEP skip|*` affiche chaque ligne en les numérotant à partir de `START` et par incrément de `STEP`. Lorsque la dernière option vaut `"skip"`, les lignes vides `"\n"` sont sautées (elles sont toujours affichées mais elles ne sont pas numérotées). Pour toute autre valeur, les lignes vides sont numérotées comme les autres. La numérotation continue pour tous les fichiers :

```
$ (echo abc ; echo ; echo rst; echo xyz) | ./textutil -nl 100 10 skip
   100  abc

   110  rst
   120  xyz
```

```
$ (echo abc ; echo ; echo rst; echo xyz) | ./textutil -nl 100 10 noskip
   100  abc
   110
   120  rst
   130  xyz
```

```
$ ./textutil -nl 100 10 noskip <(echo aaa) <(echo bbb) <(echo ccc; echo ddd)
   100  aaa
   110  bbb
   120  ccc
   130  ddd
```

La commande `textutil -rev` affiche les lignes de chaque fichier en les renversant. Mais le retour à la ligne reste à sa place.

```
$ (echo abc ; echo rst; echo xyz) | ./textutil -rev
cba
tsr
zyx
```

La commande `./textutil -tac` affiche les lignes de chaque fichier dans l'ordre inverse, c'est-à-dire de la dernière à la première. Au contraire de `nl` ou `rev`, cette commande est donc obligée de lire tout le fichier avant de pouvoir afficher quoi que ce soit :

```
$ (echo abc ; echo rst; echo xyz) | ./textutil -tac
xyz
rst
abc
```

La commande `textutil -fgrep WORD` affiche les lignes qui contiennent le mot `WORD`. Lorsque la sortie standard est le terminal, l'occurrence trouvée est entourée de balises de marquage définies par les variables d'environnement `MARKUP_START` et `MARKUP_END`. En l'absence de la définition simultanée de ces deux variables, on utilise les valeurs par défaut `"\x1B[31m"` et `"\x1B[0m"`, qui sont les séquences d'échappement ANSI pour afficher l'occurrence en rouge. Lorsque la sortie standard est redirigée vers autre chose qu'un terminal, aucune balise n'est utilisée.

```
$ export MARKUP_START='<' MARKUP_END='>'
$ (echo aabbcc ; echo bbaaccaa; echo bbcc ) | ./textutil -fgrep  aa
<aa>bbcc
bb<aa>ccaa
```

```
$ (echo aabbcc ; echo bbaaccaa; echo bbcc ) | ./textutil -fgrep  aa | cat
aabbcc
bbaaccaa
```

La commande `textutil -paste SEP` lit les lignes de chaque fichiers en parallèle, et affiche le collage de chaque ligne de même numéro, en les collant avec le séparateur SEP. Lorsqu'un fichier n'a pas assez de lignes vis-à-vis des autres, il est réputé avoir des lignes vides pour compléter :

```
$ ./textutil -paste '<*>' <(echo a; echo b; echo c) <(echo x) <(echo r; echo s)
a<*>x<*>r
b<*><*>s
c<*><*>
```

La commande `textutil -cut SEP NUM` considère la chaîne `SEP` comme un séparateur de colonne, et affiche la colonne numéro `NUM` de chaque ligne. En l'absence d'une telle colonne, une ligne vide est affichée :

```
$ (echo 'aa<*>bb<*>cc'; echo 'rr' ; echo 'xx<*>yy') | ./textutil -cut '<*>' 2
bb

yy
```

On souhaite aussi que ces commandes se comportent bien lorsqu'il y a des `'\0'` sur une ligne, ou lorsque la dernière ligne ne se termnine pas par `'\n'` :

```
$ (printf "a\n" ; printf "r\0s\n" ; printf "z") | ./textutil -tac
z
rs
a
```

```
$ (printf "a\n" ; printf "r\0s\n" ; printf "z") | ./textutil -tac | hexdump -c
0000000   z  \n   r  \0   s  \n   a  \n
0000008
```

1 Gestion d'une ligne : le module `Line`

La présence potentielle de `'\0'` sur une ligne interdit d'utiliser `fgets()` pour lire une ligne. En effet, cette fonction ne retourne aucune information quant au nombre de caractères lus, et on ne peut donc savoir si des caractères ont été lus après le `'\0'`. On codera donc notre propre fonction de lecture de ligne en bouclant sur la fonction `getc()` de lecture de caractère. Les caractères d'une lignes sont alloués dynamiquement avec `malloc()` et `realloc()` au fur et à mesure que la ligne lue croît et dépasse sa capacité.

C'est la structure `Line` qui se charge de représenter une ligne :

```
typedef struct Line {
  char * chars;  // malloc'ed() and can have some '\0' in the middle
  size_t length; // this count includes all '\0' except the last one
} Line;
```

La fonction `Line_read()` lit une ligne sur un flux d'entrée. On utilise `initialCapacity` pour la taille du tableau pointé par `chars` lors de son `malloc()` initial. À chaque fois que la capacité du tableau pointé par `chars` est dépassée, on réalloue ce tableau avec `realloc()` en doublant sa capacité. Lorsqu'on rencontre `EOF` sans avoir lu de caractères, la fonction génère une chaîne vide `""` (c'est-à-dire de longueur 0 et ne contenant que le `'\0'` terminal). Lorsqu'on rencontre `EOF` après avoir lu des caractères, c'est que la dernière ligne du fichier ne se termine pas proprement par `'\n'`. Une fois la ligne complètement lue, on réalloue le tableau pointé par `chars` sur sa longueur définitive.

```
Line Line_read (FILE * file,  size_t initialCapacity);
```

Lorsqu'une ligne obtenue via `Line _read()` n'est plus utilisée, la mémoire allouée dynamiquement pour ses caractères est libérée par la fonction `Line_clean()`.

```
void Line_clean (Line line);
```

Il est pratique de disposer des trois prédicats suivants pour une ligne `line` : le prédicat `Line_isEnd()` teste si une ligne est la chaîne vide `""`, représentant `EOF`; le prédicat `Line_endsWithNL()` teste si la ligne est proprement terminée par `'\n'` ; enfin le prédicat `Line_startsWithNL()` teste si la ligne est vide, c'est-à-dire de longueur 1 et uniquement constituée de `'\n'`.

```
bool Line_isEnd        (Line line);
bool Line_endsWithNL   (Line line);
bool Line_startsWithNL (Line line);
```

Les deux fonctions de recherche suivantes sont pratiques pour la programmation de `-fgrep` et `-cut` : `Line_findString()` recherche une chaîne dans une ligne. Rappelons que cette dernière peut contenir des `'\0'` intermédiaires et donc un simple appel à `strstr()` ne suffit pas. La variante `Line_findNthString()` recherche la `nth` occurrence pour `nth >= 1`. En l'absence de la chaîne recherchée, `NULL` est retrourné par les deux fonctions.

```
char * Line_findString    (Line line, char const string[]);
char * Line_findNthString (Line line, char const string[], int nth);
```

La présence potentielle de `'\0'` intermédiaires sur une ligne empêche d'utiliser simplement `printf()` pour afficher une ligne. On écrit donc une fonction `Line_print()` affichant une ligne dans un flux de caractères `file`. La variante `Line_printFromTo()` permet d'afficher un fragment de ligne de l'index `start` inclus à l'index `end` exclu. Elle est utile pour la commande `-cut`, ainsi que pour la commande `-fgrep` lorsqu'on utilise des balises de marquage. Enfin, la variante `Line_printBackward()` est utile pour la commande `rev` et affiche une ligne à l'envers.

```
void Line_print         (Line line, FILE * file);
void Line_printFromTo   (Line line, size_t start, size_t end, FILE * file);
void Line_printBackward (Line line, FILE * file);
```

2 Gestion du contenu entier d'un fichier : le module `Content`

Si la plupart des commandes peuvent afficher le traitement d'une ligne immédiatement après l'avoir lue et oublier les lignes précédemment lues, ce n'est pas le cas de la commande `textutil -tac` qui doit mémoriser le contenu entier du fichier avant de pouvoir faire son affichage.

C'est la structure `Content` qui mémorise le contenu d'un fichier structuré en lignes :

```
typedef struct Content {
  Line * lines;
  size_t lineCount;
} Content;
```

La structure `Content` possède un tableau pointé par `lines` de `lineCount` structures de type `Line`. Ce tableau est alloué dynamiquement avec `malloc()` et `realloc()` par la fonction de lecture `Content_read()`. Cette fonction réalloue le tableau pointé par `lines` via `realloc()` en doublant sa capacité à chaque fois que celle-ci est atteinte. La dernière ligne est toujours la chaîne vide, représentant `EOF`. Une fois le contenu du fichier entièrement lu, on réalloue le tableau une dernière fois sur sa longueur réelle.

```
Content Content_read (FILE * file, size_t initialCapacity);
```

La mémoire allouée par `Content_read()` est libérée par `Content_clean()` :

```
void Content_clean (Content content);
```

3 Le module `TextUtil` : l'option de commande `-nl`

La fonction `TextUtil_runNlOnFile()` lit les lignes d'un fichier `file` et les affiche dans le fichier `output`, en les numérotant à partir de `numStart` par incréments de `numStep`. Le booléen `skipEmpty` contrôle si on affiche les lignes vides sans les numéroter. La prochaine ligne à numéroter est retournée par la fonction (c'est utile si l'on numérote les lignes de plusieurs fichiers en invoquant autant de fois la fonction).

```
size_t TextUtil_runNlOnFile (FILE * file, size_t numStart, size_t numStep,
                             bool skipEmpty, FILE * output);
```

En déduire la fonction `TextUtil_runNl` qui gère l'option de commande `textutil -nl`. La fonction retourne le nombre de fichiers pour lesquel il y a eu un échec d'ouverture.

```
int TextUtil_runNl (int argc, char * argv[], FILE * output);
```

4 Le module `TextUtil` : l'option de commande `-rev`

La fonction `TextUtil_runRevOnFile()` lit les lignes d'un fichier `file` et les affiche chacune renversée dans le fichier `output`.

```
void TextUtil_runRevOnFile (FILE * file, FILE * output);
```

En déduire la fonction `TextUtil_runRev` qui gère l'option de commande `textutil -rev`. La fonction retourne le nombre de fichiers pour lesquels il y a eu un échec d'ouverture.

```
int  TextUtil_runRev (int argc, char * argv[],  FILE * output);
```

5 Le module `TextUtil` : l'option de commande `-tac`

La fonction `TextUtil_runTacOnFile()` lit les lignes d'un fichier `file` et les affiche de la dernière à la première dans le fichier `output`. Cette fonction fait usage de la structure `Content` car il faut mémoriser tout le flux avant de pouvoir afficher quoi que ce soit.

```
void TextUtil_runTacOnFile (FILE * file, FILE * output);
```

En déduire la fonction `TextUtil_runTac` qui gère l'option de commande `textutil -tac`. La fonction retourne le nombre de fichiers pour lesquels il y a eu un échec d'ouverture.

```
int TextUtil_runTac (int argc, char * argv[], FILE * output);
```

6 Le module `TextUtil` : l'option de commande `-fgrep`

La fonction `TextUtil_runFgrepOnFile()` lit les lignes d'un fichier `file` et affiche dans le fichier `output` celles qui contiennent la chaîne `word`. La première occurrence du mot est encadré par les balises de marquage `markupStart` et `markupEnd`.

```
void TextUtil_runFgrepOnFile (FILE * file, char const word[],
     char const markupStart[], char const markupEnd[], FILE * output);
```

En déduire la fonction `TextUtil_runFgrepWithMarkup` qui gère l'option de commande `textutil -fgrep` avec pour balises de marquage `markupStart` et `markupEnd`. La fonction retourne le nombre de fichiers pour lesquels il y a eu un échec d'ouverture.

```
int TextUtil_runFgrepWithMarkup (int argc, char * argv[],
    char const markupStart[], char const markupEnd[], FILE * output);
```

En déduire la fonction `TextUtil_runFgrep` qui gère l'option de commande `textutil -fgrep` en prenant pour balises de marquage deux chaînes vides lorsque `output` n'est pas un flux vers un terminal (utiliser `isatty()` et `fileno()` pour le savoir). Sinon, prendre les valeurs des variables d'environnement `MARKUP_START` et `MARKUP_END` lorsque ces deux variables sont simultanément définies (utiliser `getenv()` pour obtenir les valeurs). Sinon, prendre les séquences d'échappement ANSI `"\x1B[31m"` et `"\x1B[0m"` qui permettent d'écrire en rouge sur le terminal et de rétablir la couleur par défaut. La fonction retourne le nombre de fichiers pour lesquels il y a eu un échec d'ouverture.

```
int TextUtil_runFgrep (int argc, char * argv[], FILE * output);
```

7 Le module `TextUtil` : l'option de commande `-cut`

La fonction `TextUtil_printCutLine()` prend en argument une ligne `line` organisée en colonnes séparées par la chaîne `sep`, et affiche dans le fichier `output` sa colonne numéro `num`.

```
void TextUtil_printCutLine (Line line, char const sep[], int num, FILE * output);
```

La fonction `TextUtil_runCutOnFile()` lit les lignes d'un fichier `file` organisé en colonnes séparées par la chaîne `sep`, et affiche dans le fichier `output` la colonne numéro `num` de chaque ligne, ou une ligne vide à défaut.

```
void TextUtil_runCutOnFile (FILE * file, char const sep[], int num, FILE * output);
```

En déduire la fonction `TextUtil_runCut` qui gère l'option de commande `textutil -cut`. La fonction retourne le nombre de fichiers pour lesquels il y a eu un échec d'ouverture.

```
int TextUtil_runCut (int argc, char * argv[], FILE * output);
```

8 Le module `TextUtil` : l'option de commande `-paste`

La fonction `TextUtil_printPastedLines()` prend en argument un tableau de lignes `lines`, et affiche dans le fichier `output` leur collage en les séparant par la chaîne `sep`.

```
void TextUtil_printPastedLines (Line lines[], int lineCount,
                                char const separator[], FILE * output);
```

La fonction `TextUtil_runPasteOnFiles()` lit en parallèle les lignes d'un tableau de fichiers `files` et affiche dans le fichier `output` le collage des lignes de même numéro, en les séparant avec la chaîne `sep`. En absence d'un nombre suffisant de lignes dans un fichier, la ligne vide est utilisée.

```
void TextUtil_runPasteOnFiles (FILE * files[], int fileCount,
                               char const separator[], FILE * output);
```

En déduire la fonction `TextUtil_runPaste` qui gère l'option de commande `textutil -paste`. La fonction retourne le nombre de fichiers pour lesquels il y a eu un échec d'ouverture.

```
int TextUtil_runPaste (int argc, char * argv[], FILE * output);
```

On utilise les fonctions auxiliaires suivantes pour faciliter la lecture de lignes lues en parallèle à partir de plusieurs fichiers dont certains ont déjà atteint `EOF` :

- La fonction `TextUtil_openFiles()` ouvre `fileCount` fichiers `files` en mode `mode` à partir de leur `fileCount` noms de fichier `filenames`. Quand un nom de fichier est le *dash* `"-"`, on ouvre via `fopen()` l'entrée standard `"/dev/fd/0"`. Le nombre d'erreurs d'ouverture est retourné, et les entrées dans `files` sont `NULL` pour les fichiers non-ouverts.

  ```
  int TextUtil_openFiles (FILE * files[], int fileCount,
                          char * const filenames[], char const mode[]);
  ```

- La fonction `TextUtil_closeFiles()` ferme les `fileCount` fichiers du tableau `files`, sauf les entrées `NULL` qui correspondent à des échecs d'ouverture.

  ```
  void TextUtil_closeFiles (FILE * files[], int fileCount);
  ```

- La fonction `TextUtil_readOneLinePerFile()` lit une ligne pour chacun des `fileCount` fichiers du tableau `files` qui n'ont pas encore atteint `EOF`, ce qu'indique le tableau de Booléens `eofs` de même longueur. La ligne lue est stockée dans l'entrée correspondante du tableau de lignes `lines`. Pour les fichiers qui ont déjà atteint `EOF` avant l'appel de la fonction, on alloue dynamiquement la chaîne vide `""` de longueur nulle via `strdup()`. On le fait également pour les fichiers qui atteignent `EOF` lors de cet appel sans qu'aucun caractère ne puisse être lu, en faisant passer l'entrée correspondante du tableau `eofs` à `true`.

  ```
  int TextUtil_readOneLinePerFile (Line lines[], FILE * files[],
                                   bool eofs[], int fileCount);
  ```

- Enfin, la fonction `TextUtil_cleanLines()` permet de libérer la mémoire dynamiquement allouée pour les caractères des `lineCount` lignes du tableau `lines`. (Attention, le tableau `lines` lui-même n'est pas alloué dynamiquement).

  ```
  void TextUtil_cleanLines (Line lines[], int lineCount);
  ```