Perfectionnement à la programmation en C
Fiche de TP 6

L2 Informatique 2021-2022

Compter les mots

Ce TP se déroule en **une seule séance** et est à faire par binômes. Le travail réalisé doit être envoyé au plus tard exactement **une semaine après le début** de la séance de TP. Il sera disposé sur la plate-forme prévue à cet effet et constitué des programmes répondant aux questions et des éventuels fichiers annexes qui peuvent être demandés.

L'objectif de ce TP est d'écrire un utilitaire opérant sur des fichiers texte. Nous utiliserons ici les notions de

— manipulation de fichiers textes et de chaînes de caractères ;

— entrée et sortie standard ;

— arguments d'un programme ;

— structures de données dynamiques d'ensembles ;

— modularisation d'un projet relativement abstrait ;

— écriture de tests.

Contrairement aux sujets de TP précédents, nous donnons ici une spécification du projet à lire, à comprendre et, finalement, à satisfaire. Les définitions de types et de fonctions sont laissées pour la plupart libres de choix.

1 Spécification

Il est toujours utile, lors de l'écriture d'un texte, d'éviter d'employer trop souvent des mêmes mots. Détecter l'usage trop fréquent d'un même terme est aussi utile pour rédiger des textes agréables à lire. C'est pour cette raison qu'il est demandé ici de concevoir un utilitaire `clm` (« *compter les mots* ») paramétré par le nom d'un fichier texte (qui ne contient que des caractères ASCII). L'utilitaire compte les occurrences de chaque mot du fichier et produit comme réponse un affichage sur la sortie standard donnant pour chaque mot du fichier texte de départ son nombre d'occurrences.

Plus précisément, est considéré comme un mot toute suite de caractères contigus délimitée à gauche et à droite par des espaces, des fins de lignes, des tabulations, des caractères de ponctuation ou des début ou fin de fichier. Toute suite de caractères contigus qui contient un chiffre ou tout autre caractère non alphabétique est ignorée. De plus, dans le comptage des occurrences d'un mot, on ne fait pas de différence entre majuscules et minuscules.

La commande `clm X`, où `X` est le nom d'un fichier texte, affiche sur la sortie standard des lignes qui contiennent chacun un mot en minuscules suivi d'espaces, suivies du nombre (en base dix) d'occurrences du mot en question dans `X`. Chaque mot y est mentionné suivant l'ordre d'apparition de sa première occurrence dans le texte. Par exemple, à partir du fichier texte `texte` de contenu

```
TROIS Un un un1un deux DeUx
deux deuxdeux un quatre4
```

la commande `clm texte` affiche sur la sortie standard

```
trois    1
un       3
deux     3
deuxdeux 1
```

Notons que les valeurs affichées sont alignées en fonction de la longueur du plus long mot de sorte à produire un affichage agréable. Cette convention d'affichage sera adoptée implicitement dans toute la suite.

Il figurera une option `-a` faisant en sorte de trier les mots de la réponse dans l'ordre lexicographique. Par exemple, la commande

```
clm texte -a
```

affiche

```
deux     3
deuxdeux 1
trois    1
un       3
```

Il figurera aussi une option `-n` faisant en sorte de trier les mots de la réponse selon leur nombre d'occurrences de manière décroissante. Les mots ayant un même nombre d'occurrences apparaissent dans l'ordre lexicographique. Par exemple, la commande

```
clm texte -n
```

affiche

```
deux     3
un       3
deuxdeux 1
trois    1
```

Tout ceci constitue la première partie de l'application. Dans cette deuxième partie, nous souhaitons mettre en place un mécanisme pour renseigner sur les mots qui suivent ou qui précèdent d'autres. En pratique, ceci permet de détecter, par exemple, des erreurs de grammaire.

Ainsi, la commande `clm X -s MOT`, où `X` est le nom d'un fichier texte et `MOT` un mot, affiche sur la sortie standard des lignes qui contiennent chacune chaque mot `u` qui suit `MOT` dans `X` suivi d'espaces, suivies du nombre (en base dix) d'occurrences de `u` situées après `MOT`. Chaque mot `y` est mentionné suivant l'ordre d'apparition de sa première occurrence dans le texte qui suit une occurrence de `MOT`. Par exemple, à partir du fichier texte `texte` de contenu

```
ab aaa abc a ab
aaa ab a ab aaa
ab ab b b ab a ab b
```

la commande

```
clm texte -s ab
```

affiche

```
aaa 3
a   2
ab  1
b   2
```

Ceci renseigne, par exemple, qu'il y a exactement deux mots `b` qui suivent immédiatement une occurrence du mot `ab`. Comme dans la première partie de la spécification, on ajoutera les options `-a` et `-n` pour trier les mots qui apparaissent dans la réponse. Par exemple,

```
clm texte -s ab -n
```

affiche

```
aaa 3
a   2
b   2
ab  1
```

Il figurera également dans l'application une option `-p` qui possède la même spécification que l'option `-s` mais qui prend en compte les mots qui précèdent le mot spécifié dans la commande.

La dernière partie de l'application est la plus intéressante à utiliser et à mettre en œuvre. Nous souhaitons maintenant recenser les expressions composées d'un nombre fixé de mots. Plus précisément, est considérée comme une expression toute suite de mots séparés par une ou plusieurs espaces ou une ou plusieurs fins de lignes.

Il est ainsi question de disposer d'une commande `clm X -e N`, où `X` est le nom d'un fichier texte et `N` un entier strictement positif exprimé en base dix. Cette commande affiche sur la sortie standard une ligne pour chaque expression composée de `N` mots extraite de `X` suivie d'espaces, suivies du nombre (en base dix) d'occurrences de l'expression en question dans `X`. Par exemple, à partir du fichier texte `texte` de contenu

```
un un un deux trois trois
un un trois deux trois
```

la commande

```
clm texte -e 2
```

affiche

```
un un       3
un deux     1
deux trois  2
trois trois 1
trois un    1
un trois    1
trois deux  1
```

Comme dans les autres parties de la spécification, on ajoutera les options `-a` et `-n` pour trier les mots qui apparaissent dans le fichier résultat. Par exemple,

```
clm texte -e 2 -a
```

affiche

```
deux trois  2
trois deux  1
trois trois 1
trois un    1
un deux     1
un trois    1
un un       3
```

2 Écriture du programme

Exercice 1. (Conception du projet)

L'objectif de cet exercice est de concevoir une architecture viable pour le projet présenté.

1. Lire attentivement **l'intégralité du sujet** avant de commencer à répondre aux questions. La description du sujet constitue une spécification de projet.
2. Établir la liste de toutes les fonctionnalités demandées, option par option.
3. Une fois ceci fait, proposer un découpage en modules cohérent du projet. Pour chaque module proposé, décrire les types qu'il apporte ainsi que ses objectifs principaux.
4. Au fil de l'écriture du projet, il est possible de se rendre compte que le découpage initialement prévu n'est pas complet ou adapté. Si c'est le cas, mentionner l'historique de ses modifications.
5. Maintenir un `Makefile` pour compiler le projet.

Exercice 2. (Tests)

Les tests sont des éléments de première importance dans le processus d'écriture d'un projet. Ils sont également primordiaux pour la maintenance d'un projet. Ils servent à s'assurer que chaque partie du programme fonctionne et sont utiles pour capturer des erreurs de programmation à leur source.

1. Avant de commencer l'écriture du projet, créer un module `Test`. Ce module va inclure tous les autres modules (exception faite pour le demi module principal) et va servir à tester les fonctions du projet. Il contient pour le moment une fonction

```c
int test();
```

   Cette fonction renverra `1` si tous les tests se passent bien et `0` si au moins un test produit une erreur. Il doit s'agir de la seule fonction visible depuis l'extérieur du module `Test`.

2. Au cours de l'écriture du projet, pour chaque fonction importante `fct` nouvellement écrite, ajouter dans le module `Test` une fonction

```c
int test_fct();
```

   qui réalise des tests cohérents de la fonction `fct`. Cette fonction `test_fct` doit renvoyer `1` si tous les tests se passent bien et `0` si au moins un test produit une erreur. Les tests sont construits en appelant `fct` avec des arguments pour lesquels la réponse est connue. Le test consiste à vérifier si la fonction répond conformément à la réponse prévue. Pour chaque test aboutissant à une erreur, la fonction `test_fct` affiche sur la sortie standard des informations pour identifier le test problématique.

   Voici à présent un exemple complet de ce mécanisme. Supposons que soyons dans le contexte où un projet contient un module `Tableau` regroupant des fonctions de manipulation de tableaux. Ce module apporte la fonction

```c
int derniere_occurrence(int *tab, int n,
        int val) {
    int i;

    assert(tab != NULL);
    assert(n >= 0);

    for (i = n - 1 ; i >= 0 ; --i)
        if (tab[i] == val)
            return i;
    return -1;
}
```

   qui renvoie l'indice de la dernière occurrence de `val` dans le tableau d'entiers `tab` de taille `n`. Lorsque `tab` n'a aucune occurrence de `val`, la fonction renvoie `-1`. Pour tester cette fonction, nous ajoutons dans le module `Test` la fonction

```c
int test_derniere_occurrence() {
    int t1[7] = {1, 1, 2, 1, 3, 1, 4};
    int t2[4] = {3, 1, 4, 2};
    int t3[0] = {};

    if (derniere_occurrence(t1, 7, 1) != 5){
        printf("ERREUR test 1\n");
        return 0;
    }
    if (derniere_occurrence(t2, 4, 3) != 0){
```

```
        printf("ERREUR test 2\n");
        return 0;
    }
    if (derniere_occurrence(t2, 4, 8) != -1){
        printf("ERREUR test 3\n");
        return 0;
    }
    if (derniere_occurrence(t3, 0, 1) != -1){
        printf("ERREUR test 4\n");
        return 0;
    }

    return 1;
}
```

Comme nous pouvons l'observer, cette fonction réalise trois tests où l'on compare ce que la fonction testée renvoie effectivement aux la valeurs attendues que l'on connaît à l'avance.

Travail à faire : écrire de telles fonctions de test pour les fonctions les plus importantes du projet. En moyenne, 6 à 12 fonctions doivent ainsi être testées dans le projet. Pour chacune d'elles, il faut imaginer les tests les plus pertinents à réaliser et qui recouvrent tous les cas de figure possible.

3. Compléter la fonction `test` du module `Test`.

   Important : ceci doit être fait au fur et à mesure de l'écriture de l'application et surtout pas à la fin. Ceci constitue en effet une aide au développement du projet.

4. Incorporer une option `-test` au programme qui exécute la fonction `test` du module `Test`. Il est important, lors de la phase de programmation du projet ou encore lors d'une de ses mises-à-jour, de lancer ces tests pour vérifier que tout fonctionne comme attendu.

Quelques conseils :

— il est possible d'utiliser les fonctions `strcpy`, `strlen` et `strcmp` (entre autres) du module `string` ;

— il est possible d'utiliser les fonctions `isalpha`, `isblank` et `tolower` (entre autres) du module `ctype` (se référer aux documentations de ces fonctions accessibles via le manuel) ;

— bien décomposer les tâches à accomplir. Par exemple, il peut être utile d'avoir une fonction paramétrée par un chemin vers un fichier et qui créé la liste des mots qu'il contient ;

— concernant les tests, il est possible de créer un fichier texte sur lequel des requêtes vont être formulées. Il s'agira de vérifier que les principales fonctions du programme donnent les réponses attendues. Il est inutile et trop laborieux de tester toutes les fonctions : il suffit de tester celles principales, œuvrant pour répondre aux options de lancement décrites de `clm`.

Remarque importante 1 : *le bon découpage en modules du projet est une étape essentielle. Cette étape, si elle est bien menée, fait gagner beaucoup de temps et facilite l'écriture du projet. Il est ainsi fortement déconseillé de commencer à écrire du code avant d'avoir conçu l'architecture du projet.*

Remarque importante 2 : *l'exercice 2 compte beaucoup dans l'évaluation de ce TP. Il est important de proposer pour chaque fonction testée un jeu de tests cohérent et complet. Il est possible que l'écriture des tests occupe une proportion significative et majoritaire de temps de développement.*

Remarque importante 3 : *il n'est pas question dans ce TP d'utiliser les structures de données et les algorithmes les plus efficaces qui répondent au problème (il est en effet possible de proposer des solutions très efficaces en étant astucieux). L'objectif principal est de concevoir un projet* **sans faille** *dans son architecture globale. Cependant, un programme efficace sera apprécié.*