

Corrigé TD 2

Encore des exercices simples

1 Comptage

1.1 Première version

Dans une première version, on ne va pas s'occuper de reconnaître les lettres et les chiffres seulement, mais compter toutes les occurrences de **tous** les caractères. Du coup, c'est assez simple:

- on sait qu'il y a 256 caractères \Rightarrow on va déclarer le tableau `compteurs` de 256 cases qui nous permettra d'avoir un compteur chaque caractère possible;
- En C, comme les caractères sont des (petits) entiers, on trouvera donc le compteur associé au caractère `'X'` dans la case `compteurs['X']`.

Bien sûr, toutes les cases du tableau `compteurs` doivent être initialisées à 0 avant de commencer à parcourir le fichier d'entrée. Pour cela, on peut faire une boucle qui parcourt les 256 case et leur affecte la valeur 0. On peut aussi utiliser la notation `{}` qui permet d'initialiser un tableau d'entiers à 0 (on verra pourquoi un peu plus tard en cours).


Le début de notre programme peut donc être:

```
int compteurs[256] = {};  
int c;  
  
while ((c=getchar())!=EOF)  
    compteurs[c] += 1;
```

Une fois que l'on a lu tout fichier d'entrée, on doit afficher le nombre de chiffres et de lettres lues. Pour cela on va faire 3 boucles (une pour les chiffres, une pour les minuscules et une pour les majuscules). Cela donne:

```
for (c = '0'; c <= '9'; c++)  
    if (compteurs[c] != 0) printf("%c: %2d fois\n", c, compteurs[c]);  
for (c = 'a'; c <= 'z'; c++)  
    if (compteurs[c] != 0) printf("%c: %2d fois\n", c, compteurs[c]);  
for (c = 'A'; c <= 'Z'; c++)  
    if (compteurs[c] != 0) printf("%c: %2d fois\n", c, compteurs[c]);
```

Noter que l'on n'affiche pas ici les compteurs nuls pour ne pas afficher des informations inutiles.

On obtient donc le code suivant [l_comptage_v0.c](#) .

1.2 Seconde version

La première version présente deux inconvénients:

- l'inconvénient majeur de cette version est que l'on met à jour des compteurs qui ne seront jamais affichés (les compteurs pour les caractères qui ne sont ni des lettres, ni des chiffres).
- un autre problème est que l'on utilise plus de mémoire que nécessaire, puisqu'on pourrait se contenter d'un tableau de 10 cas pour les chiffres et de deux tableaux de 26 cases (voire un seul, si on se contente de mettre ensemble majuscules et minuscules).

Ainsi, pour compter les caractères, il va falloir reconnaître les différentes classes de caractères. Par exemple, pour savoir si `c` est une minuscule, cela peut se faire en vérifiant que `('a' <= c && c <= 'z')`.

Il faut ensuite trouver la case du tableau où se trouve le compteur correspondant au caractère `c` rencontré. Pour cela, il faut calculer le décalage de la lettre dans l'alphabet. Pour une lettre minuscule, ce décalage est `c - 'a'`.

Pour les chiffres, c'est la même chose:

- le test est `('0' <= c && c <= '9')`
- l'indice d'un digit dans le tableau est `c - '0'`

On obtient donc le code suivant `l_comptage.c`:

```
/* Affichage du nombre d'occurrences de lettres et chiffres lus sur stdin */
#include <stdio.h>

int main() {
    int letters[26] = {};
    int digits[10] = {};
    int c;

    //
    // Lecture du fichier
    //
    while ((c=getchar())!=EOF)
        if ('0' <= c && c <= '9') digits[c - '0'] += 1;           // chiffre
        else
            if ('a' <= c && c <= 'z') letters[c - 'a'] += 1;       // minuscule
            else
                if ('A' <= c && c <= 'Z') letters[c - 'A'] += 1;   // majuscule

    //
    // Affichage des caractères vus (pas d'affichage si le compteur est à 0)
    //
    for (c = '0'; c <= '9'; c++)
        if (digits[c - '0'] != 0) printf("%c: %2d fois\n", c, digits[c - '0']);
    for (c = 'a'; c <= 'z'; c++)
        if (letters[c - 'a'] != 0) printf("%c: %2d fois\n", c, letters[c - 'a']);

    return 0;
}
```

Notes:

1. Les tableaux `letters` et `digits` ont bien leur cases initialisées à 0 (grâce à l'utilisation de l'agrégat `{}` lors de la déclaration)
2. Les majuscules sont comptées dans la case de la lettre minuscule correspondante dans le tableau `letters`.

2 Commande wc

Pas de difficulté ici pour compter les lignes et les caractères. Le seul point difficile dans cet exercice consiste à savoir quand on a un nouveau mot. Le piège est de penser que l'on a un nouveau dès que l'on voit un séparateur. En fait, c'est un peu plus compliqué: on a un nouveau mot lorsque le caractère courant n'est pas un séparateur et que le caractère précédent est un séparateur. Pour éviter des tests trop fastidieux, on écrit donc une fonction

```
int separateur(char c);
```

pour voir si `c` est un séparateur (`c` est à dire espace, tabulation ou newline). Cette fonction renvoie 1 si `c` est un séparateur et 0 sinon.

On obtient donc le code suivant [2_wc.c](#) :

```
// Version simplifiée de la commande wc qui compte sur stdin
#include <stdio.h>

int separateur(char c) {
    return (c == ' ' || c == '\t' || c == '\n');
}

int main() {
    int lines, words, chars, c;
    int prec = '\n';

    lines = words = chars = 0;

    while((c=getchar())!=EOF) {
        chars += 1;
        if (c == '\n')
            lines += 1;
        if (separateur(prec) && !separateur(c)) // !x veut dire 'NOT x'
            words += 1;
        prec = c; // mettre à jour prec
    }
    printf("lines: %d\nwords: %d\nchars: %d\n", lines, words, chars);

    return 0;
}
```

Note:

Ici, `prec` est initialisé à `'\n'`. Ainsi, si le premier caractère du fichier n'est pas un séparateur, on "détecte" une transition de type *séparateur* → *non-séparateur*, et le compteur de mots est bien incrémenté.

3 Insertion dans un tableau

On veut écrire la fonction

```
void insertion(int array[], int nbval, int n);
```

qui insère l'entier `n` dans le tableau `array` rempli de `nbval` valeurs.

3.1 Première implémentation

Une première version de cet algorithme pourrait être

1. Trouver la position `j` où doit se faire l'insertion
2. Décaler les éléments à droite de `j` de 1 case vers la droite
3. Mettre `n` dans la case `j` du tableau

Une implémentation directe de cet algorithme pourrait donc être:

```
void insertion(int array[], int nbval, int n)
{
    int i, j;

    // Trouver la position j où doit se faire l'insertion
    for (j = 0; j < nbval; j++) {
        if (array[j] > n) break; // break sort de la boucle
    }

    // décaler les éléments à droite de j de 1 case vers la droite
    for (i = nbval; i > j; i--) {
        array[i] = array[i-1];
    }


    // Mettre n dans array[j]
    array[j] = n;
}
```

3.2 Seconde implémentation

En fait, on peut faire bien mieux en combinant les deux premières boucles. Puisque de toutes les façons on va insérer notre nouveau nombre `n`, on peut commencer par décaler systématiquement d'un élément vers la droite, tant que cela est nécessaire. On obtient un algorithme, plus beau (mais moins évident à trouver...).

```
void insertion(int array[], int nbval, int n)
{
    int j;

    for (j = nbval; (j > 0) && (array[j-1] > n); j--) { // Décalage des éléments > à n
        array[j] = array[j-1];
    }
    array[j] = n; // Insertion
}
```

Le programme complet est donné ci-dessous [3_insert.c](#) 

```

/*
 * 3_insert.c  -- Insertion dans un tableau
 */

#include <stdio.h>

// Insertion de n à sa place dans le tableau array rempli de nbval valeurs
// Si le tableau n'est pas assez grand ... ça plante
//---
void insertion(int array[], int nbval, int n)
{
    int j;

    for (j = nbval; (j > 0) && (array[j-1] > n); j--) { // Décalage des éléments > à n
        array[j] = array[j-1];
    }
    array[j] = n;                                     // Insertion
}
// ---

void print_array(int array[], int nbval){
    int i;

    printf("[ ");
    for (i = 0; i < nbval; i++)
        printf("%d ", array[i]);
    printf("]\n");
}

int main(void)
{
    int array[100];
    int n, items, nbval = 0;

    do {
        printf("Entrer un entier: ");
        items = scanf("%d", &n);
        if (items == 1 && n >= 0) {
            /* Le nombre saisi est positif */
            insertion(array, nbval++, n);
            print_array(array, nbval);
        }
        // Sortir quand:
        //   - items != 1 (on a pas réussi à lire un nombre erreur ou fin de fichier (on a tapé ^D) )
        //   - Le tableau est plein (nbval == 100)
    } while (items == 1 && nbval < 100);
}

```

4 Dump hexadécimal

Dans ce programme, on passe par un tableau de caractères de longueur 16. Ce tableau permet de conserver les valeurs que l'on a lues et qu'il faudra imprimer sous forme d'entiers puis sous forme de caractères (ce tableau est affiché dès qu'il est rempli dans la fonction `imprimer_ligne`).

Noter que pour afficher un élément du tableau

- **sous forme hexadécimale**: on utilise le format `"%02x"` (`x` pour hexa, `2` pour être sur deux caractères et `0` pour indiquer que les valeurs qui n'utilisent qu'un chiffre doivent être précédées d'un espace – ce qui fait vraiment pro!!!)
- **sous forme de caractère**: on utilise simplement le format `"%c"`. Par contre, il faut éviter l'affichage de caractères «bizarres» (passage à la ligne, beep sonore, tabulation, ...) qui casseraient le résultat (voir sujet). Du coup, on utilise ici l'opérateur ternaire dans l'expression

```
( ' ' <= c && c <= '~' ) ? c : '.'
```

qui se lit: *si c est dans le bon intervalle, l'afficher sinon afficher un point.*

Attention, quand on est en fin de fichier, on a peut-être des caractères dans le tableau qui n'ont pas été imprimés (si la taille du fichier n'est pas un multiple de 16, c'est effectivement le cas). Il faudra donc penser à imprimer le tableau quand on sort de la boucle `while`.

Le code du programme [4_dump_hexa.c](#) est donné ci-dessous:

```

/*
 * 4_dump_hexa.c      -- Dump hexadécimal
 */

#include <stdio.h>
#define MAXLIGNE 16

void imprimer_ligne(char ligne[], int lg)
{
    int j;

    // Affichage du code des caractères
    for (j = 0; j < MAXLIGNE; j++) {
        if (j < lg)
            // Impression en hexadécimal . Le cast sert au cas où le char serait signé.
            printf("%02x ", (unsigned char) ligne[j]);
        else
            // Afficher des espaces pour alignement
            printf("   ");
    }

    // Affichage des caractères
    printf(" | ");
    for (j = 0; j < lg; j++) {
        char c = ligne[j];

        printf("%c", (' ' <= c && c <= '~') ? c : '.');
    }
    printf("\n");
}

int main()
{
    int c, i = 0;
    char ligne[MAXLIGNE];

    while ((c = getchar()) != EOF) {
        ligne[i++] = c;
        if (i == MAXLIGNE) {           // Le tableau est plein, l'imprimer et le "vider"
            imprimer_ligne(ligne, i);
            i = 0;
        }
    }

    // Quand on est en fin de fichier, on a peut-être des caractères
    // dans le tableau (si la taille du fichier n'est pas un multiple de
    // MAXLIGNE, c'est le cas). Imprimer le morceau de tableau
    if (i != 0) imprimer_ligne(ligne, i);

    return 0;
}

```

Note importante:

Dans ce programme, comme dans tous les programmes qui utilisent la forme

```

while ((c=getchar()) != EOF) {
    ...
}

```

on doit déclarer la variable `c` comme un `int` et non pas un `char`. En effet, la fonction `getchar()` est capable de renvoyer 257 résultats différents (un des 256 caractères possibles de l'intervalle `[0..255]` et la valeur spéciale `EOF` (généralement codée en la valeur `-1`)).

Si `c` est égale à 255 (en binaire, cela s'écrit `11111111`), la comparaison `(c != EOF)` est

- **vraie** si `c` est déclarée comme un `int` (en effet l'entier `-1` est différent de l'entier `255`).
- **fausse** si `c` est déclarée comme un `char`. Cela est dû au fait qu'il est nécessaire de convertir ce caractère en entier avant de faire la comparaison. Or, la conversion de la valeur (8 bits) `11111111` en entier donne l'entier (32 bits) `11111111111111111111111111111111`¹ qui est la représentation binaire du nombre `-1` (vous verrez pourquoi dans le cours d'architecture). Du coup, quand on tombe sur le caractère de valeur 255, on pense que l'on est en fin de fichier :-)

5 Exercices facultatifs

Ces exercices ne sont pas corrigés.

NOTES

1. On suppose ici que les nombres entiers sont codés sur 32 bits, on obtient un résultat similaire si on code les entiers sur 64 bits.↩