DM3 MP2I : lecture et écriture dans un fichier

Thèmes

1. Lecture/écriture dans un fichier en C.
2. Structures.
3. Compilation séparée.
4. Recherche de facteurs et de sous-mots.
5. Fichiers d'image en noir et blanc.

Avant de commencer ce devoir, il faut se documenter sur la lecture/écriture dans des fichiers en C (par exemple ici).

1 Prise en main

Pour cette partie de prise en main (PM), le projet est organisé selon les fichiers suivants :

Fichiers d'en-têtes — le fichier `bibli.h` contient toutes les inclusions de fichiers d'en-têtes des fonctions de bibliothèques.

— le fichier `pm.h` est le fichier d'en-tête des fonctions de cette partie. Il contient tous les prototypes de fonctions.

Fichiers de code — Le fichier `mainpm.c` contient tous les tests et le `main`.

— Le fichier `pm.c` contient tous les codes des fonctions de cette partie mais aucun test.

L'inclusion du fichier d'en-tête `string.h` est interdite. Je supprimerai toute occurrence de `#include <string.h>`.

Exercice 1. Dans cet exercice, on veut retrouver le comportement de la commande bash `wc` qui donne le nombre de lignes, de mots et de caractères dans un fichier de texte.
Considérons le fichier `toto1.txt` dont le contenu en langage créole réunionnais est :

```
1  Toto lé      motivé
2 Gogo lé fatigué
3 La Dodo, salé bon mém!
```

Listing 1 – contenu de toto1.txt

Dans ce fichier (donné dans l'archive), il y a des espaces, des tabulations et des sauts de lignes.

Listing 2 – Rendu wc

```
$ wc toto1.txt
 3 11 62 toto1.txt
```

Pour réaliser cet exercice, il peut être utile de lire le fichier caractère par caractère avec `fscanf` auquel on passe le spécifieur de format `%c`.

Q1 Écrire une fonction `int compte_c(FILE * f)` qui renvoie le nombre de caractères d'un fichier.

Q2 Écrire une fonction `int compte_l(FILE * f)` qui compte les lignes du fichier.

Q3 On veut compter les mots du fichier. Deux mots sont séparés par un ou plusieurs espace(s), caractère(s) de tabulation ou de fin de ligne. Écrire la fonction `int compte_m(FILE * f)` qui renvoie le nombre de mots d'un fichier.

Donnons quelques définitions :

Définition 1. On dit qu'un mot x est *facteur* d'un mot m s'il existe u, v, deux mots (éventuellement vides) tels que $m = uxv$.

Exemple 1. Le mot **sol** est facteur de **insolent**.

Définition 2. Un mot $x = x_1 \dots x_k$ de k lettres est appelé un *sous-mot* d'un mot m s'il existe $k+1$ mots $u_0, \dots, u_k$ (éventuellement vides) vérifiant :

$$m = u_0 x_1 u_1 x_2 u_2 \dots x_k u_k$$

Exemple 2. Le mot **games** est un sous-mot de **zzghhhaaamtttezsdddd**.

Exercice 2. Dans cet exercice, on se donne un fichier t et un mot m.

Q1 Écrire la fonction `bool facteur(FILE* f, char *mot)` qui prend en paramètre un flot parcourant en mode texte un fichier et un mot. La fonction retourne un bouléen si le mot est un facteur du texte dans le fichier.

L'algorithme gère une position p dans le mot :

— qui est incrémentée si le caractère courant du texte est égal à la lettre p du mot,

— et remise à zéro sinon.

Cet algorithme est dit « naïf ».

```c
void test_facteur(){
  char * nom = "toto1.txt";
  FILE * f = fopen(nom, "r");

  char* facteur1 = "salé";
  printf("%s est facteur de %s : %d\n",
    facteur1,nom,facteur(f,facteur1));
  // remettre le curseur au début
  //du fichier
  rewind(f);

  char* facteur2 = "gogi";
  printf("%s est facteur de %s : %d\n",
    facteur2,nom,facteur(f,facteur2));

  fclose(f);
}
```

Listing 3 – Test facteur

Listing 4 – Rendu test facteur

```
$ ./pm
salé est facteur de toto1.txt : 1
gogi est facteur de toto1.txt : 0
```

Q2 Écrire la fonction `bool subword(FILE * f, char* mot)` qui détermine si le mot `m` est un facteur du texte dans le fichier pointé par `f`.

```c
void test_subword(){
  char * nom = "toto1.txt";
  FILE * f = fopen(nom, "r");

  char* facteur1 = "mofaLa";
  printf("%s est sous-mot de %s : %d\n"
    ,facteur1,nom,subword(f,facteur1));
  rewind(f);
  char* facteur2 = "lotiz";
  printf("%s est sous-mot de %s : %d\n"
    ,facteur2,nom,subword(f,facteur2));
  rewind(f);

  fclose(f);
}
```

Listing 5 – Test subword

Listing 6 – Rendu test subword

```
$ ./pm
mofaLa est sous-mot de toto1.txt : 1
lotiz est sous-mot de toto1.txt : 0
```

2 Images en noir et blanc

Dans cette partie, nous retrouvons les fonctions `mymalloc` et `myfree` du devoir 2. Toutes les allocations/libérations se font avec ces fonctions.

Les fichiers d'images au format `PBM` représentent des images en noir et blanc. Un pixel est codé par un caractère 0 pour blanc et 1 pour noir. Dans ce devoir, nous nous intéressons aux fichiers `PBM` en mode *texte*. Un fichier de ce type commence toujours par la chaîne de caractère `P1` (dite « nombre magique ». Après ce nombre magique, viennent :

- un caractère d'espacement (espace, tabulation, nouvelle ligne) ;
- la largeur de l'image (nombre de pixels, écrit explicitement sous forme d'un nombre en caractères ASCII) ;
- la hauteur de l'image (idem) ;
- un caractère d'espacement ;
- les données de l'image : succession des valeurs associées à chaque pixel, l'image est codée ligne par ligne en partant du haut, chaque ligne est codée de gauche à droite.
- Toutes les lignes commençant par croisillon `#` sont ignorées (ce sont des lignes de commentaires).

Dans l'archive, on trouve un fichier `j.pbm` représentant le caractère `J`. Il est reproduit figure 1.

Aucune ligne ne doit dépasser 70 caractères. Pour les gros fichiers, il est possible de poser une ligne de texte par pixel et sauter une ligne de texte pour chaque ligne d'image.

Pour afficher une image `pbm`, il faut installer le visionneur `eog` :

```
$ sudo apt install eog
```

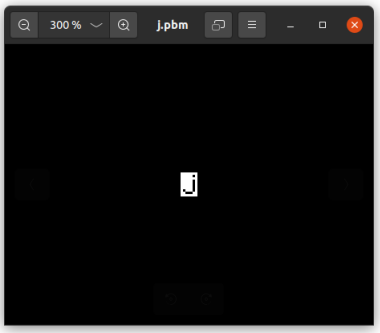
```
P1
# Un exemple bitmap de la lettre "J"
7 10
#une ligne vide
0 0 0 0 0 0 0
0 0 0 0 0 1 0
0 0 0 0 0 0 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 0 0 0 0 1 0
0 1 0 0 0 1 0
0 0 1 1 1 0 0
0 0 0 0 0 0 0
```

FIGURE 1 – Un J

Listing 7 – Commande pour afficher `j.pbm`

```
$ eog j.pbm
```

Entrons la commande ci-dessus.



Une fenêtre contenant l'image s'ouvre alors. Comme notre image est toute petite (70 pixels carrés), il peut être utile de zoomer (avec la combinaison de touche `CTRL +`).

Le repère que nous utilisons pour les images est représenté figure 2. Les pixels ont des coordonnées entières. Les pixels visibles sont situés dans le 1/4 de plan $\begin{cases} x \geq 0 \\ y \geq 0 \end{cases}$. Par exemple, le point du j de la figure 1 a pour coordonnées $(1, 5)$.

Les fichiers pour ce projet sont décrits ci-dessous :

Fichiers d'en-têtes — le fichier `bibli.h` est le même que pour le projet précédent.

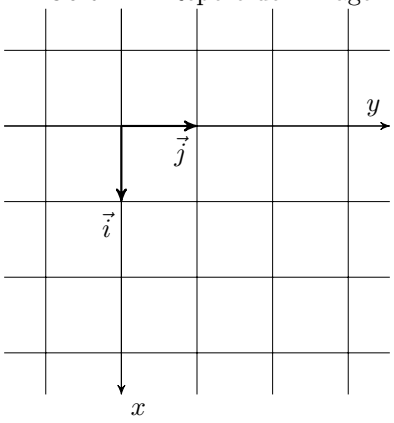
— le fichier `im.h` est le fichier d'en-tête des fonctions de cette partie. Il contient tous les prototypes de fonctions, la déclaration de la structure `image` et un alias de type pour cette structure.

Fichiers de code — Le fichier `mainim.c` contient tous les tests et le `main`.

— Le fichier `im.c` contient tous les codes des fonctions de cette partie mais aucun test.

Q1 Définir dans `im.h` une structure `image` qui contient un champ `mat` de type `bool **` (matrice de bouléens) et deux champs `nbl,nbc` nombre de lignes et nombre de colonnes.

FIGURE 2 – Repère de l'image



Lorsqu'un coefficient de la matrice vaut `false`, c'est que le pixel correspondant est blanc ; `true` et le pixel est noir.

L'alias que nous utilisons pour cette structure est `im`.

Q2 Définir dans `im.h` une structure `point` qui contient un champ `color` de type `float` (une couleur en niveau de gris, à priori entre 0 et 1) et deux champs `x,y` (abscisse et ordonnée) de type `float`.

L'alias que nous utilisons pour cette structure est `pt`. L'utilité de ces points de coordonnées non entières sera décrite plus tard.

Nous nous donnons les moyens de passer d'un objet `image` à un fichier PBM et réciproquement :

Q3 Écrire la fonction `void save(char * nom, im * img)` qui prend en paramètre un nom de fichier et une image. La fonction met le contenu de l'image dans le fichier en respectant la syntaxe PBM.

```
43 void test_save(){
44   char * croix = "croix.pbm";
45   int n = 10; int m=7;
46   im * img = make_im(n,m,true);//
       image noire
47   for (int i=0; i<img->nbl; i++)
48     img->mat[i][img->nbc/2]=false;
49   for (int i=0; i<img->nbc; i++)
50     img->mat[img->nbl/2][i]=false;
51   save(croix,img);
52   free_image(img);
53 }
```

Listing 8 – Écriture et sauvegarde d'une croix

Listing 9 – Fichier produit

```
$ cat croix.pbm
P1
7 10

1110111
1110111
1110111
1110111
1110111
0000000
1110111
1110111
1110111
1110111
```

À tester avec `eog croix.pbm`.

Q4 Écrire la fonction `im * load(char * nom)` qui prend en paramètre un nom de fichier PBM et retourne un pointeur sur image. Il faudra bien prendre en compte la ligne vide entre les déclarations de nombres de lignes et de colonnes et le premier coefficient 0 ou 1. De plus, l'absence possible d'espace entre les coefficients doit être géré.

```c
void test_load(){
  char * croix = "croix.pbm";
  im * img = load(croix);
  display_im(img);
  free_image(img);
}
```

Listing 10 – Test load

Listing 11 – Rendu Test load

```
$ ./im
 1 1 1 0 1 1 1
 1 1 1 0 1 1 1
 1 1 1 0 1 1 1
 1 1 1 0 1 1 1
 1 1 1 0 1 1 1
 0 0 0 0 0 0 0
 1 1 1 0 1 1 1
 1 1 1 0 1 1 1
 1 1 1 0 1 1 1
 1 1 1 0 1 1 1
```

Nous allons maintenant tracer des segments de droite. Il faut commencer par tracer un point.

Q5 Soit x une abscisse décimale comprise entre deux abscisses entières x_0, x_1. On suppose qu'en x_1 la couleur est c_1, et qu'en x_0 la couleur est c_0 (c_1 et c_0 sont deux flottants désignant un *niveau de gris*).

On suppose qu'il y a une *évolution linéaire* entre la couleur en x_0 et celle en x_1.

Écrire la fonction `float lineaire(float x, int x0, int x1,float c0, float c1);` qui renvoie la couleur (en niveau de gris) estimée de x si x est entre x_0 et x_1.

```c
void test_lineaire(){
  printf("lineaire(2.25,2,3,6,7)=%.3f\n",
     lineaire(2.25,2,3,6,7));

  printf("lineaire(2.83,2,3,6,8)=%.3f\n",
         lineaire(2.83,2,3,6,8));

   printf("lineaire(1.9,1,2,0.2,0.8)=%.3f
    \n",
     lineaire(1.9,1,2,0.2,0.8));
}
```

Listing 12 – Évolution linéaire des couleurs

Listing 13 – Rendu

```
$ ./im
lineaire(2.25,2,3,6,7)=6.250
lineaire(2.83,2,3,6,7)=7.660
lineaire(1.9,1,2,0.2,0.2)=0.740
```

Considérons un point virtuel de coordonnées (x, y) décimales. Il s'agit de trouver la couleur que ce point virtuel aurait dans une image donnée (on rappelle que les coordonnées des pixels sont entières). Le point virtuel est ainsi entouré de 4 pixels dans l'image initiale dont les abscisses sont comprises entre `(int x)` et `(int) x + 1` et les ordonnées entre `(int) y` et `(int y) +1` (voir figure 3).

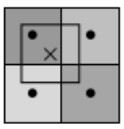


FIGURE 3 – Illustration du point virtuel entouré des 4 pixels voisins dans l'image

Par convention, si les coordonnées du pixel sont hors de l'image (ce qui se produit en particulier pour une abscise ou une ordonnée négative), la couleur du point virtuel est considérée comme blanche.

Pour trouver la couleur du point virtuel, on utilise la valeur des 4 pixels voisins en réalisant une approximation bilinéaire qui consiste :

— prenant les deux pixels voisins de la première ligne, à trouver la valeur du niveau de gris du point virtuel en supposant une évolution linéaire selon la coordonnée y entre le pixel de gauche et le pixel de droite ;

— à faire de même en prenant les pixels de la deuxième ligne ;

— enfin en travaillant sur la coordonnée x, à supposer une évolution linéaire entre les deux valeurs trouvées aux deux étapes précédentes.

Si l'un des pixels de la figure 3 n'est pas sur l'image, on considère que sa couleur est 0 (blanche).

La valeur α retournée constitue donc la couleur (en niveau de gris) du point virtuel. Cette technique est efficace pour les images en niveau de gris. Pour les images en noir et blanc, il suffit d'arrondir α au plus proche entre 0 et 1.

Q6 Écrire la fonction `float bilineaire(im * img, float x, float y)` qui renvoie la couleur que devrait avoir un pixel p virtuel de coordonnées (x, y). On rappelle que si p n'est pas dans les limites de l'image, il doit avoir la couleur blanche (c.a.d 0).

```
100 void test_bilineaire(){
101   int n = 4; int m=6;
102   im * img = make_im(n,m,false);
103   float x = 1.9, y = 1.8;
104   printf("x=%f,y=%f\n",x,y);
105
106   img->mat[2][1]=true;img->mat
        [2][2]=true;
107   display_im(img);
108   printf("bilineaire(img,x,y)=%f\n",
109       bilineaire(img,x,y));
110
111   img->mat[1][2]=true;
112   display_im(img);
113   printf("bilineaire(img,x,y)=%f\n",
114           bilineaire(img,x,y));
115
116   free_image(img);
117 }
```

Listing 14 – Test bilinéaire

Listing 15 – Rendu

```
$ ./im
x=1.900000,y=1.800000
 0 0 0 0 0 0
 0 0 0 0 0 0
 0 1 1 0 0 0
 0 0 0 0 0 0
bilineaire(img,x,y)=0.900000
 0 0 0 0 0 0
 0 0 1 0 0 0
 0 1 1 0 0 0
 0 0 0 0 0 0
bilineaire(img,x,y)=0.980000
```

Q7 Écrire la fonction `void putpixel(im* img,int i,int j,bool c)` qui colorie le pixel (i, j) avec la couleur c.

on donne la fonction suivante qui implémente une partie de l'algorithme de tracé continu de segment proposé par Jack E. Bresenham en 1962.

```
1  void trace_quadrant_est(im *img, pt p0, pt p1){
2    //efficace si dx< dy
3    assert(p1.color == p0.color);//même couleur
4    float dx = p1.x-p0.x, dy = p1.y-p0.y;
5    putpixel(img,(int) p0.x,(int) p0.y,p0.color);
6    for (int i=1;i<dx;i++){
7      putpixel(img,p0.x + i,
8           p0.y + floor(0.5 + dy * i / dx),
9           p0.color);
10          }
11   putpixel(img,(int) p1.x,(int) p1.y,p1.color);
12   }
```

```c
void test_trace_quadrant_est(){
  int n = 10; int m=10;
  im * img = make_im(n,m,true);//
    image noire
  pt p0 = {0.,0.,0.};
  pt p1 = {7.,3.,0.};
  trace_quadrant_est(img,p0,p1);
  printf("trace_quadrant_est(img,");
  display_pt(p0);printf(",");
    display_pt(p1);
  printf(")\n");

  display_im(img);
  save("seg1.pbm",img);
  free_image(img);
}
```

Listing 16 – Test trace quadrant est

Listing 17 – Rendu

```
$ ./im
trace_quadrant_est(img,{0.000000,0.000000,0}
                       {7.000000,3.000000,0})
 0 1 1 1 1 1 1 1 1 1
 0 1 1 1 1 1 1 1 1 1
 1 0 1 1 1 1 1 1 1 1
 1 0 1 1 1 1 1 1 1 1
 1 1 0 1 1 1 1 1 1 1
 1 1 0 1 1 1 1 1 1 1
 1 1 1 0 1 1 1 1 1 1
 1 1 1 0 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1
 1 1 1 1 1 1 1 1 1 1
```

On observe une certaine continuité dans la ligne de zéros.

Il y a deux problèmes avec cette fonction :

— Le premier problème (pas si grave) arrive si `p1.x < p0.x` : le segment n'est pas tracé. En effet, le « est » dans le nom signifie que le second pixel est à l'est du premier. Il suffit donc de faire attention à l'ordre des coefficients passés en argument.

— Le second problème peut être testé avec l'appel `trace_quadrant_est(img,p0,p1)` ; lorsque p_0 a pour coordonnées $(3, 0)$ et p_1 vaut $(5, 8)$. Les étudiants devront bien identifier ce qui donne cette impression de discontinuité.

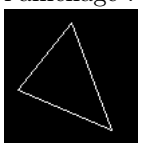
Q8 Écrire la fonction `void trace_quadrant_sud(im *img, pt p0, pt p1)` appelée lorsque le second pixel est au sud du premier et qui règle sutout le second problème précédent. Cette fonction outil ne sera pas testée lors de la correction.

Q9 Écrire la fonction `void trace_segment(im *img, pt p0, pt p1)` qui trace un segment entre les points de coordonnées entières p_0 et p_1.

```c
void test_trace_segment(){
  int n = 100; int m=100;
  im * img = make_im(n,m,true);//
    image noire
  pt p0 = {100.,50.,0.};
  pt p1 = {60.,10.,0.};
  pt p2 = {90.,80.,0.};

  trace_segment(img,p0,p1);
  trace_segment(img,p1,p2);
  trace_segment(img,p0,p2);

  save("triangle.pbm",img);

  free_image(img);
 }
```

Listing 18 – Tracé d'un triangle

Avec `eog triangle.pbm&`, on obtient l'affichage :



Un point M est à l'intérieur d'un triangle A, B, C si et seulement si il existe $(\alpha, \beta) \in \mathbb{R}^+$ tels que

$$\overrightarrow{AM} = \alpha\overrightarrow{AB} + \beta\overrightarrow{AC}$$

avec $\alpha + \beta \leq 1$.

Q10 Écrire une procédure `is_in(pt M, pt A, pt B, pt C)` qui renvoie un bouléen indiquant si le point M est à l'intérieur du triangle ABC.

```
173 void test_is_in(){
174   pt p0 = {1.,5.,0.};
175   pt p1 = {5.,1.,0.};
176   pt p2 = {9.,8.,0.};
177   pt M = {6.2,4.1,0.};
178
179   display_pt(p0);printf("\n");
180   display_pt(p1);printf("\n");
181   display_pt(p2);printf("\n");
182   printf("M=");display_pt(M);
183   printf("\n");
184   printf("is_in(M,p0,p1,p2)=%d\n",
        is_in(M,p0,p1,p2));
185
186   M.x = 7.2;M.y = 3.1;
187   printf("M=");display_pt(M);
188   printf("\n");
189   printf("is_in(M,p0,p1,p2)=%d\n",
        is_in(M,p0,p1,p2));
190 }
```

Listing 19 – Test d'appartenance

Listing 20 – Rendu appartenance

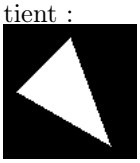
```
$ ./im
{1.000000,5.000000,0}
{5.000000,1.000000,0}
{9.000000,8.000000,0}
M={6.200000,4.100000,0}
is_in(M,p0,p1,p2)=1
M={7.200000,3.100000,0}
is_in(M,p0,p1,p2)=0
```

Q11 Écrire la procédure `draw_triangle(im *img, pt A, pt B, pt C)` qui prend en paramètres 3 points de coordonnées entières et de même couleur. La fonction remplit le triangle ABC avec la couleur de A.

```
199 void test_draw_triangle(){
200   int n = 100; int m=100;
201   im * img = make_im(n,m,true);//
        image noire
202   // 3 points de la même couleur :
203   pt p0 = {10.,50.,false};
204   pt p1 = {50.,10.,false};
205   pt p2 = {90.,80.,false};
206
207   display_pt(p0);printf("\n");
208   display_pt(p1);printf("\n");
209   display_pt(p2);printf("\n");
210
211   draw_triangle(img,p0,p1,p2);
212   save("triangle_blanc.pbm",img);
213   free_image(img);
214 }
```

Listing 21 – Remplissage d'un triangle

Avec `eog triangle_blanc.pbm`, on obtient :



Beaucoup de choses restent à faire :

— Tracé de lignes polygonales ;

— Rotation d'une figure d'un certain angle autour d'un point donné ;

— Tracé de fractales etc.

Mais il faut bien s'arrêter un jour, non ?