AP1 TP

10

Semaine 10 Fichiers - 1/1

En C++ nous faisons la différence entre les variables qui permettent de manipuler un fichier : les fichiers logiques (*descripteurs*, *flux*...), et les fichiers physiques désignés par leur nom (une chaîne de caractères). Nous devons donc établir la liaison entre les fichiers *logiques* et les fichiers *physiques*. Pour cela, nous disposons de la bibliothèque `fstream`.

Les fichiers `fstream`

Déclaration :

```
fstream f ;
```

`f` est un *flux* (dans le même sens que les flux d'entrée/sortie `cin` et `cout`) qui sera associé à un fichier physique lors d'une opération d'ouverture. Ainsi il peut exister plusieurs flux correspondant à un même fichier physique (autrement dit plusieurs processus peuvent accéder au même fichier simultanément).

Ouverture et Fermeture : Avant toute utilisation d'un fichier `fstream` il faut l'**ouvrir** selon le mode souhaité (lecture, écriture, concaténation). Voici les différentes manières d'ouvrir un fichier :

```
1  // ios::in   ->  ouverture en lecture
2  f.open ("nomFichier",ios::in) ;
3  // ios::out  ->  ouverture en ecriture
4  f.open ("nomFichier",ios::out) ;
5  // ios::app  ->  ouverture pour concatenation
6  f.open ("nomFichier",ios::out|ios::app) ;
```

Après toute utilisation un fichier doit être **fermé** (quel que soit le mode d'ouverture). Pour cela nous utilisons la primitive :

```
f.close () ;
```

Une règle d'or consiste à toujours ouvrir et fermer un fichier dans la même fonction. De même, dans la mesure du possible, il est conseillé de faire les opérations de lecture et d'écriture là où le fichier a été ouvert (ce n'est pas toujours possible ...).

Différentes méthodes :

```
1  f.fail ()     // renvoie true si la derniere operation sur le flux a
       echoue
2  f.eof ()      // renvoie true si la fin de fichier a ete atteinte
3                // lors de la derniere lecture
```

Exercice 1 : Analyse d'un exemple

1. Commentez le programme ci-dessous. Notez en particulier la manière dont se font les lectures et écritures dans le fichier.

```
1  #include <iostream>
2  #include <fstream>
3  #include <cstdlib>
4  using namespace std;
5
6  int main()
7  {
```

```
8   int i, nombre;
9   fstream f;
10
11  f.open( "ficNb", ios::out );
12  if( f.fail() ) {
13    cerr << "ouverture en ecriture impossible" << endl;
14    return EXIT_FAILURE;
15  }
16  for( i=0 ; i<10 ; i++ )
17    f << i << " ";
18  f.close();
19
20  f.open( "ficNb", ios::in );
21  if( f.fail() ) {
22    cerr << "ouverture en lecture impossible" << endl;
23    return EXIT_FAILURE;
24  }
25  f >> nombre;
26  while( !f.eof() ) {
27    cout << "nombre lu = " << nombre << endl;
28    f >> nombre;
29  }
30  f.close();
31  return EXIT_SUCCESS;
32 }
```

Tests	Résultat(s) attendu(s)	Résultat(s) observé(s)
Contenu de "ficNB" après exécution du programme		
Affichage après exécution du programme		

2. Reprenez le programme ci-dessus (`/net/Bibliotheque/AP1/TPSem10/exempleFichier.cc`) et découpez-le en deux fonctions (`saisie` et `affichage`), implémentez-les et utilisez-les dans un programme principal. Comment peut-on passer des fichiers en paramètres ?

   Il y a deux approches possibles. On manipule soit des chaînes de caractères correspondant aux noms des fichiers, soit des identificateurs (variables de type `fstream`). La première sera préférée (pour l'instant) dans un souci d'**encapsulation** des opérations.
   Cependant, il peut arriver que la seconde solution soit inévitable. Le fichier (ou flux) doit alors être passé par référence comme indiqué dans le prototype :

   ```
   void fonctionFichier (fstream & fich);
   ```

3. Écrivez une fonction permettant d'ajouter les entiers allant de 10 à 19 dans un fichier passé en paramètre.

Tests	Résultat(s) attendu(s)	Résultat(s) observé(s)
Tester avec les deux modes de passage de paramètre		

Exercice 2 : Image numérique

Nous allons considérer dans cet exercice les images numériques en niveaux de gris. Une image numérique est un ensemble de pixels (“picture elements”), chacun pouvant prendre un niveau de gris qui lui est propre. Usuellement, l'intensité maximale que peut prendre un pixel est 255. Ainsi un pixel blanc sera codé par la valeur 255, un pixel noir par la valeur 0 et un pixel gris moyen par un 128. L'image ainsi définie peut être stockée sous plusieurs formes ; elle peut être compressée (exemple du JPEG) ou alors les valeurs de chacun des pixels peuvent être stockées à la suite dans un fichier. C'est le cas du format PGM à partir duquel nous allons travailler. Comme les pixels sont les éléments des lignes et des colonnes, on peut représenter une image par un tableau à deux dimensions de niveaux de gris.

Le stockage d'une image au format PGM doit suivre des rêgles strictes ; seront écrits à la suite dans le fichier :

1. l'en-tête :
   - ➤ "P2" (définissant quelle valeur du format est utilisée) ;
   - ➤ la largeur et la hauteur ;
   - ➤ l'intensité maximale que peut prendre un niveau de gris ;
2. stockage des niveaux de gris des pixels.

Par exemple :

```
P2
100 126
255
75
80
75
81
81
83
82
81
83
81
83
86
87
82
...
```

L'objectif de cet exercice est la manipulation d'images au format PGM (ascii) et plus particulièrement l'implémentation d'un algorithme de détection de contours.

Notre image de référence sera la suivante `essai.pgm`

Il s'agit du "Golden Gate" `/net/Bibliotheque/AP1/TPSem10/essai.pgm`. Pour le visualiser, lancer `display essai.pgm`. De plus, vérifier la structure du fichier en lançant `emacs`.

Nous allons implémenter une suite de manipulations d'images qui va nous permettre de détecter les contours.

Nous utiliserons dans la suite la structure de données `TImage` :

```
const int MAX=300;
typedef int TImage [MAX][MAX];
```

1. **Saisie et sauvegarde**.
   (a) Écrire une action qui permet à partir d'un fichier au format PGM (ascii) de remplir une variable de type `TImage`.
   (b) Écrire une action qui à partir d'une variable de type `TImage` construit un fichier au format PGM (ascii).
2. **Filtre**. Afin d'effectuer la détection des contours, nous devons tout d'abord appliquer plusieurs masques à l'image. Un masque permet de calculer les valeurs d'un pixel à partir d'une pondération des pixels de son voisinage.

   Voici la structure de données ainsi que les masques que nous utiliserons :

```
1  const int TF=3;
2
3  typedef int TFiltre[TF][TF];
4
5  const TFiltre gradH={{-1,0,1},
6                       {-2,0,2},
7                       {-1,0,1}};
8
9  const TFiltre gradV={{-1,-2,-1},
10                      {0,0,0},
11                      {1,2,1}};
```

   Écrire la fonction :
   `void filtrerImage(TImage src, int longueur, int largeur, TImage & res, const TFiltre f);`
   qui permet de construire l'image `res` à partir du passage du masque `f` sur `src`.
3. **Contours**. Écrire l'action `contourImage` qui étant données les deux images `src1` et `src2` construit une image `res` dont la valeur de chaque pixel p de coordonnées i, j est $\sqrt{(src1[i][j])^2 + (src2[i][j])^2}$.
4. **Inversion**. Écrire une action qui inverse la valeur de chaque pixel.
5. **Seuil**. Étant donnée une valeur de seuil, on souhaite réaliser l'opération suivante : pour tout pixel de l'image, si la valeur du pixel est supérieure au `seuil` alors sa valeur passe à 255, sinon à 0.
6. **Détection des contours**
   (a) Appliquer `gradH` à l'image initiale, on obtient `src1`.
   (b) Appliquer `gradV` à l'image initiale, on obtient `src2`.
   (c) Appliquer l'action `contourImage` à partir de `src1` et `src2`.
   (d) Inverser l'image résultante et seuillez-la à 125.
   (e) Observez le résultat.

Exercice 3 : Fusion de deux fichiers triés

Écrire un algorithme qui fusionne deux fichiers d'entiers triés dans un troisième fichier trié.