Centre Informatique pour les Lettres et les Sciences Humaines

Apprendre C++ avec Qt : Leçon 7
Utiliser des fichiers de données

De nombreux programmes exigent une méthode permettant de leur fournir des données autrement qu'en les tapant au clavier ou en les transférant depuis une autre application par copier/coller. Il se peut aussi que les résultats produits par un programme méritent d'être stockés en vue d'un usage ultérieur. La librairie Qt propose plusieurs classes qui facilitent l'utilisation de fichiers, c'est à dire le transfert d'informations entre disques et mémoire.

1 - La classe QFile

La classe QFile permet de mettre en relation le programme et les fichiers de données. C'est en effet à une instance de cette classe qu'il faut désigner le fichier concerné, et c'est ensuite en agissant sur cette variable que le programme pourra, indirectement, manipuler le fichier.

L'utilisation du type QFile implique la présence d'une directive #include "QFile.h"

Désignation du fichier

Initialiser une instance de QFile à l'aide d'une chaîne contenant le nom d'un fichier permet d'obtenir une variable associée à ce fichier :

```
1  QFile leFichier("essai.txt");
```

Si une variable de type QFile existe déjà, la fonction name() permet d'obtenir le nom du fichier auquel elle correspond :

```
2  QString nom = leFichier.name();
   //nom contient "essai.txt"
```

La fonction setName() permet de changer le fichier associé à une variable de type QFile :

```
3  leFichier.setName("unAutre.txt");
   //la variable leFichier est maintenant associée au fichier unAutre.txt
```

Notez bien que setName() ne permet en aucune façon de renommer un fichier. Il s'agit simplement d'utiliser la même variable de type QFile pour travailler sur un *autre* fichier, ce qui suppose que le fichier précédemment utilisé est en état d'être ainsi "abandonné".

Le "nom" utilisé pour désigner un fichier peut être un chemin d'accès (complet ou relatif). L'utilisation de la barre oblique '/' permet, dans ce cas, de décrire un chemin valide quel que soit le système d'exploitation utilisé :

```
4  leFichier.setName("data/mes_donnees.txt"); //correct, quel que soit l'OS
   //sous Windows, on peut aussi écrire : leFichier.setName("data\\mes_donnees.txt");
   //sous MacOs, on peut aussi écrire : leFichier.setName("data:mes_donnees.txt");
```

Renseignements disponibles

Dès lors qu'elle est associée à un nom de fichier, une variable de type QFile peut fournir un certain nombre d'informations concernant ce fichier.

La fonction exists() renvoie un booléen qui confirme (ou infirme...) l'existence du fichier associé à la variable QFile au titre de laquelle elle est invoquée :

```
5  QString message;
6  if(leFichier.exists())
7     message = leFichier.name() + " existe bel et bien";
8  else
9     message = "Il n'existe aucun fichier nommé " + leFichier.name();
```

La fonction exists() peut également être invoquée au titre de la classe QFile, à condition de lui passer comme argument le nom du fichier concerné :

```
10    bool ilEstLa = QFile::exists("monFichier.txt");
```

La fonction size() renvoie la taille (en octets) du fichier associé à la variable QFile au titre de laquelle elle est invoquée :

```
11 unsigned int tailleFichier = leFichier.size();
```

La fonction `atEnd()` renvoie `false` tant que le fichier contient des données non encore "extraites" :

```
12 while (! leFichier.atEnd())
13    {
14    //extraction des données (cf. la suite de cette Leçon)
15    }
```

Effacer un fichier

La classe `QFile` permet également d'effacer du disque un fichier devenu inutile (un fichier temporaire créé par le programme lui-même, par exemple).

Attention : l'effacement ainsi obtenu est définitif (pas de mise "à la poubelle" ou "à la corbeille").

C'est la fonction `remove()` qui assure ces basses œuvres, et elle peut être invoquée soit au titre d'une instance de `QFile` *(11)*, soit au titre de la classe elle-même *(12)*. Dans ce dernier cas, il faut évidemment qu'un argument lui désigne le `condamné` :

```
16 leFichier.remove();
17 QFile::remove("temp.txt");
```

Tournez sept fois votre compilateur dans votre bouche avant d'écrire un programme effaçant autre chose qu'un fichier temporaire que vous venez de créer vous-mêmes.

Ouvrir un fichier

Lorsqu'un programme manipule des fichiers, il s'intéresse généralement à leur contenu. Pour accéder, à ce contenu, il est nécessaire d'ouvrir le fichier.

> L'ouverture d'un fichier ne se traduit pas par l'apparition à l'écran d'une fenêtre contenant un texte correspondant au contenu de ce fichier. Du point de vue d'un programme, un fichier est "ouvert" lorsqu'il est prêt à participer à des transferts de données entre disque et mémoire.

C'est la fonction `open()` qui assure l'ouverture du fichier associé à la variable `QFile` au titre de laquelle elle est invoquée. Cette fonction exige un argument qui indique à quelles opérations le fichier doit ensuite se prêter. La valeur de cet argument est fixée en utilisant des constantes prédéfinies dans la librairie Qt :

Mode d'ouverture	Opérations autorisées par l'ouverture
`IO_ReadOnly`	Lecture du contenu du fichier
`IO_WriteOnly`	Ecriture dans le fichier, en écrasant son contenu antérieur (si le fichier n'existe pas, il est créé).
`IO_WriteOnly | IO_Append`	Ecriture dans le fichier, à la suite de son contenu antérieur

```
1 QFile lesDonnees("data.txt");
2 lesDonnees.open(IO_ReadOnly); //on va LIRE les données !
3 QFile resultats("produit.txt");
4 resultats.open(IO_WriteOnly); //on va remplacer le contenu du fichier
5 QFile journalDeBord("historique.txt");
6 journalDeBord.open(IO_WriteOnly | IO_Append); //on va écrire à la fin du fichier
```

Ces modes d'ouverture peuvent être complétés, à l'aide de la constante `IO_Translate`, d'une demande de traduction automatique de la représentation des marques de paragraphe.

> Certains fichiers représentent les sauts de paragraphe à l'aide de deux caractères consécutifs ("\r\n", c'est à dire 0x0D 0x0A) alors que d'autres se contentent d'un seul ('\n'). L'écriture de programmes gérant correctement tous les fichiers est facilitée par le mode `IO_Translate`, qui permet au programme de ne jamais voir apparaître les '\r' précédant un '\n'. Notez que l'utilisation de `IO_Translate` peut donc provoquer une divergence entre la taille du fichier (exprimée en octets) et la taille du texte qu'il contient (exprimée en caractères).

La traduction est demandée en combinant `IO_Translate` à l'un des modes d'utilisation du fichier à l'aide de l'opérateur "OU bitaire", noté `|`.

```
1 QFile leTexte("blabla.txt");
2 leTexte.open(IO_ReadOnly | IO_Translate);
```

L'usage des opérateurs bitaires est présenté en détails dans l'Annexe "Quand un octet est trop grand", qui vous permettra de comprendre pourquoi la conjonction de deux modes est obtenue à l'aide d'un OU et non, comme on pourrait s'y attendre, à l'aide d'un ET.

Erreurs d'ouverture

La fonction `open()` renvoie un booléen qui indique si le fichier a répondu favorablement à la demande d'ouverture.

Tout programme sérieux doit gérer les cas d'échec d'ouverture des fichiers qu'il utilise.

De nombreuses causes peuvent conduire à un tel échec, même si le fichier existe (l'utilisateur du programme peut, par exemple, ne pas avoir le droit d'accéder au fichier concerné).

Lorsqu'un programme se heurte à un échec d'ouverture d'un fichier nécessaire à son fonctionnement, il devrait, au minimum, prévenir l'utilisateur en indiquant clairement quel est le fichier incriminé.

La "simple" ouverture d'un fichier devrait donc donner lieu à des constructions du genre :

```
1  const QString signature = "monProgramme"; //le nom du programme
2  QString message;
3  QFile config("config.txt");
4  if(!config.exists())
5     {
6     message = "Le fichier " + config.name() + " n'existe pas";
7     QMessageBox::critical(0, signature, message);
8     }
9  else
10    {
11    bool resultat = config.open(IO_ReadOnly | IO_Translate);
12    if(! resultat)
13       {
14       message = "Impossible d'ouvrir le fichier " + config.name();
15       QMessageBox::critical(0, signature, message);
16       }
17    else
18       {
          //tout va bien, on peut essayer de lire le contenu du fichier
19       }
20    }
```

Les débutants rechignent fréquemment devant la perte de temps que représente, selon eux, la mise en place d'un tel dispositif. Ils s'en dispensent donc, et passent ensuite des heures à essayer de trouver une erreur imaginaire dans le traitement qu'ils ont programmé, pour finir par comprendre que, du fait d'une erreur d'ouverture d'un fichier, leur programme manque tout simplement des données nécessaires pour produire le résultat attendu.

Cette coupable désinvolture est malheureusement encouragée par le fait que, pour des raisons évidentes de gain de place et de lisibilité, les exemples figurant dans les livres ou les cours de programmation (même les meilleurs) ne répètent pas systématiquement les lignes de code assurant les mesures de sécurité élémentaires.

Un exemple de code n'est pas un programme complet et sérieux.

Autrement dit : faîtes ce que je dis, ne faîtes pas ce que je fais (ici).

Fermer un fichier

Une fois son utilisation terminée, un fichier peut être refermé en appelant la fonction `close()` au titre de la variable de type `QFile` qui lui est associée. La séquence d'instructions prenant place entre les lignes *18* et *19* du fragment de code précédent pourrait donc se terminer par :

```
config.close();
```

Une fois refermé, le fichier ne se prête plus à aucune opération de lecture ou d'écriture, et la variable de type `QFile` qui servait à le manipuler peut être en toute sécurité associée à un autre fichier (à l'aide de la fonction `setName()` présentée précédemment).

La disparition de la variable de type `QFile` (pour cause de fin du bloc dans lequel elle est définie, par exemple) assure également la fermeture du fichier correspondant, sans qu'il soit besoin d'appeler la fonction `close()`.

Bien entendu, avant de refermer un fichier, le programme aura vraisemblablement besoin de l'utiliser pour en lire le contenu ou pour y placer ses résultats. Dans la plupart des cas, toutefois, ces opérations ne se font pas en utilisant directement la variable de type `QFile` associée au fichier, mais en passant par l'intermédiaire d'une variable de type `QTextStream`.

2 - La classe `QTextStream`

Bien qu'elle soit ici décrite dans le contexte de l'accès à des fichiers de données, la classe `QTextStream` est capable de fournir les mêmes services dans d'autres contextes, ce qui justifie qu'elle ait une existence indépendante de la classe `QFile`.

La classe `QTextStream` peut ainsi servir à lire/écrire des données dans une simple `QString` ou à faire communiquer deux ordinateurs connectés par leur port série.

L'utilisation du type `QFile` implique la présence d'une directive `#include "QTextStream.h"`

Connexion à un fichier

L'initialisation d'une variable de type `QTextStream` avec l'adresse d'une instance de `QFile` permet d'obtenir la connexion du `QTextStream` au fichier associé au `QFile`. Les opérations de lecture ou d'écriture portant sur le `QTextStream` se traduiront donc par un transfert d'information depuis ou vers ce fichier.

```
1  QFile qFichier("unFichier.txt");
2  qFichier.open(IO_ReadOnly | IO_Translate);
3  QTextStream leFichier(& qFichier);
```

Lorsqu'une variable de type `QTextStream` est connectée à un fichier, cette connexion peut être rompue en appelant la fonction `unSetDevice()` :

```
4  leFichier.unSetDevice();
```

Un `QTextStream` peut être connecté à un nouveau fichier à l'aide de la fonction `setDevice()`, à laquelle il faut passer l'adresse d'un `QFile` associé au fichier concerné :

```
5  QFile unAutre("unAutreFichier.txt");
6  unAutre.open(IO_WriteOnly | IO_Translate);
7  leFichier.setDevice(& unAutre);
```

Ecrire

Lorsqu'un `QTextStream` est connecté à un fichier ouvert en écriture, l'opérateur d'insertion noté `<<`, permet de placer dans ce fichier la séquence de caractères représentant la valeur sur laquelle est appliqué l'opérateur. Le fragment de code suivant place donc dans le fichier nommé "unAutreFichier.txt" les caractères '0', '1', '2', '3', '4', '5', '6', '7', '8' et '9', séparés par des passages à la ligne :

```
8  int n;
9  for (n = 0 ; n < 10 ; n = n + 1)
10   leFichier << n << "\n";
```

Si vous avez scrupuleusement fait les exercices proposés dans le TD 4, l'opérateur d'insertion devrait vous paraître familier. La "magie noire" pratiquée en début de ce TD a en effet pour objectif de vous permettre d'écrire à l'écran exactement comme on écrit dans un fichier lorsqu'on utilise un `QTextStream`.

Tous les types prédéfinis (`int`, `double`, etc), ainsi que les classes Qt pour lesquelles cela à un sens (`QString`, par exemple) acceptent de voir leurs instances ainsi "injectées" dans un `QTextStream`. Pour ce qui est des classes que nous définirons nous-mêmes, il nous appartiendra de les doter de cette caractéristique (cf. Leçon 15).

Notez que les booléens `true` et `false`, lorsqu'ils sont insérés dans un `QTextStream`, sont respectivement représentés par les caractères '1' et '0', et non par les chaînes "true" et "false".

Lire

Lorsqu'un `QTextStream` est connecté à un fichier ouvert en lecture, l'opérateur d'extraction noté `>>`, permet de placer dans la variable sur laquelle il est appliqué une valeur construite à partir d'une suite de caractères lus dans le fichier.

> Il s'agit d'une simple opération de lecture : contrairement à ce que le terme "extraction" pourrait laisser croire, le contenu du fichier n'est nullement modifié.

La suite de caractères utilisée pour construire la valeur attribuée à la variable s'arrête dès qu'un caractère non interprétable dans ce contexte est rencontré. Ainsi, si un fichier contient

```
124.32abc
```

et qu'il est lu au moyen d'un `QTextStream` nommé `fichier`, on aura :

```
1 double x;
2 fichier >> x;          //x reçoit la valeur 124.32
3 QString texte;
4 fichier >> texte;      //texte reçoit la valeur "abc"
```

Comme le caractère '.' ne peut pas figurer dans la représentation d'une valeur entière, la lecture du même fichier donnera lieu à une interprétation différente si on écrit :

```
1 int a;
2 fichier >> a;          //a reçoit la valeur 124
3 QString texte;
4 fichier >> texte;      //texte reçoit la valeur ".32abc"
```

Les caractères dits "séparateurs" (espace, tabulation, saut de ligne...) sont considérés comme n'étant interprétables dans aucun contexte, c'est à dire qu'ils mettent fin à la suite de caractères utilisée pour construire la valeur lue, quel que soit le type de la variable dans laquelle cette valeur va être stockée. Ainsi, si notre fichier contient

```
abc 124
```

nous aurons :

```
1 QString texte;
2 fichier >> texte;      //texte reçoit la valeur "abc"
3 int a;
4 fichier >> a;          //a reçoit la valeur 124
```

Lorsqu'un fichier contient du texte "ordinaire" (ie. une suite de mots formant des phrases), sa lecture au moyen de l'opérateur d'extraction ne permet donc que d'obtenir les "mots" un par un, sans indication sur la nature du caractère qui les séparait (un espace ? un saut de ligne ?). Cette façon de lire le fichier ne convient donc pas à toutes les situations, et la classe `QTextStream` offre d'autres possibilités.

La fonction `readLine()` renvoie une `QString` qui contient l'équivalent d'un paragraphe, c'est à dire tous les caractères compris entre la position courante de lecture du fichier et le saut de ligne suivant.

> Le saut de ligne n'est pas présent en fin de `QString`. Cette façon de lire le fichier est très adaptée aux cas où la notion de "paragraphe" structure effectivement celui-ci, comme par exemple lorsqu'il s'agit d'une liste, avec un item par ligne.

La fonction `read()` renvoie une `QString` qui contient tous les caractères compris entre la position courante de lecture du fichier et la fin de celui-ci.

> Cette façon de lire convient donc très bien aux fichiers de taille modérée dans lesquels les paragraphes n'ont pas une importance fondamentale. Le texte d'un roman, par exemple, pourra tout à fait être lu de cette façon, qu'il s'agisse de l'afficher à l'écran où d'y rechercher des phénomènes particuliers.

La fonction `atEnd()` permet de détecter l'épuisement des données contenues dans le fichier au titre de laquelle elle est appelée. Cette fonction permet donc de lire intégralement un fichier dont on ignore la longueur. Le fragment de code suivant, par exemple, calcule la moyenne des valeurs contenues dans le fichier "données.txt", quel que soit le nombre de celles-ci :

```
1  QFile qFichier("données.txt");
2  qFichier.open(IO_ReadOnly | IO_Translate);
3  QTextStream leFichier(& qFichier);
4  int nbValeurs = 0;
5  double valeurLue;
6  while( !lefichier.atEnd()) //tant qu'il reste des données…
7     {
8     leFichier >> nbValeurs;
9     somme = somme + valeurLue;
10    ++nbValeurs;
11    }
12 double moyenne = somme / nbValeurs;
```

3 - Savoir quel fichier utiliser

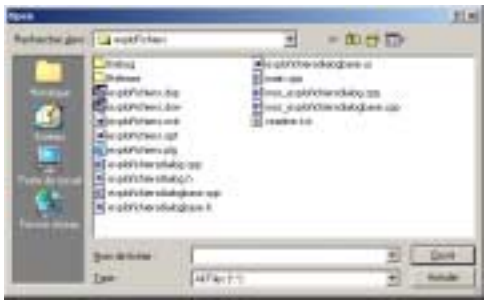
Dans bien des cas, c'est l'utilisateur du programme qui doit être en mesure d'indiquer quel fichier de données il faut utiliser, ou quel fichier doit être créé pour y stocker les résultats obtenus. La classe `QFileDialog` permet d'offrir à l'utilisateur un dialogue lui permettant d'exprimer son souhait.

L'utilisation du type `QFileDialog` implique la présence d'une directive
`#include "QFileDialog.h"`

Pour faire désigner un fichier de données, on utilisera la fonction `getOpenFileName()` :

```
1  QString chemin = QFileDialog::getOpenFileName();
```

Dans un environnement Windows, l'exécution de cette ligne de code fait apparaître la fenêtre représentée ci-contre, et l'exécution du programme est suspendue jusqu'à ce que l'utilisateur referme ce dialogue.



Si cette fermeture est déclenchée avec le bouton "Annuler", la `QString` renvoyée est vide. Si elle est obtenue à l'aide du bouton "Ouvrir"

La fonction `getSaveFileName()` fonctionne de façon analogue, mais le dialogue proposé est de type "Enregistrer sous...", ce qui permet à l'utilisateur de créer un nouveau fichier.

4 - Bon, c'est gentil tout ça, mais ça fait déjà 6 pages. Qu'est-ce que je dois vraiment en retenir ?

1) Pour manipuler un fichier (vérifier s'il existe, connaître sa taille, l'effacer, l'ouvrir...) on crée une instance de la classe `QFile` et on l'initialise avec le nom du fichier.
2) Tout fichier .h ou .cpp qui contient le mot `QFile` doit comporter une directive
   `#include "QFile.h"`
3) Pour manipuler le contenu d'un fichier, on crée généralement une instance de la classe `QTextStream` que l'on initialise avec le `QFile` associé au fichier.
4) Tout fichier .h ou .cpp qui contient le mot `QTextStream` doit comporter une directive
   `#include "QTextStream.h"`
5) Dès qu'un programme utilise des fichiers de données, les risques qu'un incident l'empêche de fonctionner normalement cessent d'être négligeables. Il doit donc être conçu de façon à réagir correctement en cas d'anomalie.
6) On écrit dans un `QTextStream` en utilisant l'opérateur d'insertion.
7) On lit un `QTextStream` soit en utilisant l'opérateur d'extraction, soit en utilisant les fonctions `readLine()` ou `read()`.