



Centre Informatique pour les **L**ettres
et les **S**ciences **H**umaines

Apprendre C++ avec Qt : Leçon 7

Utiliser des fichiers de données

1 - La classe QFileDialog	2
2 - La classe QFile.....	2
Désignation du fichier	2
Informations disponibles	3
Effacer un fichier.....	3
Ouvrir un fichier	3
Erreurs d'ouverture	4
Gérer la position courante	5
Fermer un fichier	5
3 - La classe QTextStream	6
Connexion à un fichier	6
Ecrire	6
Lire	7
4 - En pratique.....	8
Lecture d'un fichier comportant du texte et des données numériques.....	8
Transmission d'un fichier à une fonction.....	9
Recherche dans le code source d'une erreur qui est dans le fichier de données.....	9
5 - Bon, c'est gentil tout ça, mais ça fait déjà 8 pages. Qu'est-ce que je dois vraiment en retenir ?.....	10

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 QFileDialog

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 moyen d'exprimer son choix.

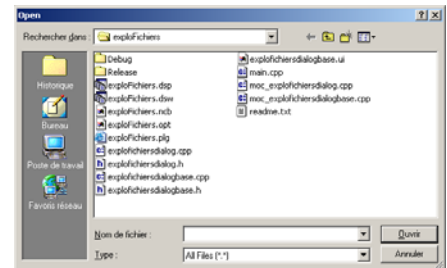
L'utilisation du type QFileDialog implique la présence d'une directive `#include "QFileDialog.h"`

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

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

Sous 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 mette fin au dialogue.

Si le dialogue s'achève par un clic sur [Ouvrir], la QString renvoyée contient le chemin complet d'accès au fichier désigné. Dans le cas contraire, elle est vide.



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.

2 - 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 désignant un fichier permet d'obtenir une variable associée à ce fichier :

```
2 QFile leFichier(chemin);
```

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

```
3 leFichier.setName("unAutre.txt");
```

Lors de l'initialisation du QFile ou de l'appel de setName(), le fichier peut être désigné par un chemin d'accès (complet ou relatif). L'utilisation de la barre oblique '/' permet 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");
```

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".

Inversement, la fonction `name()` fournit le nom du fichier auquel un QFile est associé :

```
5 QString nom = leFichier.name();
```

Informations disponibles

Dès lors qu'elle est associée à un nom de fichier, une variable de type `QFile` peut fournir un certain nombre de renseignements 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 :

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

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 :

```
9 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 :

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

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** :

```
11 leFichier.remove();
12 QFile::remove("temp.txt");
```

Tournez sept fois votre code source dans votre compilateur avant de commettre un programme effaçant autre chose qu'un fichier temporaire qu'il vient lui-même de créer...

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
<code>IO_ReadOnly</code>	Lecture du contenu du fichier.
<code>IO_WriteOnly</code>	Écriture dans le fichier, en écrasant son contenu antérieur. Si le fichier n'existe pas encore, il est créé.
<code>IO_WriteOnly IO_Append</code>	Écriture dans le fichier, à la suite de son contenu antérieur Si le fichier n'existe pas encore, l'ouverture échoue.

Typiquement, l'utilisation d'un fichier de données donnera donc lieu à des séquences telles que

```
1 QFile lesDonnees("data.txt");
2 lesDonnees.open(IO_ReadOnly); //on va LIRE les données !
3
4 QFile resultats("résultats.txt");
5 resultats.open(IO_WriteOnly); //on va remplacer le contenu du fichier
6
7 QFile journalDeBord("historique.txt");
8 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 contenant du texte est facilitée par le mode `IO_Translate`, qui permet de ne jamais voir apparaître les "`\r`" qui, dans certains fichiers, précèdent les "`\n`".

Remarquez que la combinaison de plusieurs caractéristiques d'ouverture du fichier est obtenue à l'aide de l'opérateur "OU bitaire" :

```
7 QFile leTexte("blabla.txt");
8 leTexte.open(IO_ReadOnly | IO_Append | IO_Translate);
```

L'usage des opérateurs bitaires est présenté en détails dans la [Leçon 24](#), 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 indiquant si tout s'est bien passé.

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 ne parvient pas à ouvrir un fichier, il devrait, au minimum, prévenir l'utilisateur en indiquant clairement quel est le fichier incriminé.

Une fonction ouvrant un fichier devrait donc toujours contenir une construction 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     return;
9 }
10 if(!config.open(IO_ReadOnly | IO_Translate))
11 {
12     message = "Impossible d'ouvrir le fichier " + config.name();
13     QMessageBox::critical(0, signature, message);
14     return;
15 }
//tout va bien, on peut essayer de lire le contenu du fichier
```

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'un échec 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 : faites ce que je dis, ne faites pas ce que je fais (ici).

Gérer la position courante

Qu'il s'agisse de lecture ou d'écriture, les opérations effectuées sur un fichier modifient automatiquement la "position courante", c'est à dire l'endroit où débutera l'opération suivante.

En clair : si vous écrivez deux fois de suite dans le même fichier, la seconde opération n'efface pas l'effet de la première mais écrit *à la suite* de ce qu'a écrit la première. De même, deux lectures consécutives ne donneront pas la même information, mais deux informations qui, dans le fichier, sont à la suite l'une de l'autre.

La fonction `atEnd()` renvoie `true` lorsque la position courante a atteint la fin du fichier. En d'autres termes, elle renvoie `false` tant que le fichier contient des données non encore lues, et c'est donc généralement cette fonction qui est utilisée pour organiser la lecture complète d'un fichier de données :

```
16 while (! config.atEnd())
17 {
18     //extraction des données (cf. la suite de cette Leçon)
    }
```

La fonction `at()` renvoie la position courante, exprimée en nombre d'octets depuis le début du fichier.

La fonction `at()` peut aussi être utilisée pour modifier la position courante : il suffit pour cela de lui passer la position désirée, exprimée en nombre d'octets depuis le début du fichier. Utilisée ainsi, la fonction `at()` renvoie un booléen qui indique si le déplacement demandé a bien été effectué.

Deux cas particuliers sont à signaler :

- Si la position d'écriture est déplacée au-delà du dernier caractère actuel, la fonction `at()` insère automatiquement des octets nuls dans le *no man's land* ainsi créé, mais ces octets ne sont effectivement ajoutés au fichier que si une opération d'écriture est ensuite effectuée.
- Lorsqu'un fichier a été ouvert en mode `IO_WriteOnly` | `IO_Append`, toutes les écritures s'effectuent en fin de fichier, même si la position courante a été préalablement déplacée.

Le déplacement explicite de la position courante est une manœuvre qui doit être effectuée avec circonspection, surtout lorsque le fichier concerné est ouvert en écriture.

Une erreur de programmation peut, dans ce cas, se traduire par un écrasement de données...

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 17 et 18 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 en toute sécurité être 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`.

3 - 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 `QTextStream` 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 leFichier ("unFichier.txt");
2 leFichier.open(IO_WriteOnly | IO_Translate);
3 QTextStream destination(& leFichier);
```

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

```
4 destination.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 destination.setDevice(& unAutre);
```

Ecrire

Lorsqu'un `QTextStream` est connecté à un fichier ouvert en écriture, l'opérateur d'insertion, 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', séparés par des passages à la ligne :

```
8 int n;
9 for (n = 0 ; n < 6 ; n = n + 1)
10     destination << n << "\n";
```

Si vous avez fait les exercices proposés dans le TD 4, l'opérateur `<<` devrait vous paraître familier. La "magie noire" pratiquée dans ce TD a en effet pour but 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 11](#)).

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".

Il faut aussi savoir que l'insertion de données dans un `QTextStream` ne se traduit pas toujours *immédiatement* par l'écriture de ces données dans le fichier. Pour améliorer la vitesse d'exécution, le système garde en effet les données en mémoire en attendant d'en avoir suffisamment pour justifier un accès au disque concerné (cette opération est relativement longue et sa durée n'est pas directement proportionnelle au volume de données à transférer). Si un programme doit faire en sorte que les données en attente soit effectivement écrites sur le disque à un moment donné, il peut faire appel à la fonction `QFile::flush()` :

```
leFichier.flush();
```

Remarquez que la fonction `flush()` doit être invoquée au titre du `QFile`, et non au titre du `QTextStream` qui lui est associé. Bien entendu, la fermeture du fichier (par `close()` ou par disparition de la variable de type `QFile` utilisée) garantit aussi que toutes les données en attente sont physiquement écrites sur le disque.

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 `source`, on aura :

```
1 double x;
2 source >> x;           //x reçoit la valeur 124.32
3 QString texte;
4 source >> 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 source >> a;           //a reçoit la valeur 124
3 QString texte;
4 source >> 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

```
TOTO 999
```

nous aurons :

```
1 QString texte;
2 source >> texte;       //texte reçoit la valeur "TOTO"
3 int a;
4 source >> a;           //a reçoit la valeur 999
```

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 sépare (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 tous les caractères compris entre la position de lecture courante et le saut de ligne suivant.

Notez que le saut de ligne ne figure pas à la fin de la `QString`. Cette façon de lire est très adaptée aux cas des fichiers composés de "paragraphes", comme par exemple lorsqu'il s'agit d'une liste comportant 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 relativement modérée et 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 ou d'y rechercher des phénomènes particuliers.

La fonction `atEnd()` (analogue à celle disponible dans la classe `QFile`) permet de détecter l'épuisement des données rendues accessibles par le `QTextStream`. 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 leFichier("données.txt");
2  leFichier.open(IO_ReadOnly | IO_Translate);
3  QTextStream source(&leFichier);
4  int nbValeurs = 0;
5  double valeurLue;
6  while( !source.atEnd()) //tant qu'il reste des données...
7  {
8      source >> valeurLue;
9      somme = somme + valeurLue;
10     ++nbValeurs;
11 }
12 double moyenne = somme / nbValeurs;
```

4 - En pratique

Si les principes en sont simples (il s'agit essentiellement de mettre en place un `QFile` et un `QTextStream`), les fonctions de gestion des fichiers restent souvent la partie du programme dont la mise au point s'avère la plus laborieuse. Trois difficultés méritent d'être signalées.

Lecture d'un fichier comportant du texte et des données numériques

Comme nous l'avons vu, l'opérateur d'extraction considère que la donnée qu'il est chargé de lire s'arrête au premier caractère "non interprétable" étant donné le type de valeur qu'il cherche à lire. Ce caractère "non interprétable" n'est pas extrait du fichier et reste disponible pour l'opération de lecture suivante.

Dans notre exemple où le fichier contient

```
124.32abc
```

sa lecture par

```
double x;
fichier >> x;           //x reçoit la valeur 124.32
QString texte;
fichier >> texte;       //texte reçoit la valeur "abc"
```

fait intervenir deux fois le 7^e caractère du fichier : il indique dans un premier temps que la séquence de caractères décrivant la valeur décimale est terminée, et il est ensuite le premier caractère placé dans la `QString`.

Comme l'opérateur d'extraction considère que les "séparateurs" ne sont jamais interprétables, son usage est peu pratique lorsqu'il s'agit de lire un fragment de texte susceptible de contenir des espaces (un chemin d'accès à un fichier du genre "c:/Mes Documents/...", par exemple, ou un nom propre comme "du Pont"). Dans ce genre de cas, on préfère souvent extraire le texte en appelant la fonction `readLine()` qui, comme son nom l'indique, lit jusqu'à la fin de la ligne. Le défaut de `readLine()` est qu'elle renvoie une `QString` qui, dans le cas des données numériques, doit être convertie pour devenir utilisable. Face à un fichier dont certaines lignes contiennent du "vrai" texte alors que d'autres doivent être interprétées comme des valeurs numériques, il est donc tentant d'utiliser `readline()` pour les unes et l'opérateur d'extraction pour les autres. Appliquée naïvement, cette méthode ne fonctionne malheureusement pas.

Considérons un fichier dont le contenu est le suivant :

```
178.56
Aix en Provence
```

Essayer d'extraire les données à l'aide de la séquence

```
double valeurNumerique;
source >> valeurNumerique;
QString nomDeLaVille = source.readLine(); //nomDeLaVille reste vide !
```

ne donne pas le résultat attendu. Le "caractère non interprétable" qui met fin à l'extraction de la valeurNumérique est en effet le caractère de fin de ligne. Il reste donc disponible pour l'opération de lecture suivante qui, dans ce cas, utilise `readLine()`. Puisque les données restant à lire commencent par un caractère de fin de ligne, cette fonction renvoie évidemment une chaîne vide... Une première solution consiste à utiliser systématiquement `readLine()` :

```
QString ligne = source.readLine();           //lecture de "178.56"
double valeurNumerique = ligne.toDouble();   //calcul de la valeur 178.56
QString nomDeLaVille = source.readLine();    //lecture de "Aix en Provence"
```

Une autre méthode, parfois plus agréable, est d'utiliser la fonction `skipWhiteSpace()` :

```
double valeurNumerique;
source >> valeurNumerique;
source.skipWhiteSpace();                     //extraction du passage à la ligne
QString nomDeLaVille = source.readLine();    //lecture de "Aix en Provence"
```

Transmission d'un fichier à une fonction

Du fait même de leur rôle, les instances des classes `QFile` et `QTextStream` doivent contenir des pointeurs leur permettant d'accéder à l'objet auquel elles sont associées (un fichier dans le cas des `QFile`, un `QFile` dans le cas des `QTextStream`). Lorsque la valeur d'une instance est attribuée à une autre instance, ces pointeurs sont copiés, et les deux instances se retrouvent associées au même objet.

Outre les incohérences que ceci risque d'entraîner (si le même objet est manipulé tantôt via une instance, tantôt via l'autre), cette situation crée un problème lors de la disparition des instances qui ont été copiées l'une sur l'autre. La première d'entre elles qui disparaît provoque en effet un "grand nettoyage" qui rend invalides les pointeurs contenus dans l'autre instance. Même si celle-ci n'est pas utilisée, sa disparition provoquera inmanquablement une erreur d'exécution lorsqu'un second "grand nettoyage" sera entrepris sur des objets qui, du fait du premier, n'existent plus. Les deux instructions suivantes, par exemple, suffisent à générer une erreur d'exécution (ie. un "plantage" du programme) :

```
QFile un;
QFile deux = un; //HORREUR ! Copie d'un QFile !
```

On peut s'interroger sur la logique qui conduirait à placer de telles instructions dans un programme, mais il existe bien d'autres situations conduisant à copier la valeur d'un objet dans un autre. Une de ces situations est l'initialisation d'un paramètre avec la valeur transmise par la fonction appelante. Par conséquent :

Un paramètre ne doit jamais être de type `QFile` ou `QTextStream`. Utilisez un pointeur ou une référence pour donner aux fonctions qui en ont besoin accès aux objets de ce type.

Comme vous le savez, la transmission de l'adresse d'un objet ne nécessite aucune recopie de cet objet, la fonction travaillant sur l'original (auquel elle accède en déréférençant le pointeur qui contient cette adresse). De façon similaire, un paramètre "référence" sera associé à l'objet original, sans qu'aucune copie ne soit effectuée.

Recherche dans le code source d'une erreur qui est dans le fichier de données

Si la mise au point du code gérant les fichiers de données est souvent laborieuse, c'est en particulier parce que les programmeurs débutants perdent beaucoup de temps à essayer de corriger des programmes parfaitement corrects.

Avant de tester votre programme, vérifiez que les fichiers de données qu'il va utiliser sont bien structurés comme prévu. Une simple ligne vide inattendue peut avoir pour effet de décaler les données, ce qui donne parfois l'impression (très décourageante) que tout le programme est inepte et doit être ré-écrit en partant de zéro.

Dans un programme idéal, chaque acquisition de données devrait immédiatement faire l'objet de vérifications de cohérence donnant lieu, en cas de besoin, à des messages de diagnostic circonstanciés. En réalité, la mise en place de ces diagnostics exige un effort qui ne se justifie que pour des programmes qui vont rencontrer d'autres utilisateurs que leur auteur, et il est vraisemblable que vos premiers projets devront s'en passer.

5 - Bon, c'est gentil tout ça, mais ça fait déjà 8 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()`.
- 8) L'opérateur d'extraction n'extraie pas le caractère qui suit la fin de la donnée lue. Si ce caractère est un saut de ligne et que la lecture suivante est faite par `readLine()`, cette fonction va interpréter ce saut de ligne comme la fin des données qu'elle doit lire et renvoyer une chaîne vide. Il faut, dans ce cas, utiliser `skipWhiteSpace()` pour franchir l'obstacle.
- 9) Aucun paramètre ne doit être de type `QFile` ou `QTextStream`. Utilisez des pointeurs ou des références.