6. Gestion des fichiers sous Qt

6.1. Introduction

La gestion des fichiers en Qt repose principalement sur deux classes : **QFile** et **QIODevice**. Il est important de distinguer deux types de fichiers :

- **Les fichiers texte** : Ils contiennent des lignes de texte et sont souvent utilisés pour stocker des informations lisibles par l'utilisateur ou des fichiers de configuration.
- **Les fichiers binaires** : Contrairement aux fichiers texte, ils contiennent des données brutes. Ils sont plus compacts et efficaces, mais nécessitent un logiciel spécifique pour être interprétés correctement.

Avant d'utiliser un fichier, il est nécessaire de l'ouvrir. En effet, l'accès aux fichiers est géré par le système d'exploitation, et l'application doit y accéder de manière contrôlée. L'ouverture permet ainsi d'établir une connexion entre l'application et le fichier sur le disque, ce qui permet au système d'exploitation :

- D'accéder aux données contenues dans le fichier pour les lire ou les modifier sans interférence d'autres programmes.
- D'allouer des ressources système telles que la mémoire et des descripteurs de fichiers, nécessaires pour interagir avec le fichier.

Lors de l'ouverture, il est indispensable de spécifier le mode d'accès (lecture, écriture, ajout, etc.).

Les opérations suivantes se déroulent lors de l'ouverture d'un fichier :

- **Vérification de l'existence du fichier** : Le système d'exploitation s'assure que le fichier existe (en cas de lecture) ou qu'il peut être créé (en cas d'écriture).
- **Attribution d'un descripteur de fichier** : Il s'agit d'un identifiant unique qui représente le fichier ouvert au niveau du système, utilisé pour toutes les opérations ultérieures (lecture, écriture).
- **Allocation de ressources système** : Mémoire et autres ressources nécessaires pour la gestion de l'accès au fichier.

Si l'ouverture échoue (fichier non trouvé, permissions insuffisantes, etc.), une erreur est renvoyée et une gestion spécifique doit être mise en place.

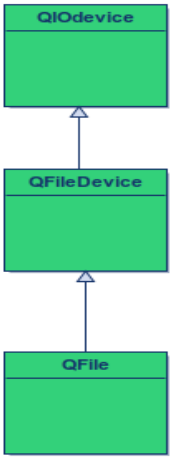
Lorsque les opérations sur le fichier sont terminées, il est crucial de fermer le fichier pour plusieurs raisons :

- **Libération des ressources système** : Chaque fichier ouvert utilise des ressources, comme des descripteurs de fichier. Si le fichier n'est pas fermé, ces ressources ne sont pas libérées, ce qui peut entraîner une surcharge et limiter le nombre de fichiers ouverts simultanément. La fermeture libère ces ressources pour d'autres applications.
- **Assurer la bonne écriture des données** : Lors de l'écriture, les données ne sont pas toujours immédiatement stockées sur le disque, mais peuvent être mises en cache. La fermeture du fichier garantit que le cache est vidé et que toutes les données sont correctement écrites sur le disque.
- **Préservation de l'intégrité des données** : Fermer le fichier correctement permet d'éviter la corruption potentielle qui pourrait survenir si le fichier reste ouvert et que l'application plante ou est interrompue.

Ainsi, la fermeture d'un fichier est indispensable pour libérer les ressources, garantir l'intégrité des données et éviter les fuites de la mémoire.

6.2. Classes et concepts en Qt

6.2.1. Hiérarchie des classes



La classe `QFile` est une classe de haut niveau qui facilite la manipulation des fichiers avec Qt. Elle hérite de `QFileDevice` qui est une classe intermédiaire, elle ajoute des fonctionnalités spécifiques aux fichiers, mais au niveau bas : manipulation des descripteurs, etc. Elle hérite elle-même de `QIODevice` qui est la base pour toute gestion d'entrées/sorties.

Cette hiérarchie permet une gestion modulaire et cohérente des fichiers tout en maximisant la réutilisabilité des classes et fonctionnalités dans différents types d'applications et de périphériques d'entrées/sorties.

6.2.2. Fonctionnalités apportées par la classe QIODevice

QIODevice est la classe de base dans Qt pour tous les types d'entrées/sorties (I/O). Elle fournit une interface commune pour les classes qui effectuent des opérations de lecture/écriture sur différents types de périphériques, fichiers, sockets, buffers, etc.

Les principales fonctionnalités offertes par **QIODevice** incluent :

- La gestion de l'ouverture et la fermeture des périphériques.
- Les modes d'accès (lecture seule, écriture seule, lecture/écriture, etc.).
- La lecture et l'écriture de données de manière générique.
- La gestion de la position courante dans un flux de données (seek, tell, etc.).

6.2.3. Fonctionnalités apportées par la classe QFileDevice

QFileDevice est une classe intermédiaire qui hérite de **QIODevice** et ajoute des fonctionnalités spécifiques aux fichiers. Elle introduit des méthodes et fonctionnalités pour interagir avec un fichier sur le système de fichiers, au niveau du système d'exploitation, mais sans fournir directement les moyens de spécifier un nom de fichier ou d'ouvrir un fichier particulier.

QFileDevice représente le modèle bas niveau d'un fichier et gère des aspects tels que :

- Le verrouillage des fichiers (file locking).
- La gestion de types d'erreurs spécifiques aux fichiers.
- L'interface pour des opérations de lecture et d'écriture basées sur les descripteurs de fichier au niveau du système.

Par exemple, certaines méthodes définies dans **QFileDevice** incluent :

- `flush()`: Cette méthode force l'écriture des données mises en cache sur le disque, ce qui est important pour assurer que toutes les modifications sont effectivement enregistrées.
- `map()`: Cette méthode permet de mapper un fichier en mémoire, ce qui est une technique avancée permettant d'accéder directement à une partie du fichier sans le charger entièrement en mémoire.

6.2.4. La classe QFile

QFile est une classe plus haut niveau qui hérite de **QFileDevice**. Elle encapsule des fonctionnalités spécifiques pour l'accès et la gestion des fichiers sur le système de fichiers. Contrairement à **QFileDevice**, **QFile** vous permet de manipuler un fichier via un chemin spécifié. **QFile** fournit des méthodes pour :

- Ouvrir un fichier à partir de son nom.
- Lire et écrire des données dans le fichier (en utilisant les méthodes de **QIODevice**).
- Gérer des flux de texte ou binaires.

Le constructeur de la classe permet de déterminer le nom du fichier qui va être traité.

6.3. Les principales méthodes pour l'accès aux fichiers

- **open()** : Ouvre le fichier avec un mode défini (lecture, écriture, etc.). Le mode est précisé à l'aide de constantes définies dans la classe QIOdevice comme `QIODevice::ReadOnly`, `QIODevice::WriteOnly`, `QIODevice::Append`. Éventuellement le mode d'ouverture combiné éventuellement avec la constante QIODevice::Text pour préciser que le fichier contient du texte. Pour les fichiers binaires, il n'est pas nécessaire de le préciser.

  La méthode retourne `true` en cas de succès, sinon `false`. Dans ce dernier cas, un traitement d'erreur peut être mis en place.

- **read() / readAll()** : Lit les données d'un fichier. `read()` permet de lire un nombre donné d'octets, tandis que `readAll()` lit tout le fichier.

- **readLine()** : Lit une ligne dans un fichier texte jusqu'au retour chariot.

- **write()** : Écrit des données dans un fichier. L'écriture peut être faite en mode texte ou binaire.

- **AtEnd() :** retourne `true` si la fin de fichier est atteinte.

Voici deux exemples permettant de lire le contenu d'un fichier.

Lecture d'un fichier texte

```
#include <QFile>
#include <QIODevice>
#include <QDebug>

int main() {
    QFile file("exemple.txt");

    // Ouvrir le fichier en mode lecture seule
    if (file.open(QIODevice::ReadOnly | QIODevice::Text)) {
        // Lire le fichier ligne par ligne
        while (!file.atEnd()) {
               // Lire une ligne sous forme de QByteArray
            QByteArray line = file.readLine();
               // Afficher la ligne sans les espaces en fin de ligne
            qDebug() << line.trimmed();
        }
        file.close();  // Fermer le fichier après lecture
    } else {
        qDebug() << "Impossible d'ouvrir le fichier.";
    }

    return 0;
}
```

Pour un fichier binaire, le traitement reste similaire :

Lecture d'un fichier binaire

```
#include <QFile>
#include <QIODevice>
#include <QDebug>

int main() {
    QFile file("exemple.bin");

    // Ouvrir le fichier en mode lecture seule (binaire)
    if (file.open(QIODevice::ReadOnly)) {
        // Lire le fichier par blocs de 1024 octets
        const int bufferSize = 1024; // Taille du bloc à lire
        while (!file.atEnd()) {
            QByteArray buffer = file.read(bufferSize);  // Lire un bloc de données binaires
            qDebug() << "Bloc lu de taille:" << buffer.size();

            // Afficher les données en hexadécimal pour plus de lisibilité
            qDebug() << buffer.toHex();
        }
        file.close();  // Fermer le fichier après lecture
    } else {
        qDebug() << "Impossible d'ouvrir le fichier.";
    }

    return 0;
}
```

6.4. Manipulation des fichiers avec les flux de données

6.4.1. Introduction

Pour faciliter la lecture et l'écriture dans les fichiers ou d'autres périphériques, Qt propose des classes spécialisées pour traiter les flux de données. Les deux principales classes sont :

- **QTextStream** : Utilisé pour lire et écrire du texte formaté, comme du texte UTF-8 ou ASCII, dans des fichiers ou des flux. Cette classe simplifie la gestion des fichiers texte en proposant des fonctionnalités pour manipuler les chaînes de caractères et formater les données.
- **QDataStream** : Utilisé pour la lecture et l'écriture de données binaires. Il permet de lire et d'écrire des types de données primitifs comme `int`, `float`, `double`, ainsi que des objets Qt comme `QString`, `QDate`, ou `QList`.

6.4.2. QTextStream

La classe **QTextStream** est conçue spécifiquement pour manipuler des données texte. Elle convertit automatiquement les caractères selon l'encodage défini, par défaut, UTF-8. Elle permet de lire ou d'écrire des fichiers texte ligne par ligne, mot par mot, et peut être configurée pour utiliser différents encodages, comme UTF-8, UTF-16, ou ISO 8859-1. La syntaxe de **QTextStream** est similaire à celle des flux C++ standard, comme `std::cout` ou `std::cin`, avec la possibilité de formater les données.

Écriture dans un fichier texte en utilisant un QTextStream

```
#include <QFile>
#include <QTextStream>
#include <QDebug>

int main() {
    QFile file("exemple.txt");

    if (file.open(QIODevice::WriteOnly | QIODevice::Text)) {
        QTextStream out(&file);
        out << "Hello, world!\n";
        out << "Voici un exemple d'écriture dans un fichier texte.\n";
        file.close();
    } else {
        qDebug() << "Impossible d'ouvrir le fichier en écriture.";
    }

    return 0;
}
```

Lecture d'un fichier texte, ligne par ligne, en utilisant un QTextStream

```
#include <QFile>
#include <QTextStream>
#include <QDebug>

int main() {
    QFile file("example.txt");

    if (file.open(QIODevice::ReadOnly | QIODevice::Text)) {
        QTextStream in(&file);
        while (!in.atEnd()) {
            QString line = in.readLine();
            qDebug() << line;
        }
        file.close();
    } else {
        qDebug() << "Impossible d'ouvrir le fichier en lecture.";
    }

    return 0;
}
```

Résumé

QTextStream permet de formater les données, par exemple en réglant la précision des nombres à virgule flottante ou en modifiant le comportement d'affichage des chaînes de caractères. Par défaut, **QTextStream** utilise UTF-8, mais il est possible de changer l'encodage avec la méthode `setEncoding()`. Quelques valeurs possibles de paramètre pour cette fonction, `QStringConverter::Utf8`, `QstringConverter::Utf16`, `QstringConverter::Latin1`, `QstringConverter::System`...

6.4.3. `QDataStream`

Contrairement à **QTextStream**, **QDataStream** est utilisé pour écrire des données dans leur format binaire natif, sans conversion. Il gère automatiquement des aspects comme l'indianité, ordre des octets, et la taille des types de données, ce qui garantit la portabilité des fichiers entre différentes architectures. La gestion des problèmes d'indianité par exemple peut être changée en appelant `setByteOrder()` :

- `QDataStream::LittleEndian` : Octet de poids faible en premier
- `QDataStream::BigEndian` : Octet de poids fort en premier (valeur par défaut)

Il est possible de spécifier une version du flux pour garantir la compatibilité des fichiers entre différentes versions d'une application avec la méthode `setVersion()`.

Écriture de données binaire en utilisant un QTextStream

```
#include <QFile>
#include <QDataStream>
#include <QDebug>

int main() {
    QFile file("example.dat");

    if (file.open(QIODevice::WriteOnly)) {
        QDataStream out(&file);
        out << qint32(42);            // Écrire un entier 32 bits
        out << QString("Bonjour");    // Écrire une chaîne de caractères
        file.close();
    } else {
        qDebug() << "Impossible d'ouvrir le fichier en écriture.";
    }

    return 0;
}
```

Écriture de données binaire en utilisant un QTextStream

```
#include <QFile>
#include <QDataStream>
#include <QDebug>

int main() {
    QFile file("example.dat");

    if (file.open(QIODevice::ReadOnly)) {
        QDataStream in(&file);
        qint32 number;
        QString text;

        in >> number;   // Lire un entier 32 bits
        in >> text;     // Lire une chaîne de caractères

        qDebug() << "Nombre:" << number;
        qDebug() << "Texte:" << text;
        file.close();
    } else {
        qDebug() << "Impossible d'ouvrir le fichier en lecture.";
    }

    return 0;
}
```

Résumé

QDataStream ne traite pas d'encodage de texte, mais gère la portabilité binaire (indianité, taille des types de données). Il est principalement utilisé pour stocker et restaurer des données binaires, comme pour sauvegarder l'état d'une application ou échanger des données entre systèmes.

6.5. Autres classes associées à la gestion de fichiers

6.5.1. `QFileDialog`

`QFileDialog` est une classe de Qt qui fournit une interface graphique pour la sélection de fichiers ou de répertoires. Elle permet à l'utilisateur d'ouvrir ou d'enregistrer des fichiers en utilisant une boîte de dialogue standard, couramment utilisée dans les applications. La classe `QFileDialog` peut être utilisée pour diverses tâches :

- **Ouvrir un fichier** : Sélectionner un fichier existant pour le lire ou le modifier.
- **Enregistrer un fichier** : Choisir un emplacement pour enregistrer un fichier.
- **Sélectionner un répertoire** : Choisir un dossier sans sélectionner un fichier en particulier.

Ouvrir un fichier

L'une des utilisations les plus courantes de `QFileDialog` est l'ouverture de fichiers. Voici un exemple d'utilisation pour ouvrir un fichier texte :

Boîte de dialogue pour choisir un fichier pour une lecture

```
QString fileName = QFileDialog::getOpenFileName(this,
    tr("Ouvrir un fichier"),
    "/home",
    tr("Fichiers texte (*.txt);;Tous les fichiers (*)"));
if (!fileName.isEmpty()) {
    QFile file(fileName);
    if (file.open(QIODevice::ReadOnly | QIODevice::Text)) {
        QTextStream in(&file);
        // Lecture du contenu du fichier
    }
}
```

Dans cet exemple, `getOpenFileName()`, une méthode statique de la classe, ouvre une boîte de dialogue pour sélectionner un fichier. Le troisième argument, `"/home",` spécifie le répertoire initial affiché, et le quatrième argument est un filtre de type de fichiers, ici, les fichiers `.txt` et tous les fichiers.

Enregistrer un fichier

Pour enregistrer un fichier, la méthode `getSaveFileName()` peut-être utilisée, elle permet de spécifier un nom de fichier et un emplacement. Voici un exemple :

Boîte de dialogue pour enregistrer un fichier

```
QString fileName = QFileDialog::getSaveFileName(this,
    tr("Enregistrer un fichier"),
    "/home/untitled.txt",
    tr("Fichiers texte (*.txt);;Tous les fichiers (*)"));
if (!fileName.isEmpty()) {
    QFile file(fileName);
    if (file.open(QIODevice::WriteOnly | QIODevice::Text)) {
        QTextStream out(&file);
        out << "Contenu à sauvegarder";
    }
}
```

Sélectionner un répertoire

Parfois, il est nécessaire de sélectionner un dossier plutôt qu'un fichier. Cela peut être fait avec `getExistingDirectory()` :

Boîte de dialogue pour sélectionner un répertoire

```
QString dir = QFileDialog::getExistingDirectory(this,
    tr("Choisir un répertoire"),
    "/home",
    QFileDialog::ShowDirsOnly | QFileDialog::DontResolveSymlinks);
if (!dir.isEmpty()) {
    // Utiliser le répertoire sélectionné
}
```

Le drapeau `QFileDialog::ShowDirsOnly` permet d'afficher uniquement les dossiers. Le drapeau `QFileDialog::DontResolveSymlinks` empêche la résolution des liens symboliques, ce qui peut être utile dans certains cas.

Personnalisation de QFileDialog

Il est possible de personnaliser la boîte de dialogue en modifiant son apparence ou en ajoutant des fonctionnalités spécifiques. En utilisant un objet `QFileDialog` au lieu des méthodes statiques mentionnées ci-dessus, le contrôle est plus important :

Boîte de dialogue pour sélectionner un répertoire

```
QFileDialog dialog(this);
dialog.setFileMode(QFileDialog::ExistingFiles);  // Permet de sélectionner plusieurs fichiers
dialog.setNameFilter(tr("Images (*.png *.xpm *.jpg)"));  // Filtre pour afficher uniquement les
images
dialog.setViewMode(QFileDialog::Detail);  // Affichage détaillé

if (dialog.exec()) {
    QStringList fileNames = dialog.selectedFiles();  // Liste des fichiers sélectionnés
    foreach (QString fileName, fileNames) {
        qDebug() << fileName;
    }
}
```

Avec ce code, on peut configurer la boîte de dialogue pour permettre la sélection de plusieurs fichiers (`ExistingFiles`) et restreindre l'affichage à certains types de fichiers (`NameFilter`). Ici, il est possible de sélectionner plusieurs fichiers.

`QFileDialog` propose plusieurs options pour personnaliser le comportement de la boîte de dialogue :

- **QFileDialog::ShowDirsOnly** : Affiche uniquement les répertoires.
- **QFileDialog::DontResolveSymlinks** : Ne pas résoudre les liens symboliques.
- **QFileDialog::ReadOnly** : La boîte de dialogue est en mode lecture seule.
- **QFileDialog::HideNameFilterDetails** : Masque les détails du filtre de fichiers dans la boîte de dialogue.

6.5.2. QFileInfo

Pour obtenir des informations sur un fichier, comme la taille, la date de modification, etc., on peut utiliser `QFileInfo` :

Information sur les fichiers

```
QFileInfo fileInfo("example.txt");
qDebug() << "Nom du fichier:" << fileInfo.fileName();
qDebug() << "Taille du fichier:" << fileInfo.size();
```

Conclusion

La gestion des fichiers avec Qt est puissante et flexible. `QFile` et `QIODevice` permettent de gérer à la fois les fichiers texte et binaires, avec une attention particulière à la portabilité et à la gestion des erreurs. `QTextStream` et `QDataStream` simplifient la lecture et l'écriture respectivement de fichiers texte et binaires.

`QFileDialog` est une classe flexible et pratique pour intégrer des fonctionnalités de gestion de fichiers dans une interface utilisateur. Elle offre des moyens simples pour ouvrir, enregistrer des fichiers et sélectionner des répertoires, tout en offrant des options de personnalisation.