La FAQ Qt

Date de publication : 12/09/2006

Dernière mise à jour : 07/04/2009

La troisième version de Qt a été largement améliorée avec la sortie de la dernière version majeure : Qt 4. Qt est maintenant disponible sur les trois plateformes traditionnelles sous triple licences. Ce framework a été retravaillé en profondeur, bien des objets ont été transformés, supprimés ou ajoutés, afin de permettre à l'utilisateur de la bibliothèque de profiter pleinement de la puissance offerte par Qt.

Ont contribué à cette FAQ :

Matthieu Brucher (http://miles.developpez.com/) (Blog) - Alp Mestan (Site perso de Alp) (Blog) - Yan Verdavaine - IrmatDen - Shugo78 - Nykoo - mac&cheese - Benjamin Poulain - François Jaffré - kinji1 - gulish - haraelendil - Louis du Verdier - Aurelien.Regat-Barrel -

Sommaire > Introduction à Qt4

Quelle est la licence d'utilisation de Qt4 ?

Auteurs : Matthieu Brucher , Yan Verdavaine , Benjamin Poulain ,

Depuis la version majeure 4.5, Qt4 est distribué sous trois licences :

- **commercial : vous permet de faire ce que vous voulez avec le code de Qt et votre code.**
- **GPL V3 : tout code développé avec la version GPL de Qt doit aussi être GPL. Tout le code doit donc être accessible aux utilisateurs de l'application. C'est la licence généralement choisie pour les projets Open Source.**
- **LGPL v2.1 : similaire à la GPL, mais si Qt est lié dynamiquement à votre application, le code de votre application peut être fermé/propriétaire. Les modifications sur les sources de Qt seront obligatoirement LGPL. Avec cette licence, l'utilisateur doit avoir la possibilité de remplacer Qt par sa propre version. Cela n'est généralement un problème que lorsque Qt est intégré directement dans un appareil (télévision, GPS, etc).**

lien : Les licences Qt sur le site de Trolltech
lien : GPL
lien : LGPL

Où trouver la documentation de Qt ?

Auteurs : Alp Mestan ,

Trolltech propose une documentation plus qu'exhaustive pour chacune des versions de Qt. Le sommaire de toutes ces documentations se trouve ici : Online Reference Documentation

De plus, chaque version de Qt est distribuée avec un outil permettant de parcourir la documentation: Qt-assistant. La version de la documentation accessible avec Qt-assistant est correspond à la version de Qt avec lequel il a été distribué.

lien : Documentation de Qt 4.5

Quels sont les chemins des en-têtes à inclure lors de la compilation ?

Auteurs : Matthieu Brucher , Alp Mestan , Benjamin Poulain ,

Tout d'abord, Qt4 propose 2 types d'en-têtes, les en-têtes standards avec un .h comme extension, et les autres, sans extension.

Pour que la compilation se déroule correctement, il faut que le chemin vers les en-têtes de Qt soit inclus dans la liste des dossiers à parcourir pour les en-têtes, mais aussi chacun des sous-dossiers des bibliothèques que vous utiliserez.

Enfin, il faut savoir que pour chaque classe de Qt, il existe un en tête qui porte le même nom que la classe. Si par exemple vous utilisez QString, il faut un include du type

```
#include <QString>
```

Par défauts, les fichiers d'en-tête se trouve dans les frameworks sous Mac OS X (/Library/Frameworks), dans les includes sous Linux (généralement dans /usr/include/qt4), et dans le repertoire où Qt est installé sous Windows (généralement dans C:\Program Files\Qt\4.x.x\include). Pour compiler une application Qt, vous pouvez utiliser qmake qui définira ces chemins pour vous.

Comment débuter avec Qt ?

Auteurs : Benjamin Poulain ,

Qt est un framework orienté objet qui se base sur le langage C++. Pour bien profiter de Qt, il vaut donc mieux commencer par apprendre les bases de C++ et de la programmation orienté objet

Sur ces bases, un bon départ est de lire (et appliquer!) les tutoriels "Débuter dans la création d'interfaces graphiques avec Qt 4" et Carnet d'adresses. Qt introduit de nombreux concepts pour faciliter la programmation graphique, et lire le tutoriel peut vous être utile même si vous avez déjà utilisé une bibliothèque graphique.

Pour essayer les exemples du tutoriel, il faut un environnement de développement configuré pour utiliser Qt. Le plus simple pour commencer est d'utiliser Qt SDK (Qt + QtCreator) ou d'installer Qt et d'utiliser Monkey Studio ou QDevelop.

Lorsque vous avez acquis les concepts du tutoriel, vous pouvez continuer avec les exemples de Qt. Chaque version de Qt est distribuée avec un logiciel nommé "QtDemo", celui-ci est bourré d'exemple avec leur code source.

lien : Tutoriel : Installer Qt sur Mac avec Xcode ou QtCreator

lien : Tutoriel : installez 4.4.3 facilement et compilez vos applications

Sommaire > Généralités

Est-ce compliqué d'utiliser Qt 4 ?

Auteurs : Alp Mestan ,

En réalité, utiliser Qt est plus simple que ce que l'on pense. En effet, le code suivant affiche un bouton qui ferme l'application lorsque l'on clique dessus, et pourtant il ne fait que 13 lignes.

```
// Nécessaire pour créer une application avec Qt
#include <QApplication>
// Nécessaire pour pouvoir créer un bouton
#include <QPushButton>

// La fonction main() habituelle
int main(int argc, char **argv)
{
    // Qt récupère les arguments passés au programme
    QApplication    app(argc, argv);
    // on crée notre bouton, intitulé "Hello World"
    QPushButton     quit("Hello World!");
    // on le redimensionne
    quit.resize(300, 40);
    // on change la police et la taille
    quit.setFont(QFont("Arial", 18, QFont::Bold));
    // on explique à Qt que l'application doit se terminer lors du clic
    //sur le bouton créé précédemment
    QObject::connect(&quit, SIGNAL(clicked()), &app, SLOT(quit()));
    // on affiche le bouton
    quit.show();
    // on laisse Qt gérer le code de retour du programme
    return app.exec();
}
```

Comme vous pouvez le voir, il est très simple de gérer ses composants. Pour en découvrir plus, il est bon de consulter la documentation et les exemples de Trolltech, ainsi que les tutoriaux présents sur developpez.

Comment compiler des projets utilisant Qt 4 ?

Auteurs : Alp Mestan ,

Les développeurs de Qt 4 proposent des outils très utiles qui facilitent la gestion de vos projets utilisant Qt 4. En effet, il existe un outil permettant de transformer les fichiers du designer (.ui) en fichiers C++, un autre permettant de générer le code nécessaire pour la création de widgets personnalisés, ...

Cependant, ils fournissent également un outil permettant de gérer automatiquement les fichiers du designer, les fichiers dans lesquels vous définissez des widgets personnalisés, et même en réalité tout votre projet : qmake. Cet outil permet de générer un fichier de projet .pro, de générer à partir de ce dernier les règles de compilation de votre projet, et bien d'autres choses comme la détection de votre compilateur, du répertoire d'installation de Qt 4, ...

Un tutoriel a été écrit pour présenter cet outil et décrire son utilisation : Compilation des projets Qt 4.

lien : Compilation des projets Qt 4

Peut-on utiliser des pointeurs intelligents sans danger avec des QObject ?

Auteurs : Aurelien.Regat-Barrel ,

QObject (et toutes les classes qui en dérivent) ne se marrie pas très bien avec les pointeurs intelligents classiques du C++ tels que auto_ptr ou shared_ptr. Comme expliqué dans la question "Un new sans delete ?", Qt implémente via QObject un mécanisme de gestion de la mémoire dont le principe est qu'un QObject parent détruise automatiquement tous ses objets QObject enfants.

Ce mécanisme est fort pratique et permet de grandement simplifier la gestion de la mémoire. Cependant, il entre en conflit avec tout autre mécanisme de libération de la mémoire tel que ceux implémentés au moyen de pointeurs intelligents.

En effet, si vous déclarez un pointeur intelligent sur une instance de QObject, ce dernier aura en charge la destruction de cette instance. Si par malheur votre QObject se trouve être l'enfant d'un autre QObject, et que ce parent vient à être détruit, votre instance sera elle aussi automatiquement détruite, alors que votre pointeur intelligent continue de pointer vers elle! Et quand votre pointeur intelligent estimera qu'il est temps de la détruire , il effectuera un appel à delete sur un objet déjà détruit (double utilisation de delete) avec toutes les conséquences fâcheuses que l'on connait.

A noter que spécifier un parent nul au moment de la construction de votre QObject ne vous garantie pas pour autant que ce dernier ne sera pas par la suite "reparenté" via la fonction setParent(), chose qui se produit assez régulièrement entre objets graphiques (QWidget).

En conséquences, il est déconseillé d'utiliser des pointeurs intelligents sur des objets de type QObject. Si jamais vous devez néanmoins y avoir recours, vous devez protéger votre instance d'une double deletion au moyen d'un guarded pointer (QPointer en Qt 4, ou QGuardedPtr en Qt 3), comme dans l'exemple suivant :

```
shared_ptr<QPointer<QWidget> > ptr( new QWidget() );
```

La particularité d'un guarded pointer est qu'il est automatiquement mis à zéro si l'instance associée est détruite. Ainsi, dans cet exemple, quand shared_ptr effectuera un appel à delete sur une instance déjà détruite, cela reviendra à effectuer un delete sur un pointeur nul, c.a.d à ne rien faire du tout.

Ayez aussi à l'esprit que shared_ptr perd son sens originel et se comporte davantage comme un weak_ptr, c.a.d que vous devez systématiquement vous assurer qu'il ne soit pas nul avant de l'utiliser.

Vu la lourdeur de cette écriture et le changement de sémantique introduit, on comprend mieux pourquoi il est déconseillé d'employer des pointeurs intelligents sur des objets dérivant de QObjet.

lien : Faq C++ : pointeurs intelligents

lien : Tutoriel : Pointeurs intelligents

Comment avoir des classes utilisant Q_OBJECT sans .h ?

Auteurs : Yan Verdavaine ,

En C++, il peut être parfois intéressent de déclarer une classe directement dans un fichier d'implémentation (.cpp,.cxx,.cc ...). Seulement certain class exploité par Qt doivent être moccé.
Pour remédier à ce problème, il est tout à fait possible d'appliquer moc sur ce fichier. En contre partie, il faut inclure le fichier généré à la suite de la déclaration de la classe dans le .cpp.

Si vous utilisez qmake, il suffit, juste après la déclaration de la classe, d'inclure un fichier utilisant le nom du fichier cpp suivi de l'extension .moc. Cette méthode est utilisée dans certain code de la Q/R pour simplifier la compilation de l'exemple à un seul fichier

exemple : fichier nommé test.cpp

```
...
class myTest : QObject
{
    Q_OBJECT
    ...
};
#include "test.moc"
...
```

Comment Qt optimise t il les copies ?

Auteurs : Yan Verdavaine ,

Qt est basé sur une sémantique de copie. C'est à dire qu'elle utilise beaucoup la copie pour permettre de créer des logiciels robuste et évite un bon nombre de bugs. Pour optimiser cela, Qt implémente le pattern COW (Copy on Write). I.e. la copie interne est réellement effectuée lors d'un accès en écriture sur un objet interne partagé d'une classe. Dit d'une autre manière, Qt va partager en lecture un objet interne entre différent instances égaux d'une classe.

Le principe est le suivant :
1. **Lors d'une copie, un objet interne est partagé.**
2. **Tant qu'aucune des instances ne modifient cette objet, il reste partagé.**
3. **Dès qu'une des instances modifie l'objet , cette instance va copier l'objet interne et appliquer la modification.**

```
//s1 va créer un objet interne contenant "hello word"
QString s1 = "hello word";

//s2 va référencer le même objet interne que s1.
//Il n'y as pas eu de vrai copie.
QString s2 = s1;

//accès en lecture au premier caractère => inutile de faire une copie
qDebug()<< s2[0];

// on veut modifier la première lettre de s2
// s2 créé un nouvel objet interne et  copie le contenue.
// s1 et s2 n'utilisent plus le même objet interne
// s2 modifie la première lettre
s2[0] = 'b';
```

Remarque : ce pattern est thread safe dans l'implémentation Qt

lien : Liste des classes Qt basées sur le COW

Comment gérer la mémoire avec Qt ?

Auteurs : Benjamin Poulain ,

Qt est conçu de façon à rendre aisée la gestion de la mémoire. Il y a deux mécanismes pour y arriver: les types simples et les hiérarchies.

Les objets simples, tel que les chaînes de caractères, container, etc... peuvent être manipulés comme les types de bases, la mémoire est nettoyée automatiquement par le destructeur
Par exemple:

```
QString getSiteName()
{
    // pas de problème à l'allouer sur la pile
    QString siteName = "Developpez.com";
    // pas de problème à l'utiliser comme type de retour
    return siteName;
}
```

Ce type d'objet n'utilise en interne qu'un pointeur vers un délégué, ce qui veut dire que ces objets sont extrêmement rapides à manipuler:

```
QRectF rect(0.0, 0.0, 50.0, 50.0);
// le constructeur par copie est rapide
QRectF rectClone = rect;
QRectF copy;
// l'affectation est tout aussi rapide
copy = rect;
```

Ce type d'objet est donc très facile à utiliser car on peut l'employer comme si il s'agissait d'un type de base.

D'autres objets ont besoins d'une gestion de mémoire plus complexe car il faut contrôler précisément leur durée de vie. Pour ces objets, Qt a introduit le mécanisme de hiérarchie d'objet, qui est disponible pour toutes les sous-classes de QObject.

Les objets d'une hiérarchie peuvent avoir un objet parent, et des objets enfants (à ne pas confondre avec l'héritage, il s'agit ici d'encapsulation). Lorsqu'un objet est détruit, tous ses objets enfants sont détruits aussi.

Le parent est précisé dans le constructeur des objets ou grâce à la méthode QObjet::setParent(). Voici un exemple:

```
QString giveString()
{
    QObjet parent;
    MyObjet *myObjet = new MyObject(parent);
    myObjet->doSomeComputation();
    return myObjet->getString();
}
```

Étonnamment, le code précédent n'a pas de fuite de mémoire. L'objet MyObjet, créé sur le tas, a comme parent le QObject "parent". Lorsque parent arrive à la fin de la fonction giveString(), son destructeur se charge de supprimer l'objet myObjet.

L'exemple précédent est artificiel, voyons un exemple plus réaliste:

```
void showMessage()
{
    QDialog dialog;
    QVBoxLayout *layout = new QVBoxLayout(&dialog);
    QLabel *message = new QLabel("Cliquez sur le bouton");
    layout->addWidget(message);
    QButton *button = new QButton("Cliquez ici");
    layout->addWidget(button);
    dialog.exec();
}
```

Quelle magie fait que ce code fonctionne? Lorsqu'un widget est ajouté au layout, le layout le reparente automatiquement avec son widget propriétaire (dialog dans ce cas ci). Finalement, lorsque la fonction se termine, le destructeur de "dialog" supprime tous les objets et nettoie la mémoire.

lien : FAQ Comment optimiser la copie de ses classes ?

lien : FAQ Comment optimiser la copie de ses classes ?

Sommaire > Le contenu de Qt4

Que contient Qt4 ?

Auteurs : Matthieu Brucher ,

Qt4 a été découpée en plusieurs sous-bibliothèques relativement indépendantes les unes des autres.

- **QtCore, qui contient les éléments essentiels du fonctionnement de toutes les bibliothèques Qt**
- **QtGui, qui contient les éléments essentiels graphiques pour les autres bibliothèques Qt**
- **QtNetwork, contenant des classes dédiées à la programmation réseau**
- **QtOpenGL, permettant le support d'OpenGL**
- **QtSql, permettant l'intégration de bases de données dans Qt**
- **QtSvg, permettant la lecture et l'écriture de fichiers SVG**
- **QtXml, contenant les classes utiles à la lecture et à l'écriture de fichiers XML**
- **QtDesigner, permettant l'extension du QtDesigner**
- **QtUiTools, permettant l'utilisations des outils de QtDesigner dans des applications**
- **QtAssistant, l'aide de Qt**
- **Qt3Support, permettant de réutiliser presque de manière transparente les anciennes classes de Qt3**
- **QtTest, contenant des outils permettant de réaliser des tests unitaires**

Chaque bibliothèque peut être ajoutée ou retirée des projets Qt, sachant que QtCore et QtGui sont ajoutées par défaut.

lien : Les modules Qt sur le site de Trolltech

Sommaire > Le contenu de Qt4 > QtCore

Un new sans delete ?

Auteurs : Matthieu Brucher ,

Dans un code Qt4, on verra souvent un new sans delete associé.

En fait, si une nouvelle instance de QObject est crée et qu'on lui spécifie un parent - le premier argument du constructeur -, c'est ce parent qui sera chargé de la destruction du fils.

La majorité des classes de Qt4 hérite plus ou moins directement de QObject, mais attention, ce n'est pas le cas de toutes. L'indicateur est la possibilité de passer un objet parent au constructeur.

lien : La classe QObject

Les signaux et autres slots ?

Auteurs : Matthieu Brucher ,

Les signaux et slots sont l'implémentation par Trolltech du pattern observer. Ce pattern permet de prévenir les classes voulant observer un évènement.

Pour utiliser ce système, il faut hériter de QObject, soit directement, soit à l'aide d'un des objets dérivés de QObject. Ensuite, il faut définir la macro Q_OBJECT qui servira à qmake puis à moc pour générer le code nécessaire à l'utilisation de ces fonctions.

Pour déclarer des signaux, il suffit d'indiquer :

```
signal:
 void monSignal();
```

pour que moc génère le code qui va bien et pour que le signal puisse être utilisé. On appellera un signal par :

```
emit monSignal();
```

Ces signaux peuvent être connectés à d'autres signaux ou à des slots. Ces slots sont des fonctions du programmeur qui seront appelées dès que possible.

```
public slots:
  void monSlot();
```

Une fois les slots définis, il suffit de connecter les signaux et les slots entre eux. Comme il s'agit de connexion directe, il n'est pas possible de connecter un signal sans paramètre à un slot avec un paramètre. Pour cela, il faut utiliser entre-temps QSignalMapper.

```
connect(this, SIGNAL(monSignal(), somethingElse, SLOT(monSlot()));;
```

On constatera aussi que signaux et slots ne peuvent pas retourner de valeur, pour l'instant.

lien : Les signaux et slots
lien : Tutoriel sur les signaux et slots de Qt 4

Comment s'interfacent les signaux et les slots?

Auteurs : Nykoo ,

Dans la question " Les signaux et autres slots" nous avons vu comment s'interface une connexion entre signaux et slots simples, c'est à dire sans transmission de valeur.
Or, les signaux et slots ont la capacité de se transmettre des données par le biais de leurs arguments.

Prenons l'exemple de la classe QLineEdit de Qt.
Son signal textChanged(const QString&) permet de récupérer le texte présent dans le QLineEdit au moment de l'émission du signal.
De même, son slot setText(const QString&) permet de redéfinir le texte contenu dans le QLineEdit à l'aide d'un objet QString.

Rappel: Il est possible de connecter un signal avec un slot, mais on peut également connecter un signal à un autre signal. Pour créer une connexion signal/slot (ou signal/signal) qui permette la transmission de valeurs, il suffit d'écrire la signature complète de chaque signal/slot dans la fonction connect().
Autrement dit, il faut indiquer le nom des signaux/slots en question, ainsi que les types des arguments qu'ils prennent en paramètre :

Exemple de signature complète

```
monSignal(QString,int) //signature d'un signal
monSlot(QString,int) //signature d'un slot
connect(objet1,SIGNAL(monSignal(QString,int)),objet2,SLOT(monSlot(QString,int)))
```

Attention, car il faut prendre certaines précautions. Dans une fonction connect(), les types des arguments des deux fonctions doivent être compatibles et placés dans le même ordre. Des arguments sont compatibles si le même objet est en jeux. Ex: const QString& est compatible avec QString. Par contre QString* et QString& ne le sont pas. Voici un exemple de connexion avec 2 types compatibles.

```
connect(this,SIGNAL(monSignal(QString,int)),somethingElse,SLOT(monSlot(const QString&#38;,int)));
```

Cependant, le signal ou le slot qui est situé dans la partie droite de la fonction connect() peut avoir un nombre d'arguments inférieur ou égal à celui du signal situé à gauche. Par exemple la connexion suivante:

```
connect(this,SIGNAL(monSignal(QString,int)),somethingElse,SLOT(monSlot(QString)))
```

est valide. L'argument int sera simplement ignoré.

Comment créer ses propres signaux et slots avec transmission de valeurs?

Auteurs : Nykoo ,

Cette question nécessite d'avoir compris la création de signaux/slots simples (sans arguments) décrite dans la question "Les signaux et autres slots"

Pour créer un signal ou un slot avec des arguments il suffit d'ajouter les noms des types dans le prototype:

```
class myClass : public xxx
{
    Q_OBJECT
    ...
signals:
    void monSignal(type1,type2)
public slots:
    void monSlot(type1,type2);
    ...
}
```

Ainsi, pour émettre son signal il suffit d'utiliser le mot clef emit.

Emmission du signal void monSignal(nom_type)

```
void fonction()
{
    nom_type t;
    emit monSignal(t);
}
```

La connexion se fera avec un signal ou un slot qui prend un argument de type nom_type.

Comment utiliser les auto-connexions ?

Auteurs : mac&cheese ,

Sous Qt, les connexions entre les signaux et slots peuvent être mis en place soit manuellement, soit automatiquement, en utilisant la capacité qu'a QMetaObject d'établir des liens entre ces derniers.

Cette partie ne concerne pas les connexions manuelles (traitées à cette adresse), en revanche elle traite de la méthode automatique.

Bien qu'il soit plutôt aisé d'implémenter un slot et de le connecter dans le constructeur, nous pouvons tout aussi bien utiliser l'outil d'auto-connexion de QMetaObject pour connecter le signal clicked() de myButton à un slot dans notre classe.

Bien qu'il soit plutôt aisé d'implémenter un slot et de le connecter dans le constructeur, nous pouvons tout aussi bien utiliser l'outil d'auto-connexion de QMetaObject pour connecter les signals et les slot clicked() :

```
QMetaObject::connectSlotsByName(QObject *object);
```

Cette fonction du QMetaObject de Qt connecte automatiquement tous slots qui respectent la convention on_Nomobjet_Nomsignal() au signal Nomsignal() correspondant de l'object Nomobjet. Deux conditions sont à remplir:

- **L'objet NomObject doit être un enfant de l'objet passé en paramètre à la méthode connectSlotByName().**
- **Les objets (parents et enfants) doivent être nommés avec la fonction QString.**

exemple

```
class MainWindows : public QWidget
{
```

exemple

```
    Q_OBJECT
public :
    MainWindows(QWidget * parent = 0);

public slots :
    //slot qui sera connecté au signal clicked de l'objet myButton
    //par l'autoconnect
    void on_myButton_clicked();

};

MainWindows::MainWindows(QWidget * parent):QWidget(parent)
{
    ...
    QPushButton * pButton = new QPushButton("essai auto connect");
    //nom du bouton pour être retrouvé par l'autoconnect
    pButton->setObjectName("myButton");
    ...
    //auto connection des signaux/slots
    QMetaObject::connectSlotsByName(this);
}
```

Attention: supposons, cette fois-ci, que nous créons notre fenêtre via le Qt Designer et non plus en ligne de code; nous possédons alors un fichier supplémentaire, à savoir, mainwindows.ui. Lors de la compilation, l'outil uic de Qt, se charge de générer le code de la fonction setupUi() de notre fenêtre (mainwindows.ui). Le code généré utilise alors de la même façon :

- **QString : pour nommer les objets définis dans le *.ui**
- **QMetaObject::connectSlotsByName : pour autoconnecter les objets définis dans le *.ui pendant l'appel à setupUi.**

Il est donc possible d'utiliser de la même façon, les auto-connexions sur une GUI codée à la main que sur une GUI créée à l'aide de Qt Designer (comment utiliser les *.ui)

REMARQUE : Si deux enfants ont le même nom, l'auto-connect ne se fera que sur le premier enfant trouvé.

REMARQUE2 : Si QMetaObject::connectSlotsByName est appelé plusieurs fois, les connections générées seront multiples. Lors de l'utilisation d'un *.ui, ne pas oublier que la fonction setupUi appel l'autoconnection

Voici deux exemples à télécharger :

Utilisation de autoconnect sans *.ui Utilisation de autoconnect avec *.ui

Comment paramétrer un slot selon les objets qui lui sont connectés ?

Auteurs : kinji1 ,

Qt propose de nombreuses classes avec de nombreux signaux déjà définis. Cependant il peut parfois être intéressant de rajouter de l'information à ces signaux afin de les paramétrer selon l'objet émetteur. On pourrait ainsi vouloir que différents boutons réalisant une action commune, par exemple ouvrir une page web lorsque l'on clique dessus, soient connecté à un même slot auquel on précise l'URL à utiliser. Seulement le signal clicked() de la classe QPushButton ne transmet aucun paramètre, ce qui ne permet pas de spécifier l'URL.

Une première solution consiste alors à connecter le signal clicked() de chaque bouton à un slot différent qui se contentera d'appeler la fonction d'ouverture de la page web avec l'url correspondante. Cependant, comme il est nécessaire de créer un slot différent par bouton, cela rallonge inutilement la taille du code, surtout lorsqu'il y a un grand nombre de boutons.

Une autre solution est d'utiliser la classe QSignalMapper. Dans notre exemple, celle-ci va s'occuper d'appeler le slot ouvrant la page web avec un paramètre configuré pour chacun des boutons (l'URL de la page). Nous avons donc d'un côté les signaux clicked() des différents QPushButton, et de l'autre un slot openUrl(const QString& url) et le QSignalMapper au milieu pour faire les correspondances.

Tout d'abord il faut créer un objet QSignalMapper, puis connecter les signaux des boutons à son slot map(). On définit alors, pour chacun des boutons, le paramètre à utiliser pour l'appel à openUrl via setMapping().

```
mapper = new QSignalMapper();

// Bouton 1
connect(bouton1, SIGNAL(clicked()), mapper, SLOT(map()));
mapper->setMapping(bouton1, "http://url1");

// Bouton 2
connect(bouton2, SIGNAL(clicked()), mapper, SLOT(map()));
mapper->setMapping(bouton2, "http://url2");

// Autres boutons ?
```

Enfin, il suffit de connecter le signal mapped de QSignalMapper à notre slot final. Ainsi quand un bouton émettra son signal, le slot openUrl sera utilisé avec le paramètre correspondant au bouton.

```
connect(mapper, SIGNAL(mapped(const QString &)), this, SLOT(openUrl(const QString &)));
```

Note: QSignalMapper ne se limite pas à des paramètres de type QString. Il est également possible d'utiliser des entiers ou encore des QWidget* et des QObject*. Il faut dans ce cas utiliser le signal mapped correspondant au type que l'on veut transmettre.

Erreur d'édition des liens undefined reference to 'vtable for xxx' ?

Auteurs : Matthieu Brucher ,

Cette erreur se produit lorsque la partie QObject d'une classe n'a pas été ajoutée à l'édition des liens.

Lors de l'utilisation de la macro Q_OBJECT, on définit un certain nombre de méthodes et de variables statiques. Ces méthodes et ces variables sont implémentées dans un fichier généré automatiquement par qmake à l'aide de l'outil moc. Vous pouvez naturellement créer ce fichier manuellement et l'ajouter pour compilation et édition des liens.

lien : La classe QObject

lien : Les signaux et slots

Comment ouvrir une application à partir de Qt ?

Auteurs : Yan Verdavaine ,

Qt fournit la classe QProcess. Cette classe permet de contrôler l'exécution d'une application dans un nouveau process. Il est ainsi possible de contrôler :

- les variables d'environnements par la méthode setEnvironement()
- le répertoire d'exécution par setWorkingDirectory()
- de lancer l'application par la fonction start

La fonction start qui crée un process enfant (sera fermé si le process parent est fermé) avec pour paramètres :

- **path de l'exécutable**
- **liste des paramètres d'entrée de l'exe**
- **un mode d'ouverture pour interagir avec l'application par le biais des entrées/sorties standards**

Cette classe possède d'autres fonctions pour lancer une application :

- **QProcess::execute : équivalent à start. Bloquante jusqu'à la fin de l'exécution de l'application lancée**
- **QProcess::startDetached : permet de lancer une application dans un process indépendant. Peut donner le pid du process créé**

Il est bon de remarquer que ces fonctions:

- **ont des paramètres similaires à ceux de start :**
  - **path de l'exécutable**
  - **liste des paramètres d'entrée de l'exe**
- **ne donne pas de moyen pour interagir avec le process par le biais des entrées/sorties standards**
- **ces deux fonctions sont des fonctions static et peuvent être appelées sans création d'un QProcess. Dans ce cas les variables d'environnement et le répertoire d'exécution sera le même que ceux de l'application courante.**

Exemple utilisant QProcess::startDetached

```
#include <QtGui>

class MyQPushButton : public QPushButton
{
    public :
    //createur.
    //text : text du bouton
    //exe  : commande à exécuter lors de l'appui.
    MyQPushButton(const QString & text,const QString & exe, QWidget * parent = 0)
    : QPushButton (text,parent),
    m_exe(exe)
    {
        resize(75, 30);
        setFont(QFont("Times", 18, QFont::Bold));
    };
    virtual void mouseReleaseEvent ( QMouseEvent * event )
    {
        //lance la commande dans un process indépendant
        QProcess::startDetached (m_exe);
    }

    private :
        QString m_exe;

};

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QWidget w;
    QVBoxLayout vl(&w);
    //boutton : ouvre grâce au CMD la page vers DVP/Qt
    MyQPushButton bouton1("Ouvrir DVP / Qt","cmd /c start http://qt.developpez.com/",&w);

    //boutton : lance notepad
    MyQPushButton bouton2("Ouvrir notepad","notepad",&w);

     //boutton : lance invite de commande
```

Exemple utilisant QProcess::startDetached

```
    MyQPushButton bouton3("Ouvrir invite de commande","cmd" ,&w);
    vl.addWidget(&bouton1);
    vl.addWidget(&bouton2);
    vl.addWidget(&bouton3);
    w.show();
    return app.exec();
}
```

Comment interagir avec une application lancée par un QProcess?

Auteurs : Yan Verdavaine ,

QProcess permet de lancer des applications externes (Comment ouvrir une application à partir de Qt ?). Elle permet surtout d'interagir avec l'application à la manière d'un pipe au travers des entrées/sorties standards de l'application. Ainsi le troisième paramètre de la fonction start permet de spécifier quel type d'interaction on veut utiliser (par défaut, les deux types sont activés) :

- **mode read : récupération de la sortie standard (stdout) et d'erreur(stderr)**
- **mode write : écriture sur l'entrée standard de l'application (stdin)**

QProcess utilise les méthodes définies par QIODevice pour faire ces interactions. Pour la récuperation des sorties, il faut faire attention au channel que l'on est en train de lire :

- **pour la sortie standard il faut utiliser : QProcess::StandardOutput**
- **pour la sortie d'erreur il faut utiliser : QProcess::StandardError**

De plus Qprocess fournit deux signaux intéressants, qui indiquent si des données venant d'une des sorties sont arrivées :

- **readyReadStandardOutput () : des données provenant de la sortie standards sont prêtes à être traitées**
- **readyReadStandardError () : des données provenant de la sortie d'erreur sont prêtes à être traitées**

Il est aussi possible de faire ces interactions à l'aide de fichiers :

- **setStandardErrorFile : fichier où l'on récupère la sortie d'erreur de l'application lancée**
- **setStandardOutput : fichier ou l'on récupère la sortie standard de l'application lancée**
- **setStandardInputFile : fichier contenant les donné à envoyer sur l'entrée standard de l'application lancée**

Attention : sous windows, un programme lisant l'entrée standard (stdin) en parallèle de son ihm, peut se bloquer lorsqu'il est exécuté en process enfant. Ce bug est normalement corrigé sous vista.

TestQProcess_zip

Comment interagir avec les applications associées par défaut ?

Auteurs : Yan Verdavaine , IrmatDen ,

Qt fournit la classe QDesktopServices et en particulier la fonction static QDesktopServices::openUrl, qui permet d'interagir avec les applications associées par défaut. Elle peut être étendue si nécessaire pour ajouter ses propres handlers pour un schéma donné.

```
#include <QtGui>
```

```
class MyQPushButton : public QPushButton
{
public :
    //createur.
    //text : texte du boutton
    //exe  : commande à executer lors de l'appui.
    MyQPushButton(const QString & text,const QString & url, QWidget * parent = 0)
    :  QPushButton (text,parent) , m_url(url)
    {
        resize(75, 30);
        setFont(QFont("Times", 18, QFont::Bold));
    };
    virtual void mouseReleaseEvent ( QMouseEvent * event )
    {
        QDesktopServices::openUrl(m_url);
    }

private :
    QString m_url;

};

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QWidget w;

    //Action : ouvre l'url ver DVP /Qt
    MyQPushButton bouton1(" Ouvrir DVP / Qt","http://qt.developpez.com",&w);
    //Action : ouvre le fichier texte. /!\ ce fichier doit exister
    MyQPushButton bouton2("Ouvrir fichier txt","c:/test.txt",&w);
    //Action : ouvre l'edition d'un mail avec l'adresse et le sujet remplis
    MyQPushButton bouton3("Envoyer mail","mailto:qt@dvp.com?subject=test envoie mail" ,&w);

    QVBoxLayout vl(&w);
    vl.addWidget(&bouton1);
    vl.addWidget(&bouton2);
    vl.addWidget(&bouton3);
    w.show();
    return app.exec();
}
```

Comment charger et utiliser dynamiquement une .dll, .so avec Qt ?

Auteurs : IrmatDen ,

Qt fournit la classe QLibrary permettant de charger de façon multi-plateformes une bibliothèque dynamique, ainsi que d'en récupérer des pointeurs vers les fonctions exportées.

Après avoir créé un objet de ce type, il faut spécifier le nom de la bibliothèque dynamique à associer sans préciser l'extension (ce n'est plus multi-plateformes sinon). Il suffit ensuite d'appeler resolve() en fournissant le nom du symbole à trouver. Il est retourné en tant que void*, donc un cast sera bien évidemment nécessaire. Sont aussi fournies des fonctions statiques évitant l'instanciation dans le cas où l'on ne voudrait récupérer qu'un symbole.

Par exemple, admettons qu'une fonction d'une bibliothèque permette de compter le nombre de lettres dans un mot :

```
// définition du type de fonction
typedef int (*StringLength)(char*);
// création d'un objet QLibrary lié à string_util (.dll, .so ou autre)
QLibrary lib("string_util");
// récupération d'un pointeur sur notre fonction partagée
StringLength strLength = (StringLength)lib.resolve("strlen");
```

```
if(strLength)
// devrait renvoyer 9... si tout va bien  ;)
    strLength("QLibrary");
```

Comment optimiser la copie de ses classes ?

Auteurs : Yan Verdavaine ,

Une grosse partie de Qt est basé sur le COW. Il permet ainsi d'utiliser ce pattern. Pour cela, il faut créer une classe héritant de QSharedData est possédant un constructeur, un constructeur par recopie et un destructeur public. Cette classe sera l'objet interne qui sera partagé. Elle possède un compteur de référence thread safe et ne doit pas être directement accédé. Sa vie sera gérée par d'autres classes.

Pour accéder à une instance de cette classe, deux choix sont possible :

- **QSharedDataPointer: permet de partager implicitement un QSharedData. L'objet interne est partagé en lecture. L'accès à l'objet en écriture va générer une recopie de l'objet.**
- **QExpliciteSharedDataPointer : permet de partager explicitement un QSharedData. L'objet interne est partagé en lecture/écriture. L'objet interne sera recopié uniquement sur demande.**

Ces deux classes sont des pointeurs intelligents spécialisé sur la manipulation des pointeurs sur QSharedData. Ils implémentent donc la sémantique des pointeurs avec des accès const (lecture) et non const (écriture). Elles détruiront le QSharedData une fois son compteur à zéro. Ces pointeurs intelligents possèdent deux fonctions qu'il est utile de connaitre :

- **Detach() : Si le compteur de référence est > 1, le QSharedData sera copié**
- **Reset() : Initialise à null le pointeur intelligent**

Remarque : ces classes sont thread safe

lien : Comment Qt optimise t il les copies ?

lien : Faq C++ : pointeurs intelligents

lien : Tutoriel : Pointeurs intelligents

Comment utiliser un QTimer ?

Auteurs : François Jaffré ,

Les timers sont gérés avec Qt à l'aide de la classe QTimer. Cette classe est relativement simple à utiliser grâce aux méthodes suivantes :

- **int interval () const -> Permet de connaitre l'intervalle de temps (en ms) entre chaque déclanchement du timer.**
- **bool isActive () const -> Permet de connaitre si le timer est activé ou pas.**
- **bool isSingleShot () const -> Permet de connaitresi le timer est en déclanchement unique ou pas.**
- **void setInterval (int msec) -> Permet de paramétrer l'intervalle (en ms) entre chaques déclanchements du timer**
- **void setSingleShot (bool singleShot) -> Permet de mettre le timer en mode déclanchement unique**
- **void start() -> Permet de démarrer le timer**
- **void stop() -> Permet d'arrêter le timer**

Création d'un chronomètre simple précis à la seconde

```
#include <QApplication>
#include <QLCDNumber>
#include <QPushButton>
#include <QTimer>
#include <QGridLayout>

class TimerChrono : public QWidget
{
    Q_OBJECT

private :
    //Bouton servant de "Start" "Stop"
    QPushButton* m_Bouton_StartStop;
    //Bouton "Reset"
    QPushButton* m_Bouton_Reset;
    //Afficheur de type LCD
    QLCDNumber* m_LCD;
    //Variable représentant le nombre de secondes écoulées depuis le lancement du timer
    int m_Timer_value;
    //Timer servant de base à notre Chronomètre
    QTimer* m_timer;
    //Permet de savoir si l'utilisateur a cliqué sur "Start" ou "Stop"
    bool validStart;

public :
    TimerChrono()
    {
        //Creation d'un afficheur LCD de 5 digits maximum
        this->m_LCD = new QLCDNumber(5, this);
        //Creation des contrôles de types boutons
        this->m_Bouton_StartStop = new QPushButton("Start",this);
        this->m_Bouton_Reset = new QPushButton("Reset",this);

        //Gestion du layout pour le placement des boutons
        QGridLayout *layout = new QGridLayout();
        layout->addWidget(m_LCD, 0, 0);
        layout->addWidget(m_Bouton_StartStop, 2,0);
        layout->addWidget(m_Bouton_Reset, 2,1);
        this->setLayout(layout);

        //On met à zeros le compteur représentant le nombre de seconde
        this->m_Timer_value=0;
        //Création du timer
        this->m_timer = new QTimer(this);
        //A chaque fin d'interval execution de la fonction update()
        connect(this->m_timer, SIGNAL(timeout()), this, SLOT(update()));
        //On applique un interval d'une seconde (1000 ms) entre chaque timeout du timer
        this->m_timer->setInterval(1000);
        //Sert pour la gestion du bouton "Start" "Stop"
        this->validStart=true;

        //On connecte les differents signaux et slots
        connect(this->m_Bouton_StartStop, SIGNAL(clicked(bool)), this, SLOT(click_StartStop(bool)));
        connect(this->m_Bouton_Reset, SIGNAL(clicked(bool)), this, SLOT(click_Reset(bool)));
```

Création d'un chronomètre simple précis à la seconde

```
         //Gestion de la taille de la fenêtre
        resize(200, 150);
    }

private slots:
    //Fonction appellée toute les secondes par le QTimer
    void update()
    {
        //On incremente notre compteur de secondes
        m_Timer_value++;
        //On affiche le nombre de seconde ecoulé dans le contrôle QLCDNumber
        m_LCD->display(m_Timer_value);
    }

    //Bouton reset
    void click_Reset(bool valid)
    {
        // On arrete le timer
        this->m_timer->stop();
        //On met notre compteur à zero
        m_Timer_value=0;
        // On affiche Zero dans le controle LCD
        m_LCD->display(m_Timer_value);
        //Gestion du bouton "Start" "Stop"
        this->validStart=true;
        m_Bouton_StartStop->setText("Start");
    }

    //Bouton "Start" "Stop"
    void click_StartStop(bool valid)
    {
        //Si on click sur "Start"
        if(validStart == true)
        {
            //Affiche "Stop" sur le bouton
            m_Bouton_StartStop->setText("Stop");
            //Permet de savoir que le bouton est en mode "Strop"
            validStart = false;
            //On declanche le départ du Timer
            this->m_timer->start();
        }
        else//Si on click sur "Stop"
        {
            //Affiche "Sart" sur le bouton
            m_Bouton_StartStop->setText("Start");
            //Permet de savoir que le bouton est en mode "Start"
            validStart = true;
            //On arrete le compteur
            this->m_timer->stop();
        }
    }
};

#include "main.moc"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    TimerChrono w;
    w.show();
    return a.exec();
}
```

Remarque 1 : Si vous souhaitez utiliser un timer en mode "single shot" QTimer possède une methode static QTimer::singleShot() qui peut vous être très utile.

Remarque 2 : Un timer n'est jamais très précis et est fortement dépendant de l'OS. Si vous souhaitez être précis à la milliseconde près ce n'est pas la solution à utiliser.

lien : QTimer
lien : QTimer::SingleShot

Comment récupéré l'heure du PC ?

Auteurs : François Jaffré ,

La classe QTime permet de récupéré l'heure du PC facilement à l'aide de la méthode static suivantes :

- **QTime QTime::currentTime() -> Permet de récupérer un objet QTime initialisé à l'heure courante.**

```
//L'objet time est initialisé à l'heure courante
QTime time = QTime::currentTime();
```

lien : QTime

Comment mesurer un intervalle de temps ?

Auteurs : François Jaffré ,

QTime fournit la méthode elapsed() qui permet de connaitre le temps écoulé (en ms) entre l'exécution de la méthode start() ou restart() et l'appel à la fonction elapsed().

```
QTime time;
time.start();
ma_fonction();
//millisecondes contient le nombre de millisecondes entre l'appel à la fonction start()
//et l'appel a la fonction elapsed().
int millisecondes = time.elapsed();
```

Remarque : Si vous voulez vraiment mesurer précisément votre code et faire du benchmarking de votre application alors il est préférable de vous tournez vers la QTestLib qui est faite pour ça.

lien : QTime

lien : QTestLib

Comment récupérer les arguments envoyés par la fonction main ?

Auteurs : Louis du Verdier ,

Dans les années 1980 à 1990 (aucune information ne précise la date exacte), la fonction main() ne renvoyait rien, elle était donc de type void. Plus récemment, des améliorations de cette fonction ont introduit les arguments qui sont en fait dans la norme actuelle de mettre "int argc" et "char *argv[]". Ces arguments sont complémentaires : "char *argv[]" représente un tableau de pointeurs de taille donnée par "int argc". Sur la plupart des systèmes d'exploitation, le tout sert par exemple à retrouver le chemin du programme lancé (avec argv[0]), ou encore par exemple à retrouver un fichier par lequel s'est ouvert le programme (argc serait donc supérieur à 1).

Quel rapport avec Qt ? Et bien la bibliothèque permet de récupérer les arguments à l'aide de la fonction QCoreApplication::arguments. La récupération sert par exemple à récupérer le chemin d'un éventuel fichier ouvert par clic sur l'icône de celui-ci.

```
QStringList args = QCoreApplication::arguments();
QString nom_fichier = args[1];
```

Note : Lors de la déclaration de la fonction main, il faut mettre les arguments pour que ce qui est donné marche, c'est à dire utiliser un début de code comme celui-ci pour la fonction main :

```
int main (int argc, char *argv[])
{
    QApplication app(argc,argv)
    // Suite du code
}
```

Remarque : En règle générale, le premier élément correspond au nom de l'exécutable. Suivant la plateforme et la mode d'exécution, cet élément peut aussi contenir le chemin relatif ou le chemin absolu de l'exécutable concaténé avec son nom.

Voici un exemple permettant de clore :

```
const QStringList args = QCoreApplication::arguments();
// Si le programme a été ouvert par le biais d'un fichier
if(args.count() > 1)
{
    // On récupère le chemin du fichier...
    QString nom_fichier = args[1];
    // ... et on appelle une éventuelle fonction de lecture
    lireFichier(nom_fichier, this);
}
// ...
```

Attention : Sous mac, le double-clic sur un fichier ne sera pas renseigné dans les arguments, contrairement à Windows et Linux. Pour cela il faut utiliser QFileOpenEvent.

lien : QFileOpenEvent

Sommaire > Le contenu de Qt4 > QtCore > QObject

Allouer sur le tas ou sur la pile des QObject et dérivés ?

Auteurs : Matthieu Brucher ,

Chaque QObject contient une liste des pointeurs vers les QObjects fils. Cette liste permet d'effacer automatiquement les objets fils de cette liste.

Cette liste implique que les objets fils héritant de QObject devraient être alloués sur le tas et non la pile. A priori, le code d'effacement est fait de telle sorte qu'il ne devrait pas y avoir de problème en allouant un objet sur la pile car la destruction d'un objet entraîne sa suppression dans la liste des parents. En revanche, effacer manuellement ou automatiquement par destruction dans la pile les objets fils peut entraîner un surcoût. Enfin, étant utilisé avec une sémantique de pointeurs, il vaut mieux utiliser les pointeurs.

De même, une instance héritée de QObjet ne peut appartenir à 2 QObjects.

lien : La classe QObject

lien : Le modèle QObject

Héritage multiple avec QObject ?

Auteurs : Matthieu Brucher ,

L'héritage multiple de QObject n'est pas possible. En effet, l'architecture employée par Trolltech - les meta objets - rend la chose impossible.

Lorsque vous faites un héritage multiple, mettez en premier la classe héritant de QObject.

lien : La classe QObject

Sommaire > Le contenu de Qt4 > QtCore > QString

Faut-il utiliser les pointeurs avec QString ?

Auteurs : Benjamin Poulain ,

Ce n'est pas nécessaire. QString utilise un pointeur partagé pour stocker les caractères. Les opérations de copie et d'affectation sont aussi performante qu'en manipulant des pointeurs.

```
QString texte = "Developpez!";
// la copie est rapide
QString copie = texte;
QString secondeCopie;
// l'affectation est rapide
secondeCopie = copie;
// utiliser QString en argument est efficace
uneFonction(texte);
```

lien : FAQ Comment Qt optimise t il les copies ?

Transformer un QString en std::string et réciproquement ?

Auteurs : Matthieu Brucher ,

Pour transformer un std::string en QString, utilisez la fonction statique QString::fromStdString().

Pour l'opération inverse, transformer un QString en std::string, utilisez la fonction membre toStdString().

lien : La classe String

Comment convertir un QString en chaîne C (char *) ?

Auteurs : Yan Verdavaine ,

Pour convertir un object QString en chaîne de 8 bits, il faut utiliser les méthodes toAscii(), toLatin1(), toUtf8(), toLocal8Bit(). Ces méthodes retournent un objet QByteArray, qui permet de gérer la mémoire. QByteArray fournit les méthodes constData() et data() pour accéder à un pointeur sur un tableau de char. Ce pointeur est valide aussi longtemps que l'objet n'est pas modifié ou détruit.

```
QString adresseIp("127.0.0.1");
QByteArray adresseIpEncodee = adresseIp.toUtf8();
printf(adresseIpEncodee.constData());
```

Avec des chaînes de type C, il faut être attentif à la mémoire. Par exemple, le code suivant provoquera des problèmes de mémoire:

```
QStringList list;
list << "C" << "C++" << "Qt";
QList<const char*> ipListEnChar;
foreach(QString ip, list)
{
    QByteArray adresseIpEncodee = ip.toUtf8();
    ipListEnChar << adresseIpEncodee.constData(); // le pointeur sera invalide
}
foreach(const char *str, ipListEnChar)
{
    printf(str); // le résultat est imprévisible
}
```

Il faut revenir aux vielles habitudes de C et gérer les allocations à la main. Il suffit dans l'exemple de s'assurer que les objets QByteArray ne soient pas détruits:

```
QStringList list;
list << "C" << "C++" << "Qt";
QList<QByteArray> ipListEnCharData;
QList<const char*> ipListEnChar;
foreach(QString ip, list)
{
      QByteArray adresseIpEncodee = ip.toUtf8();
      ipListEnChar << adresseIpEncodee.constData();
      ipListEnCharData << adresseIpEncodee;
}
foreach(const char *str, ipListEnChar)
{
    printf(str);
}
```

Dans un problème réel, le tableau de char* serait caché dans les fonctions C utilisées, et il faut être rester attentif à la gestion de la mémoire.

Remarque : QString permet aussi une conversion vers les formats std::string et std::wstring fournis par la STL.

lien : FAQ Transformer un QString en std::string et réciproquement ?

Formatage de texte avec QString ?

Auteurs : Yan Verdavaine ,

QString implémente un équivalent au printf et boost::format pour faire du formatage de string. Pour cela, QSting utilise des formateur sous la forme %n (ou n est un nombre) et les fonctions arg.

Chaque fonction arg retourne une nouvelle string où les %n sont remplacés par ordre croissant. La fonction arg possède énormément de versions différentes qui vont permettre de remplacer un formateur par un nombre ou une string. De plus si plusieurs formateurs ont le même nombre, ils seront remplacés par la même valeur.

```
QString  s = "%2 et %1 ou %1 et %3";
//on remplace le plus petit formateur contenue dans s
//tout les %1 sont remplacé par le nombre 42
//    s1 == "%3 et 42 ou 42 et %2"
//    s n'est pas modifié
QString  s1 = s.arg(42);

//on remplace plusieurs formateurs contenue dans s1 dans l'ordre croissant
//   %2 est remplacé par hello
//   %3 est remplacé par salut
//   s2 == "salut et 42 ou 42 et hello"
//   s1 n'est pas modifié
QString s2 = s1.arg("hello","salut");
```

Cet outil est d'autant plus puissant que l'on peut faire du formatage récursif : l'on peut ajouter des formateurs lors du remplacement d'un formateur.

```
QString  s = "%1";
//%1 est remplacé par "formatage récursif : %1 %2"
//   s1 == "formatage récursif : %1 %2"
//   s1 est composé par deux nouveau formateurs.
```

```
QString  s1 = s.arg("formatage recurssif : %1  %2");
```

Comment convertir un nombre en chaîne de caractère ?

Auteurs : Benjamin Poulain ,

QString fournit la méthode statique number() pour convertir des nombres:

```
QString cinquante = QString::number(50);
QString unTier = QString::number(0.33333);
```

Avec QString::number(), il est aussi possible de spécifier la base pour les entiers, et le format et la précisions pour les réels. La méthode statique QString::number() possède un équivalent pour convertir un nombre dans un objet existant: QString::setNum(), par exemple:

```
QString cinquante("Ce texte sera supprimé");
cinquante.setNum(50); // cinquante vaut "50"
```

Pour insérer un nombre dans une chaîne de caractère, il est plus simple d'utiliser les fonctionnalités de formatage de QString

lien : FAQ Formatage de texte avec QString ?

lien : FAQ Comment formater les nombres entiers ?

Comment formater les nombres entiers ?

Auteurs : François Jaffré ,

Qt à l'aide de la classe QString peut formater des chaines de caractères. Cela est possible grâce à la méthode sprintf() qui permet de formater les chaines comme en C ou à la méthode agr() qui est plus propre à Qt.

Exemple : On souhaite formater une chaine de la forme Image_00032

avec sprintf()

```
QString str;
int val=32;
str = str.sprintf("Image_%05d" , val ); // str == "Image_00032"
```

avec arg()

```
QString str = "Image_%1";
int val=32;
str = str.arg(val, 5 , 10 , QChar('0') ); // str == "Image_00032"
```

lien : QString::sprintf

lien : QString::arg

Conversion QString vers nombre ?

Auteurs : François Jaffré ,

La classe QString implémente directement des méthodes pour transformer une chaine de caractère en variable numérique. De plus ces méthodes possèdent des paramètres optionnels :

- **Comme il est possible que la conversion soit impossible, un pointeur vers un bool permet de récupérer si la conversion à eu lieu ou non**
- **Pour la conversion vers un des types entiers, il est possible de spécifier la base (2 à 36) .**

Utilisation sans paramètre:

QString -> double
```
QString str="123.36969577";
double number = str.toDouble();//number == 123.36969577
```

Utilisation avec le paramètre qui permet la vérification de la conversion:

QString -> int
```
QString str1="128";
QString str2="Qt128";
bool ok; //Permet de verifier si la conversion a reussi ou non.
int num1 = str1.toInt(&ok); //ok == true et conversion reussi num1 ==128
int num2 = str2.toInt(&ok); //ok == false et conversion échouée num2 == 0
```

Conversion d'une chaine dans une base particulière (Ex:Hexa, Octal...):

Conversion d'une chaine dans une base hexa
```
QString str = "FFFF";
bool ok;//Toujours utile à la verification de la conversion
int num = str .toInt(&ok,16); //ok == true et num == 65535
str == "0xFFFF"; //Les conventions standards du C par exemple ici "0x" pour une chaine hexa sont reconnues.
num = str.toInt(&ok,16); //ok == true et num == 65535
```

Remarque : Le même principe de conversion existe pour les autres types de variables numériques:

- **toLong()**
- **toShort()**
- **toUInt()**
- **toUShort()**
- **toULongLong()**

Comment formater les nombres réels ?

Auteurs : Benjamin Poulain ,

La méthode statique QString::number() , prend des arguments facultatifs qui permettent de formater les nombres réels.

Le premier argument est un caractère qui précise le format, les formats possibles sont:

- **'e': format scientifique avec un e minuscule; exemple: 2.997925e+08**
- **'E': format scientifique avec un e majuscule; exemple: 2.997925E+08**
- **'f': format classique pour les réels "299792458.0"**
- **'g': choisir automatiquement le plus concis entre 'e' et 'f', c'est le mode par défaut**
- **'G': choisir automatiquement le plus concis entre 'E' et 'f'**

```
// vaut "2.997925e+08"
QString::number(299792458.0, 'e');
// vaut "299792458.000000", voir plus loin pour comprendre pourquoi
QString::number(299792458.0, 'f');
```

Le second argument permet de spécifier la précision utilisée. Par défaut, Qt utilise une précision de 6, ce qui explique le résultat de l'exemple précédent.

```
// vaut "299792458.0"
QString::number(299792458.0, 'f', 1);
```

Il est suggéré de toujours préciser le format lorsque l'on veut convertir des réels (que ce soit avec Qt ou autre). Dans le cas contraire, un affichage inattendu peut apparaître lorsque les nombres deviennent très petits ou très grand, ce qui n'apparaît généralement pas dans les tests lors du développement.

Pour insérer un réel dans une chaîne de caractère, il est plus simple la fonctionnalité de formatage de QString

lien : FAQ Formatage de texte avec QString ?
lien : FAQ Conversion QString vers nombre ?

Comment tester si une chaîne de caractère est vide ?

Auteurs : Benjamin Poulain ,

QString fournit la méthode isEmpty() pour savoir si une chaîne est vide:

```
if(texte.isEmpty()){
  ...
}
```

Il est possible de comparer QString avec un tableau de char, ce qui permet aussi de tester si une chaîne est vide:

```
// mauvaise idée
if(texte == ""){
  ...
}
```

Cela fonctionne, mais il y a une conversion implicite de "" en QLatin1String, ce qui est inutile pour une chaîne vide et est légèrement moins performant que l'exemple précédent.

Comment QString gère l'encodage des chaînes de caractères ?

Auteurs : Benjamin Poulain ,

En interne, QString stocke les caractères en Unicode 4.0 sur 16 bits, ce qui permet des opérations rapides quelque soit le type de caractère utilisé.

Pour convertir un chaînes sur 8bits en QString, il est possible de l'encoder grâce aux méthodes statiques fromAscii(), fromLatin1(), fromUtf8(), et fromLocal8Bit(). Par exemple:

```
QString infini = QString::fromUtf8("&#8734;");
```

Par défaut, QString interprète une chaîne C avec l'encodage Latin 1 (ISO 8859-1). Il est possible de changer ce comportement de façon globale grâce à la méthode statique QTextCodec::setCodecForCStrings().

Pour convertir un objet QString en tableau de char, il faut utiliser une des méthodes toAscii(), toLatin1(), toUtf8() ou toLocal8Bit(). Si d'autres encodages sont nécessaires, il suffit d'utiliser QTextCodec qui permet d'encoder à peut prêt n'importe quoi.

Sommaire > Le contenu de Qt4 > QtCore > Thread

Pourquoi ne faut-il pas faire de traitement IHM dans un thread ?

Auteurs : Yan Verdavaine ,

La grande majorité des API graphiques des divers OS ne sont absolument pas thread-safe. Le traitement IHM dans différents threads génère des accès concurrents et donc des crashs de l'application.
C'est pour cela que Qt oblige les traitements IHM dans le thread principal (celui exécutant le main() du programme).
Attention, cette vérification n'est effectuée uniquement par des assert() en compilation debug.
Si vous devez manipuler l'affichage par un thread, utilisez le système de connexion signal/slot

comment est définie l'appartenance aux threads des objets Qt ?

Auteurs : Yan Verdavaine ,

C'est très simple : un objet Qt appartient au thread qui l'a crée. Cela implique tout de même une règle importante : un slot connecté de manière indirecte sera exécuté par l'eventloop du thread auquel l'objet appartient.

Warning : Il est possible de transférer l'appartenance d'un objet entre threads grâce à la méthode moveToThread. Mais ceci est à utiliser avec beaucoup de précaution.

comment utiliser les threads avec Qt ?

Auteurs : Yan Verdavaine ,

Qt fournit plusieurs possibilités pour manipuler des threads :
*** QtConcurrent : un ensemble d'algorithmes simplifiant énormément l'utilisation des threads. Il partage un pool de threads qui s'adaptera à la machine d'exécution.**
*** QThread : c'est un classe qui interface un thread. Elle va créer, stopper, faire exécuter sa méthode run() et autres opération sur une thread.**
*** QThreadPool : pour optimiser la création de threads, il est possible de manipuler un pool de thread avec QThreadPool . Ce pool de threads exécutera des classes héritant de QRunnable et réimplementant la méthode run().**

comment fonctionne QThread ?

Auteurs : Yan Verdavaine ,

QThread est une interface simple permettant la création et manipulation d'un thread. Elle ne peut être instanciée car la méthode run() est virtuelle pure. Il faut donc créer une classe héritant de QThread et qui réimplémente la méthode run(). Voici les principales fonctionnalités à connaître :

- **run() : méthode exécutée par le thread.**
- **exec() : fonction bloquante qui va exécuter une event loop dans le thread. Cette fonction ne peut être appelée que par le run().**
- **quit() : slot qui stoppe l'eventloop du thread.**
- **start() : slot qui lance un thread qui exécute la méthode run().Cette fonction peut prendre en paramètre la priorité donnée au thread.**
- **setPriority() : modifie la priorité d'exécution du thread**
- **wait() : fonction bloquante qui attend la fin de l'exécution. Il est possible de spécifier un timeout().**

Cette classe émet le signal started() lorsqu'un thread est lancé et finished() lorsque le thread est terminé.

Il existe deux façons de l'utiliser :

- **Sans event loop : seuls des objets n'héritant pas de QObject peuvent être utilisés. C'est à vous de gérer la condition de sortie de la fonction run(). Un signal étant thread-safe, il n'y a aucun problème d'en utiliser dans ce cas.**
- **Avec event loop : si le thread doit utiliser des objets héritant de QObject, l'event loop doit être exécutée. Pour cela, il suffit d'utiliser la fonction exec() dans la méthode run().**

Avant d'utiliser une QThread, voici quelques règles à bien comprendre :

- **QThread n'est pas un thread et n'appartient pas au thread. Ses slots ne s'exécutent pas dans le thread qu'elle interface.**
- **Seule la fonction run() devrait être implémentée. Cette fonction run() est similaire au main() du programme principal. Son exécution correspond au moment où le thread existe réellement.**
- **Les QObjects appartiennent au thread qui les créé. Il faut donc créer/supprimer les QObjects utilisés par le thread dans la méthode run().**
- **Un signal étant thread-safe, le thread peut utiliser les signaux de la QThread implémentée.**

Remarque : Qthread fournit aussi un slot nommé terminate() qui termine brutalement le thread. Cette fonction est à éviter le plus possible.

Comment se passe un connect entre thread ?

Auteurs : Yan Verdavaine ,

Par défaut, la connexion entre thread est asynchrone car le slot sera exécuté dans la thread qui possède l'objet receveur. Pour cette raison, les paramètres du signal doivent être copiables. Ce qui implique quelques règles simples :

- **Ne jamais utiliser un pointeur ou une "référence non const" dans les signatures des signaux/slots. Rien ne permet de certifier que la mémoire sera encore valide lors de l'exécution du slot**
- **Si il y a une référence const, l'objet sera copié**
- **Il est préférable d'utiliser des classes Qt car elles implémentent une optimisation de la copies (cf COW)**

Il est possible d'utiliser ses propres classes. Pour cela il faut :

- **Que cette classe ait un constructeur, un constructeur de copie et un destructeur public**
- **Il faut l'enregistrer dans les meta type par la méthode qRegisterMetaType**

Exemple d'utilisation de qRegisterMetaType

```
class A
{
public :
    A() {};
    A( const A&a) {...}
    ~A() {...}

    ...
}

void desConnect()
{
    qRegisterMetaType <A>("A");
   //connection entre deux objets de deux threads
    connect( unObjet, SIGNAL( unSignal(const A &), unObjetDansUneAutreThread, SLOT(monSlot(const
 A &)));
    connect( unObjet, SIGNAL( unSignal2(A ), unObjetDansUneAutreThread, SLOT(monSlot2(A )));
}
```

Contrairement aux slots, les signaux sont thread safe et peuvent donc être appelé par n'importe quel thread.

Comment manipuler un mutex ?

Auteurs : Yan Verdavaine ,

Pour protéger des données partagées entre threads, Qt fournit la classe QMutex. Cette classe fourni une base :

- **lock : bloque le mutex**
- **tryLock : essaie de bloquer le mutex. Possibilité de mettre un timeout**
- **unlock : libère le mutex**

Afin de simplifier sa manipulation, Qt fournit la classe QMutexLocker basée sur le pattern RAII qui permet de manipuler correctement le mutex et éviter certains problèmes (un thread qui essaye de bloquer deux fois un mutex, un mutex non débloqué suite à une exception,...). QMutexLocker va bloquer le mutex lors de sa création et le libérer lors de sa destruction. Il permet aussi de libérer (unlock()) et de rebloquer (relock()) le mutex.

QMutexLocker vs à la main

```
QMutex lock;
QMutex lock2;

void f()
{
     //locker bloque le mutex lock
     QMutexLocker locker(lock);
     // on bloque le mutex lock2
     lock2.lock();

    //fonction qui génère un exeption
     uneFonction();

     // fonction non appelée suite à l'exception
     lock2.unlock();

    // locker est détruit et va libérer lock
    // malheureusement lock2 est toujours bloqué
}
```

Lorsqu'une ressource est partagée entre plusieurs threads, ces threads ont le droit d'accéder parallèlement à la ressource uniquement si ils ne font que des accès en lecture. Ainsi, pour optimiser les accès, Qt fournit QReadWriteLock qui est un autre mutex beaucoup plus adapté. Contrairement à QMutex, QReadWriteLock va différencier les 'lock()' en lecture et écriture :

Mutex bloqué en lecture :
- **si un thread essaye de bloquer le mutex en lecture : aucune attente, le thread peut accéder à la ressource**
- **si un thread essaye de bloquer le mutex en écriture : le thread attend la libération du mutex**

Mutex bloqué en écriture :
- **dans les deux cas le thread attend la libération du mutex**

Comme QMutex, cette classe fournit une base :

- **lockForRead() : bloque le mutex en lecture**
- **tryLockForRead() : essaie de bloquer le mutex en lecture. Possibilité de mettre un timeout()**
- **lockForWrite() : bloque le mutex en écriture**
- **tryLockForWrite() : essaie de bloquer le mutex en écriture. Possibilité de mettre un timeout()**

- **unlock() : libère le mutex**

De la même manière que QMutexLocker, Qt fournie deux classes qui simplifient la manipulation de ce mutex

- **QReadLoker : manipulation du mutex en lecture**
- **QWriteLoker : manipulation du mutex en écriture**

Comment faire une pause dans une QThread ?

Auteurs : Yan Verdavaine ,

Pour diverse raison il peut être intéressant de mettre en pause l'exécution d'un thread sur une durée déterminé. Pour cela la classe QThread fournie plusieurs méthodes static :

- **sleep : pause la thread pendant n secondes**
- **msleep : pause la thread pendant n millisecondes**
- **usleep : pause la thread pendant n micro secondes**

Comment effacer un fichier ?

Auteurs : François Jaffré ,

Pour cela il faut utiliser les méthodes suivantes de QFile :

- **bool QFile::remove() -> Permet d'effacer le fichier référencé par l'objet QFile**
- **static bool QFile::remove (const QString & fileName) -> Permet d'effacer le fichier "fileName"**

Première methode

```
//On réference l'objet file au fichier
QFile file("C:/existe.txt");
//Efface le fichier "existe.txt" et renvoie true ou false si l'operation à bien réussi
bool valid = file.remove();
```

Deuxième méthode

```
//Efface le fichier "existe.txt" et renvoie true ou false si l'operation à bien réussi
bool valid = QFile::remove("C:/existe.txt");
```

lien : QFile::remove

lien : static QFile::remove

Comment vérifier si un fichier existe ?

Auteurs : François Jaffré ,

Pour cela il faut utiliser les méthodes suivantes de QFile :

- **bool QFile::exists() const -> Permet de verifier si le fichier pointer par l'objet QFile existe.**
- **static bool QFile::exists(const QString & fileName) -> Permet de vérifier si le fichier "fileName" existe.**

Première méthode

```
//On réference l'objet file au fichier
QFile file("C:/existe.txt");
//Si valid == true le fichier existe
bool valid = file.exists();
```

Deuxième méthode

```
//Si valid == true le fichier existe
bool valid  = QFile::exists("C:/existe.txt");
```

lien : static QFile::exists

lien : QFile::exists

Comment copier un fichier ?

Auteurs : François Jaffré ,

Pour cela QFile met à notre disposition les méthodes suivantes :

- **bool QFile::copy (const QString & newName) -> Permet la copie du fichier contenu dans l'objet QFile.**
- **static bool QFile::copy (const QString & fileName, const QString & newName) -> Permet la copie du fichier "fileName" en "newName".**

Remarque : Ces deux méthodes renvoient true si la copie s'est bien passée et renvoie false dans le cas contraire. Si le fichier spécifié "newName" existe déjà la copie sera annulée et la méthode renverra false.

Utilisation de la première méthode

```
//On réference l'objet file au fichier
QFile file("C:/copie.txt");
//On copie le fichier copie.txt vers copieNext.txt
bool valid = file.copy("C:/copieNext.txt");
```

Utilisation de la deuxième méthode

```
//On copie le fichier copie.txt vers copieNext.txt
bool valid  = QFile::copy ("C:/copie.txt", "C:/copieNext.txt");
```

lien : QFile::copy

lien : QFile::copy static

Comment récupérer le chemin des répertoires spéciaux ?

Auteurs : François Jaffré ,

Qt à partir de sa version 4.4 permet avec la classe QDesktopServices de récupérer les différents chemins des répertoires spéciaux tels que Movies, Pictures, Temp, etc. C'est la méthode statique storageLocation() qui prend en argument une énumération de type StandardLocation qui permet d'effectuer cela.
Exemple :On souhaite récupérer le répertoire Pictures de l'utilisateur.

```
QString PicturesPath = QDesktopServices::storageLocation(QDesktopServices::PicturesLocation);
```

La liste de tous les répertoires pouvant être récupérés est ici et correspond à l'énumération StandardLocation.

Remarque : Si vous utilisez l'option QDesktopServices:: DataLocation il est impératif que vous ayez défini préalablement le nom de l'application et le nom de l'organisation à l'aide des méthodes setApplicationName() et setOrganizationDomain() de la classe QCoreApplication.

lien : QCoreApplication

lien : QDesktopServices::storageLocation

Comment sélectionner un nom de fichier ou répertoire à partir d'une boite de dialogue ?

Auteurs : François Jaffré ,

Qt permet à l'aide de la classe QFileDialog une utilisation très simple de ces différentes boites de dialogue à l'aide de méthodes static:

- **QString getOpenFileName() -> Permet la récupération du chemin d'un fichier existant selectionné par l'utilisateur**
- **QStringList getOpenFileNames() -> Permet la récupération du chemin de plusieurs fichiers existants selectionnés par l'utilisateur**
- **QString getSaveFileName () -> Permet la récupération du chemin d'un de fichier non existant dans le but d'être sauvegardé**
- **QString getExistingDirectory() -> Permet la récupération d'un répertoire sélectionné par l'utilisateur**

Remarque : toutes ses méthode fonctionnent sur le même principe :

Exemple

```
/
*On souhaite sélectionner un fichier de type MP3 à l'aide d'une boite de dialogue avec les contraintes suivantes
*-Elle doit avoir comme label "Open mp3 file"
*-Elle doit s'ouvrir à partir du répertoire C:
*-
Elle doit pouvoir filtrer les fichiers ayant l'extension .mp3 et aussi permettre de voir les fichiers ayant n'im

//Ouverture de la boite de dialogue qui est modale
QString myOpenFile = QFileDialog::getOpenFileName(this, tr("Open mp3 file"),tr("C:\\"),
 tr("MP3 files (*.mp3);;All Files (*.*)"));
//On vérifie que l'utilisateur a bien sélectionné un fichier
//Si la chaine est vide c'est que l'utilisateur a cliqué sur annuler
if(myOpenFile.isEmpty())
{
    return;
}
//Un fichier à été sélectionné par l'utilisateur
else
{
    ...
}
```

lien : QFileDIalog

Comment lister les fichiers d'un répertoire et de ses sous répertoires ?

Auteurs : François Jaffré ,

Qt fournit depuis sa version 4.3 la classe QDirIterator qui permet la navigation entre répertoires. Cette classe possède plusieurs surcharges du constructeur permettant de préciser le type de navigation que l'on souhaite entre les répertoires. La navigation dans les répertoires se fait surtout à l'aide de deux méthodes :

- **bool QDirIterator::hasNext () Retourne true tant qu'on n'est pas à la fin de l'arborescence.**
- **QString QDirIterator::next () Fait avancer l'itérateur à la prochaine entrée et renvoie le chemin absolu de celle-ci.**

Récupération des fichiers mp3 et avi se trouvant dans un répertoire et ses sous-répertoires.

```
// On sélectionne le répertoire à partir duquel on va rechercher les fichiers avi et mp3.
QString selectDir = QFileDialog::getExistingDirectory
(
    this,
    tr("Ouvrir un répertoire"),
    "",
    QFileDialog::ShowDirsOnly| QFileDialog::DontResolveSymlinks
);

// On remplit une QStringList avec chacun des filtres désirés ici "*.mp3" et "*.avi".
QStringList listFilter;
listFilter << "*.avi";
listFilter << "*.mp3";

// On déclare un QDirIterator dans lequel on indique que l'on souhaite parcourir un répertoire et ses sous-répertoires.
// De plus, on spécifie le filtre qui nous permettra de récupérer uniquement les fichiers du type souhaité.
QDirIterator dirIterator(selectDir, listFilter ,QDir::Files | QDir::NoSymLinks,
 QDirIterator::Subdirectories);

// Variable qui contiendra tous les fichiers correspondants à notre recherche
QStringList fileList;
// Tant qu'on n'est pas arrivé à la fin de l'arborescence...
while(dirIterator.hasNext())
{
    // ...on va au prochain fichier correspondant à notre filtre
    fileList << dirIterator.next();
}
```

Note 1 : Si on préfère récupérer une liste de type QFileInfoList au lieu d'une QStringList il suffit d'utiliser la méthode QDirIterator::fileInfo ().
Note 2 : Si on souhaite lister uniquement les fichiers d'un répertoire et non ceux de ses sous-répertoires il est préférable d'utiliser la méthode entryInfoList() de la classe QDir.

lien : QDirIterator
lien : QDir::entryInfoList
lien : QFileInfo

Comment écrire dans un fichier texte ?

Auteurs : François Jaffré ,

L'écriture dans un fichier texte se fait comme pour la lecture à l'aide de la classe QTextStream. Celle-ci fourni l'operateur << qui possède énormément de surcharges pour l'écriture dans un fichier. De plus on peut spécifier le jeux de caractères utilisé pour l'écriture avec la méthode QTextStream::setCodec.

Exemple: On souhaite écrire le fichier suivant avec le jeu de caractères UTF-8 :

```
Bonjour,
Nous sommes le 3 avril 2009
```

```
//Création d'un objet QFile
QFile file("Qt.txt");
//On ouvre notre fichier en lecture seul et on verifie l'ouverture
```

```
if (!file.open(QIODevice::WriteOnly | QIODevice::Text))
    return;

//Création d'un objet QTextStream à partir de notre objet QFile
QTextStream flux(&file);
//On choisie le codec correspondant au jeux de caractère que l'on souhaite ici UTF-8
flux.setCodec("UTF-8");
//Ecriture des différentes lignes dans le fichier
flux << "Bonjour," << endl << "Nous sommes le " << 3 << " avril " << 2009 << endl;
```

lien : QTextStream

lien : Liste des jeux de caractère supportés

Comment lire dans un fichier texte ?

Auteurs : haraelendil ,

La classe QTextStream offre des fonctionnalités intéressantes:

```
QString fileName = "fichier.txt";
QFile fichier(fileName);
fichier.open(QIODevice::ReadOnly | QIODevice::Text);
QTextStream flux(&fichier);
```

On peut ensuite lire le fichier de plusieurs façons:
*** Intégralement:**

```
QString tout =flux.readAll();
```

*** Ligne par ligne:**

```
QString ligne;
while(!flux.atEnd())
{
    ligne = flux.readLine();
    //traitement de la ligne
}
```

*** Mot par mot:**

```
QString mot;
while(!stream.atEnd())
{
    stream >> mot;
    //traitement du mot
}
```

La classe QTextStream offre une moyen simple d'écrire ou de lire du texte, d'en extraire ou d'ajouter des lignes, des mots ou des nombres, un QTextStream pouvant fonctionner avec un QIODevice, une QString ou même un QByteArray.
Les opérateurs surchargé << et >> permettent d'opérer sur un QTextStream de façon très simple.
Remarque : Cette classe prend en compte l'encodage du texte.

Mais on peut aussi lire intégralement le contenu d'un fichier sans passer par la classe QTextStream, mais cela nécessite de passer par la classe QByteArray:

```
QString fileName = "fichier.txt";
QFile fichier(fileName);
```

```
fichier.open(QIODevice::ReadOnly);
QByteArray data;

data = fichier.readAll();
QString ligne(data);
```

De même, on peut lire le fichier ligne par ligne:

```
QString fileName = "fichier.txt";
QFile fichier(fileName);
fichier.open(QIODevice::ReadOnly);
QByteArray data;

while(!fichier.atEnd())
{
    data = fichier.readLine();
    QString ligne(data);
    //traitement de la ligne
}
```

lien : QFile
lien : QTextStream

Comment écrire et lire dans un fichier binaire ?

Auteurs : François Jaffré ,

Qt permet facilement d'écrire ou de lire dans un fichier de type binaire. Ceci se fait à l'aide de la classe QFile pour l'ouverture et la fermeture du fichier et de la classe QDataStream pour l'écriture ou la lecture de celui-ci. L'intérêt d'utiliser la classe QDataStream est d'être indépendant de la machine au niveau de l'encodage des octets. Dans les faits cela veut dire qu'un fichier binaire étant écrit à l'aide de la classe QDataStream sous Windows par exemple peut être lu sans problème sous toutes les autres plateformes supportant Qt (Mac, Linux...).

Remarque : par défaut, QDataStream travaille en big-endian.

Ecriture d'un fichier binaire

```
//Utilisation d'un vecteur de qint32 et pas d'int pour préserver la portabilité
QVector<qint32> vec;

//On remplie le vecteur
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
vec.push_back(4);

//Utilisation d'un type qint32 pour préserver la portabilité
qint32 value = 67;

//Initialisation de la QString qui sera enregistrée dans le fichier binaire
QString str = "Cette chaine sera sauvegardée dans un fichier binaire";

QFile file("MyFile.bin");
//Ouverture du fichier en lecture seul
if (!file.open(QIODevice::WriteOnly))
    return;
//Création de notre QDataStream à partir de notre fichier précédemment ouvert
QDataStream binStream(&file);
//On spécifie la version de l'encodage qui va être utilisé par notre objet QDataStream
binStream.setVersion(QDataStream::Qt_4_4);
//On ecrit nos differents objet dans le fichier
binStream << str << value << vec;
```

Ecriture d'un fichier binaire

```
//On ferme le fichier
file.close();
```

Lecture de ce même fichier binaire

```
//Utilisation des mêmes types utilisés lors de l'ecriture
QVector<qint32> vec;
qint32 value;
QString str;

QFile file("MyFile.bin");
//Ouverture du fichier en lecture seul
if (!file.open(QIODevice::ReadOnly))
    return;
//Création de notre QDataStream à partir de notre fichier précédemment ouvert
QDataStream binStream(&file);
/*On spécifie exactement le meme type d'encodage que celui utilisé lors de l'ecriture*/
binStream.setVersion(QDataStream::Qt_4_4);
//Lecture des differents objects contenus dans notre fichier binaire
binStream >> str >> value >> vec;
//Fermeture du fichier
file.close();
```

lien : **QFile**

lien : **QDataStream**

Comment connaitre le chemin du répertoire courant ?

Auteurs : François Jaffré ,

Qt fourni la classe QDir pour manipuler les répertoires. Cette classe permet entre autre de récupérer le chemin du répertoire courant de l'éxecution et de le modifier :

- **QString QDir::currentPath() : chemin complet du répertoire**
- **QDir QDir::current () : QDir renvoie un objet de type QDir pointant sur le répertoire courant.**
- **bool QDir::setCurrent (const QString & path) : modifie le répertoire courant. Retourne true si la modification à réussi.**

Récupération du répertoire courant

```
QString CurrentDir = QDir::currentPath();
```

Spécifier le répertoire courant.

```
QString CurrentDir = "C:/Users/Developpez/Downloads";
QDir::setCurrent(CurrentDir);
```

Remarque :Il est intéressant de préciser que le répertoire courant n'est pas forcement le même que le répertoire ou se trouve l'exécutable.

lien : **QDir**

Comment récupérer le chemin du répertoire de l'exécutable ?

Auteurs : François Jaffré ,

Qt fourni avec la classe QCoreApplication des méthodes static permettant de récupérer facilement des informations comme le chemin du répertoire de l'exécutable.

On exécute une application depuis C:\Users\developpez\Documents\test.exe

```
/*Contient le chemin complet du répertoire de l'exécutable (C:\Users\developpez\Documents)*/
QString MyAppDirPath =  QCoreApplication::applicationDirPath();

/*Contient le chemin complet de l'exécutable (C:\Users\developpez\Documents\test.exe)*/
QString MyAppPath = QCoreApplication::applicationFilePath ();
```

lien : QCoreApplication

Sommaire > Le contenu de Qt4 > QtGui

Fenêtre et transparence ?

Auteurs : Yan Verdavaine ,

Qt permet diverses actions pour jouer avec la transparence d'une fenêtre :

*** Mask : à partir d'une image binaire (QBitmap), il est possible de spécifier quelle partie est visible ou non sur la fenêtre. Cette méthode s'applique à tous les éléments d'une fenêtre top-level. Par exemple la partie d'un bouton qui se trouve dans une zone transparente, ne sera pas dessinée. Pour utiliser un mask il faut créer une QBitmap et utiliser la fonction setMask (définie par QWidget) sur la widget parente.**
Sur certains OS comme windows xp il est obligatoire d'enlever la décoration avec le flag Qt::FramelessWindowHint.

***opacité : permet de rendre translucide toute une fenêtre. Pour cela il faut utiliser la méthode setWindowOpacity (définie par QWidget) sur la widget parente. Cette méthode prend en paramètre une valeur réelle entre 0 (transparent) et 1 (opaque)**

***TranslucentWidget : depuis la version 4.5, il est possible de spécifier que le background de la widget parente soit transparent. Il est ainsi possible de jouer avec la transparence des différents éléments et QPainter pour créer une fenêtre translucide. Contrairement aux deux autres méthodes, ceci ne s'applique qu'au background de la fenêtre parente et les éléments enfants ne sont pas modifiés. Pour utiliser cette option, il faut activer l'attribut Qt::WA_TranslucentBackground sur la widget parente : setAttribute(Qt::WA_TranslucentBackground, true);**
Sur certains OS comme windows xp il est obligatoire d'enlever la décoration avec le flag Qt::FramelessWindowHint.

Remarque : ces trois méthodes peuvent être utilisées en même temps

Voici un exemple vous montrant les différentes possibilités

```
#include <QtGui>

// base
// drag &drop sur la widget pour la déplacer
// bouton fermer pour fermer l'application
// dessine un dégradé RGBA en fond
class BaseWidget : public QWidget
{
    //position dans le repere widget de click gauche
    QPoint p;
    QPolygon polygon;
    QLinearGradient gradient;

public:
    BaseWidget()
    :QWidget
    (
        0,
        //enleve la decoration windows et l'entrée dans la bar de tâche
        Qt::FramelessWindowHint | Qt::SubWindow
    )
    {
        resize(200,200);

        //bouton pour fermer l'application
        QPushButton *b = new QPushButton("fermer",this);
        QLayout *l= new QVBoxLayout(this);
        l->addWidget(b);
        connect(b,SIGNAL(clicked()),qApp,SLOT(quit()));
         //initialisation du polygone
        QVector<QPoint>  points;
        //même suite aléatoire pour chaque polygone
        qsrand(150);
```

Voici un exemple vous montrant les différentes possibilités

```
        const int nbpoints = 100;
        const double r2PI = 2. * 3.14159265;
        for (int i = 0 ; i < nbpoints ; ++i)
        {
            float distance = 80.+20*qrand()/RAND_MAX;
            points << QPoint
                        (
                            100 + distance*cos(r2PI * i/nbpoints),
                            100 + distance*sin(r2PI * i/nbpoints)
                        );
        }
        polygon = QPolygon (points);

        //initialisation du gradient
        gradient.setStart (0. , 0.);
        gradient.setFinalStop (size().width() , 0.);
        {
             //Gradienstops pour varier l'apha et la couleur du gradient
            QGradientStops gs;
            gs  <<QGradientStop(0. , QColor(50 , 0   , 50  , 255))
                <<QGradientStop(.5 , QColor(0  , 255 , 255 , 100))
                <<QGradientStop(1. , QColor(50 , 50  , 0   , 255));

            gradient.setStops (gs);
        }

    }
protected:
    void mousePressEvent   ( QMouseEvent * mouseEvent )
    {
        //sauvegarde du point dans le repère widget lors du click gauche
        if(mouseEvent->buttons() == Qt::LeftButton)  p = mouseEvent->pos();

    }
    void mouseMoveEvent  ( QMouseEvent * mouseEvent )
    {
        //repositionne la fênetre en fonction de la postion de la souris
        if(mouseEvent->buttons() & Qt::LeftButton)     move( mouseEvent->globalPos() - p);
    }
    protected:
    void paintEvent(QPaintEvent * )
        {
            QPainter p(this);
            p.setPen(Qt::NoPen);
            //utilisation un gradient linéaire entre deux coin de la fenetre
            p.setBrush(gradient);
            p.drawPolygon(polygon);
        }
};


//Widget avec un fond translucide
class TranslucideWidget : public BaseWidget
{
    //position dans le repere widget de click gauche
    QPoint p;
public:
    TranslucideWidget()
    {
        //windows avec fond translucide
        setAttribute(Qt::WA_TranslucentBackground, true);
    }
};

//Widget utilisant un mask
class MaskWidget : public BaseWidget
{
```

Voici un exemple vous montrant les différentes possibilités

```
public:
    MaskWidget()
    {
        //création d'un mask sous forme de chessboard.
        QBitmap mask(5,5);
        {
            QPainter p(&mask);
            p.fillRect(mask.rect(),Qt::color0);
            p.setPen(Qt::color1);
            for (int i = 0 ; i < mask.height() ; ++i)
            for(int j = 0 ; j< mask.width() ; ++j)
            if( !(i%2 ^ j%2))
            p.drawPoint(j,i);
        }
        //retaille le mask a la taille de la fenetre
        mask = mask.scaled(size());
        //application du mask sur toute la widget
        setMask(mask);
    }
};

//Widget utilisant windowOpacity
class OpacityWidget : public BaseWidget
{
public:
    OpacityWidget()
    {
        //modifie l'opacite de la fenetre
        setWindowOpacity (.5);
    }
};


class CompositionWidget : public BaseWidget
{
public:
    CompositionWidget()
    {
        //windows avec fond translucide
        setAttribute(Qt::WA_TranslucentBackground, true);

        //création d'un mask sous forme de chessboard.
         //création d'un mask sous forme de chessboard.
        QBitmap mask(5,5);
        {
            QPainter p(&mask);
            p.fillRect(mask.rect(),Qt::color0);
            p.setPen(Qt::color1);
            for (int i = 0 ; i < mask.height() ; ++i)
            for(int j = 0 ; j< mask.width() ; ++j)
            if( !(i%2 ^ j%2))
            p.drawPoint(j,i);
        }
        //retaille le mask a la taille de la fenetre
        mask = mask.scaled(size());
        //application du mask sur toute la widget
        setMask(mask);


        //modifie l'opacite de la denetre
        setWindowOpacity (.5);
    }
};

int main(int argc, char **argv)
{
    QApplication app(argc, argv);
```

Voici un exemple vous montrant les différentes possibilités

```
    TranslucideWidget translucideWidget;
    translucideWidget.show();
    translucideWidget.move(50,50);

    MaskWidget maskWidget;
    maskWidget.show();
    maskWidget.move(300,50);

    OpacityWidget opacityWidget;
    opacityWidget.show();
    opacityWidget.move(50,300);

    CompositionWidget compositionWidget;
    compositionWidget.show();
    compositionWidget.move(300 , 300);

    return app.exec();
}
```

Comment changer de style de fenêtre ?

Auteurs : Matthieu Brucher , Benjamin Poulain ,

Par défaut, Qt utilise le style natif du système d'exploitation. Si vous voulez modifier le style, vous pouvez le faire au niveau de chaque widget à l'aide de QWidget::setStyle() ou au niveau de l'application complète avec QApplication::setStyle().

Voici par exemple comment utiliser le style "Plastique" pour toute l'application:

```
    QApplication::setStyle(new QPlastiqueStyle);
```

Remarque : il est possible de créer un style par son nom avec QStyleFactory::create() ou la fonction static QApplication::setStyle ().

Créer un style complet est généralement une tâche difficile. Heureusement Qt fournit un mécanisme simple pour personnaliser les styles: les CSS (Cascading Style Sheets). Avec CSS pour Qt, il est possible de personnaliser le design d'une application à l'aide d'une syntaxe simple dérivée des CSS de HTML. Cela peut se faire au niveau de designer, ou comme pour les styles classiques, au niveau des widgets ou de l'application.

Voici une exemple d'une feuille CSS de Qt:

```
QWidget {
    border: 1px solid gray;
    border-radius: 10px;
    padding: 1px 10px 1px 10px;
    margin: 3px;
}

QLineEdit:enabled {
    background: green;
}
QLineEdit:!enabled {
    background: grey;
}
QComboBox::drop-down {
    subcontrol-origin: padding;
    subcontrol-position: top right;
    width: 50px;
```

Voici une exemple d'une feuille CSS de Qt:

```
        border-radius: 0;
    }
```

lien : Un exemple de création et d'utilisation de style sur le site de Trollech
lien : Comment connaître les styles disponibles ?
lien : QApplication::setStyle (const QString & style)
lien : QStyleFactory

Comment connaître les styles disponibles ?

Auteurs : Benjamin Poulain ,

Les styles disponibles dépendent de la plateforme et des plugins. La classe QStyleFactory permet de lever toute ambiguïté en fournissant la liste des styles disponibles à l'exécutions. La méthode statique QStyleFactory::keys retourne une liste de nom de style qu'il est possible de charger. À partir des noms de cette liste, il est possible de charge les styles avec la méthode statique QStyleFactory::create. Notez qu'en général il est déconseillé de changer de style car l'application perd son intégration avec le reste du système.

Comment dessiner dans un QWidget ?

Auteurs : Yan Verdavaine ,

Lorsqu'un widget doit être dessiné ou redessiné, la fonction protégée QPaintEvent est appelée. Pour se dessiner, la majorité des widgets de Qt utilise un QPainter lors de l'appel de cette fonction. Il est donc naturel de suivre cette logique.

L'évènement QPaintEvent passé en paramètre indique la zone à redessiner. Cette zone est donnée sous forme de rectangle QPaintEvent::rect ou sous forme quelconque QRegion. Ces informations peuvent être utilisées pour optimiser l'affichage du widget.

La classe QPainter est un outil qui permet de dessiner sur toutes les classes graphique de Qt : QCustomRasterPaintDevice, QGLFramebufferObject, QGLPixelBuffer, QImage, QPicture, QPixmap, QPrinter, QSvgGenerator, et QWidget.

Cette classe utilise d'autres outils de Qt. Les plus importants à mon avis sont :

- **QPen : caractérise le contour d'une forme (ligne, point, contour d'une forme...)**
- **QBrush : caractérise l'intérieur d'une forme (intérieur d'un rectangle...)**

Voici un exemple :

dessiner sur une widget

```
#include <QtGui>

class MyTest : public QWidget
{
public:
    MyTest ( QWidget * parent = 0, Qt::WindowFlags f = 0 ) : QWidget  (parent,f){};
    void paintEvent ( QPaintEvent * event )
    {
        //creation d'un QPainter
        //QPen : aucun contour
```

dessiner sur une widget

```
        //QBrush : aucun remplissage
        QPainter painter(this);

        //On definie au QPainter un stylo rouge de taille 4
        QPen pen;
        pen.setColor (Qt::red);
        pen.setWidth (4);
        painter.setPen(pen);

        //On dessine un rectangle
        //QPen : rouge de taille 4
        //QBrush : aucun remplissage
        painter.drawRect (10, 10, 80, 80);

        //on definie au QPainter un pinceau vert utilisant le pattern SOLID
        QBrush brush(Qt::SolidPattern);
        brush.setColor(Qt::green);
        painter.setBrush(brush);

        //on dessine un rectangle
        //QPen : rouge de taille 4
        //QBrush : remplissage vert avec le pattern SOLID
        painter.drawRect (15, 15, 70, 70);


        //on definie au QPainter stylo bleu  de taille 8
        pen.setColor (Qt::blue);
        pen.setWidth (8);
        painter.setPen(pen);

        //dessine une ligne
        //QPen : bleu de taille 8
        //QBrush : n'est pas utilise pour une ligne
        painter.drawLine(0, 0, 100, 100);
    }
};

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    MyTest win;
    win.resize(100, 100);
    win.show();
    return app.exec();
}
```

Pourquoi je n'arrive pas à dessiner sur n'importe quels widgets ?

Auteurs : Yan Verdavaine ,

Certaines widgets sont basées sur QAbstractScrollArea dont le but est de permettre l'affichage d'une widget plus grande que sa zone d'affichage. En ajoutant des scrollbar par exemple. Appliquer un painter directement sur celle-ci n'aura pas l'effet souhaité. Cette classe implémente la méthode viewport() qui permet d'accéder à la widget qui est réellement affichée. Il faut donc appliquer le painter sur celle-ci.

On constate ainsi trois étapes lors de la redéfinition du paintevent :

- **pre-dessin :le dessin en arrière plan**
- **dessin originel : le dessin d'origine.**
- **post-dessin : le dessin au premier plan.**

Les widgets concernées sont QAbstractItemView, QGraphicsView, QMdiArea, QPlainTextEdit, QScrollArea, QTextEdit, QTextBrowser, QColumnView, QHeaderView, QListView, QTableView, QTreeView,QHelpContentWidget, QTreeWidget, QTableWidget,QHelpIndexWidget, QListWidget, QUndoView

Dessiner avec un QListView

```
#include <QtGui>

class myListView : public QListView
{
public:
    myListView()
    {}

protected:
    void paintEvent(QPaintEvent *event)
    {
       /*pre-dessin*/
        if (this->viewport())
        {
            QPainter painter(this->viewport());
            painter.setBrush(QBrush(Qt::green, Qt::SolidPattern));
            painter.drawRect(QRect(0,0,50,50));
        }

        /*dessin originel*/
        QListView::paintEvent(event);

        /*post-dessin*/
        if (this->viewport())
        {
            QPainter painter(this->viewport());
            painter.setBrush(QBrush(Qt::red, Qt::SolidPattern));
            painter.drawRect(QRect(25,25,50,50));
        }
        event->accept();
    }
};

int main(int argc, char **argv)
{
    QApplication app(argc, argv);
    QStringList list;
    list<<"dessiner"<<"avec"<<"un"<<"QListView";

    myListView w;
    w.setModel(new QStringListModel(list,&w));
    w.show();
    return app.exec();
}
```

Remarque: Comme le traitement pre-dessin se dessine en arrière plan, il peut être totalement effacé par le dessin originel ou le post-dessin

Comment ajouter un lien vers une page HTML ?

Auteurs : Yan Verdavaine ,

La façon la plus simple est d'utiliser un QLabel et ses possibilités de Rich text (http://doc.trolltech.com/4.3/richtext.html). Pour cela il suffit :

- **D'autoriser l'ouverture vers une page web**

- **De remplir le label avec un petit morceau en HTML : *XXXXX* . où XXX peut être**
  - **une phrase simple.**
  - **une phrase en HTML : * ... ***
    - **size : facultatif. Taille de la police**
    - **family : facultatif. Police à utiliser dans l'ordre de préférence**
    - **color : couleur du texte**
    - **Il est possible de compléter la phrase en utilisant les balises de formatage de texte comme gras(<b>...<\b>), italique (*...*)...**

- **une image en HTML : *<img src='MON_IMAGE' height ='H' width= 'W'>***
  - **src : path de l'image à afficher. Peut se situer dans les ressources**
  - **height : facultatif. Hauteur de l'image**
  - **width : facultatif. Largeur de l'image**

exemple avec un lien text et un lien image

```
#include <QtGui>

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QLabel label;
    //on ecrit un petit bout de html
    label. setText (    "La doc de"
                        //lien par un text HTML
                        "<a href='http://doc.trolltech.com/4.3/qlabel.html'>"
                                "<font size='15' family='Arial, Helvetica, sans-
serif' color='green'><b> Qt </b></font>"
                        "</a>"
                        "est voici "
                        //lien par une image
                        "<a href='http://qt.developpez.com/faq/'>"
                            "<img src='FAQ-Qt.gif'>"
                        "</a>");
    //on autorise l'ouverture du lien vers le viewer par defaut
    label.setOpenExternalLinks ( true );
    label.show();
    return app.exec();
}
```

Comment créer une page d'accueil ?

Auteurs : Shugo78 ,

Qt propose la classe QSplashScreen pour simplifier la gestion des pages de démarrage. Une page de démarrage est une image affichée lors du démarrage d'un programme, pour masquer un démarrage lent ou tout simplement par marketing.

Remarque :

- **Cette page se fermera si l'on clique dessus**
- **QSplashScreen est lancé avant la boucle d'événement de l'application, il est donc préférable d'appeler QApplication::processEvents() de temps en temps pour traiter les événements en cours.**

```
#include <QtGui>
```

```
#include <windows.h> //Sleep


int main (int argc, char** argv)
{
    QApplication app (argc, argv);

    // Création de la page de demarrage
    QSplashScreen splash;
    //La page sera devant toute les fenetres
    splash.setWindowFlags ( Qt::WindowStaysOnTopHint);
    splash.setPixmap(QPixmap ("c:/figure1.jpg"));
    splash.show();

    //On ecrit sur l'image l'etape en cours
    splash.showMessage (QObject::tr ("Etape 1"), Qt::AlignRight | Qt::AlignTop, Qt::white);
    //On simule un traitement de durée 2s
    // /!\ pour Windows
    Sleep(2000);
     //On lance un traitement sur les evenements
    app .processEvents();

    //On change l'image
    splash.setPixmap(QPixmap ("c:/figure2.jpg"));
    //On ecrit sur l'image l'etape en cours
    splash.showMessage (QObject::tr ("Etape 2"), Qt::AlignRight | Qt::AlignTop, Qt::white);
    //On simule un traitement de durée 2s
    // /!\ pour Windows
    Sleep(2000);
    //On lance un traitement sur les evenements
    app .processEvents();

    QWidget w;
    w.show();
    //La page se fermera une fois le widget affiché
    splash.finish (&w);

    return app.exec ();
}
```

Comment afficher une image dans mon interface ?

Auteurs : IrmatDen ,

La méthode la plus simple est d'utiliser un QLabel conjointement à un QPixmap.

```
#include <QtGui>

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QLabel label;
    //on donne l'image au label
    label.setPixmap(QPixmap("c :/image.png"));
    label.show();
    QObject::connect(&label, SIGNAL(clicked()), &app, SLOT(quit()));
    return app.exec();
```

```
}
```

Comment dégeler simplement une ihm ?

Auteurs : Yan Verdavaine ,

Si une action longue est exécuté (ex :parcours récursif de répertoire, copie de plusieurs fichiers,...) par l'ihm, celle ci va se geler et ne plus répondre... Pour remédier à cela il est souvent préférable d'utiliser une thread.

Mais il existe une alternative intéressante lorsque ce traitement n'a vraiment aucun intérêt à être mis dans un thread : QCoreApplication (et donc QApplication) possède la méthode static processevents permettant l'exécution dune partie ou de tous les events de l'eventloop.

Simulation d'un traitement long. L'un utilise le processevents l'autre non

```
#include <QtGui>
#include <cmath>
#include <ctime>

//Simule une action
double action_simulation(int nb =1000)
{
    double somme(0.);
    for (unsigned int j=0;j<nb;++j) somme+=sqrt(static_cast<double>(j));
    return somme;
}


class MyQPushButton : public QPushButton
{
    public :
    MyQPushButton(const QString & text, QWidget * parent = 0)
        :QPushButton (text,parent)
    {
        resize(75, 30);
        setFont(QFont("Times", 12, QFont::Bold));
    };
protected:
    //action lorsque le bouton est relache
    void mouseReleaseEvent ( QMouseEvent * event )
    {
        //progress bar infinie
        QProgressDialog mybar( "Wait ...", 0, 0, 0);
        mybar.setWindowModality(Qt::ApplicationModal);
        mybar.show();

        //simule une action lente de 5s
        double somme(0.);
        clock_t endwait;
        endwait = clock () + 5* CLOCKS_PER_SEC ;
        while (clock() < endwait)
        {
          somme+= action_simulation() ;
        }
        qDebug()<<somme;
    }
};


class MyQPushButton2 : public QPushButton
{
public :
    //createur.
    //text : text du boutton
```

Simulation d'un traitement long. L'un utilise le processevents l'autre non

```
    MyQPushButton2(const QString & text, QWidget * parent = 0)
    :QPushButton (text,parent)
    {
        resize(75, 30);
        setFont(QFont("Times", 12, QFont::Bold));
    };
protected :
    //action lorsque le bouton est relache
    void mouseReleaseEvent ( QMouseEvent * event )
    {
        //progress bar infinie
        QProgressDialog mybar( "Wait ...", 0, 0, 0);
        mybar.setWindowModality(Qt::ApplicationModal);
        mybar.show();

        //simule une action lente de 5s
        double somme(0.);
        clock_t endwait;
        endwait = clock () + 5 * CLOCKS_PER_SEC ;
        while (clock() < endwait)
        {
            QCoreApplication::processEvents();
            somme+= action_simulation() ;
        }
         qDebug()<<somme;
    }
};

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    QWidget w;
    QVBoxLayout * vl = new QVBoxLayout;
    {
        //boutton : L'appliquation se bloque pendant le traitement
        MyQPushButton * bouton1 = new MyQPushButton("Sans QCoreApplication::processEvents()");

        //boutton : L'appliquation ne se bloque pas pendant le traitement
        MyQPushButton2 * bouton2 = new MyQPushButton2("Avec QCoreApplication::processEvents()");

        vl->addWidget(bouton1);
        vl->addWidget(bouton2);
    }
    w.setLayout(vl);
    w.show();
    return app.exec();
}
```

Comment afficher un gif animé ?

Auteurs : Yan Verdavaine ,

Comme un Gif animé est quelque chose de non statique, on ne peut pas afficher l'animation avec une QImage ou une QPixmap. Pour cela, Qt fournie une classe dédiée aux vidéos : QMovie.

La méthode la plus simple est de créer un QMovie avec le gif et de l'associer à un label.

```
#include <QtGui>

int main(int argc, char* argv[])
{
```

```
    QApplication app(argc, argv);
    QLabel w;

    //On cree une video. Ici c'est un gif
    QMovie movie("c:/Mickey-11.gif");
    //On l'associe a un label
    w.setMovie (&movie);
    //On lance la video
    movie.start ();

    w.show();
    return app.exec();
}
```

Il faut tout de même préciser que la compilation du plugin Gif est nécessaire pour que ce code marche correctement.

Comment avoir une icône animée ?

Auteurs : Yan Verdavaine ,

Il peut être intéressant d'avoir des icônes qui évoluent dans le temps. Par exemple, une icône dans la barre des tâches qui donne l'avancement d'un traitement. Pour animer une QIcon, il suffit de remplacer l'image à utiliser à l'instant t.

Voici trois méthodes simples:

1- Un fichier image animé (gif, mng,...) et QMovie :

- **QMovie émet un signal à chaque changement d'image.**
- **Il suffit de connecter un slot qui va récupérer l'image courante de QMovie**
- **Icône peut être animée par la lecture de l'animation.**
- **Possibilité de contrôler l'avancement de l'animation avec la méthode jumpToFrame**

Le bouton permet de charger une image animée.

```
#include <QtGui>

class MyWindow : public QWidget
{
    Q_OBJECT
    //lecteur d'image animee
    QMovie      m_movie;
    //objets qui utilise une icone
    QPushButton *m_but;
    QSystemTrayIcon m_sysTray;
public :
    MyWindow( )
    {
        //layout et le boutton seront re-parente
        //par le setLayout
        QLayout * l = new QVBoxLayout;
        {
            m_but = new QPushButton;
            l->addWidget(m_but);
        }
        setLayout(l);
        //connect l'evenement updated pour changer d'image dans les icones
        connect(&m_movie, SIGNAL(updated ( const QRect & )),this,SLOT(nouvelleIcone()));
        //boutton : permet de prendre une autre images animee
        connect(m_but, SIGNAL(clicked( )),this,SLOT(nouvelleImageAnimee()));

        //ajout dans l'event loop
        QTimer::singleShot(0,this,SLOT(chargerImageAnimee()));
    }
```

Le bouton permet de charger une image animée.

```
private slots:
    //selection d'une image animee et lecture par le QMovie
    void chargerImageAnimee()
    {
        QString
 s =QFileDialog::getOpenFileName(this,"choisir une image animee",QString(),"*.gif *.mng");
        if (!s.isEmpty())
        {
           m_movie.stop();
           m_movie.setFileName(s);
           m_movie.start();
        }

    }

    //modifie l'icone pour le boutton, le systray et le titre de la fenetre
    void nouvelleIcone()
    {
        QIcon icon(m_movie.currentPixmap());
        //modifie l'icone du boutton
        m_but->setIcon(icon);
        //modifie l'icone du systray
        m_sysTray.setIcon(icon);
        m_sysTray.show();
        //modifie l'icone de la fenetre
        setWindowIcon(icon);
    }

};
//astuce pour ne pas avoir plusieurs fichiers
#include "main.moc"

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    MyWindow w;
    w.show();
    return app.exec();
}
```

2- Une liste d'image :

- **A chaque évènement de changement d'image, on choisi l'image dans la liste**
- **Un timer permet de faire évoluer l'animation de manière constante**

Charge les images d'un répertoire dans une liste.

```
#include <QtGui>

class MyWindow : public QWidget
{
    Q_OBJECT
    //liste d'image
    QList<QPixmap>     m_lImages;
    //objets qui utilise une icone
    QPushButton *m_but;
    QSystemTrayIcon m_sysTray;
public :
    MyWindow( )
    {
        //layout et le boutton seront re-parente
        //par le setLayout
        QLayout * l = new QVBoxLayout;
        {
             /*slider qui permet d'emuler l'avancement*/
            QSlider * slider = new QSlider(Qt::Horizontal);
            slider->setRange ( 0,100 );
```

Charge les images d'un répertoire dans une liste.

```
            connect (slider,SIGNAL(valueChanged  ( int )),this,SLOT (nouvelleIcone(int)));
            l->addWidget(slider);


            m_but = new QPushButton;
            l->addWidget(m_but);
        }
        setLayout(l);

        //boutton : permet de prendre une autre images animee
        connect(m_but, SIGNAL(clicked( )),this,SLOT(chargerRepertoire()));

        //ajout dans l'event loop
        QTimer::singleShot(0,this,SLOT(chargerImageAnimee()));
    }
private slots:
    //charger les image d'un repertoire dans une liste
    void chargerRepertoire()
    {
        QString s =QFileDialog::getExistingDirectory(this,"choisir une image animee",QString());
        m_lImages.clear();
        QDir d(s);
        QStringList filters;
        filters << "*.jpg" << "*.png" << "*.gif";
        //pour chaque fichier image du repertoire
        //on lie l'image et on l'ajoute dans la liste
        foreach(QString file,d.entryList (filters,QDir::Files))
        {
            QPixmap tmp(d.absolutePath()+"/"+file);
            if ( !tmp.isNull ())
                m_lImages<<tmp;
        }

    }

    //modifie l'icone pour le bouton, le systray et le titre de la fenêtre
    void nouvelleIcone(int pourcent)
    {
        if (m_lImages.size() == 0)
            return;

        //on choisie l'image
        QIcon icon(m_lImages[pourcent%m_lImages.size()]);
        //modifie l'icone du boutton
        m_but->setIcon(icon);
        //modifie l'icone du systray
        m_sysTray.setIcon(icon);
        m_sysTray.show();
        //modifie l'icone de la fenetre
        setWindowIcon(icon);
    }

};
//astuce pour ne pas avoir plusieurs fichiers
#include "main.moc"

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    MyWindow w;
    w.show();
    return app.exec();
}
```

3- Un QPainter :

- **A chaque évènement qui demande un changement de l'icône , on créé une nouvelle image avec un QPainter**

- **La plus compliqué à mettre en place mais la plus puissante.**
- **La création d'une nouvelle icône peut mettre du temps**

Compose une image à partir de deux images

```
#include <QtGui>

class myWindows : public QWidget
{
    Q_OBJECT
public :
    myWindows()
    {
        QGridLayout * l = new QGridLayout;
        {
            /*slider qui permet d'emuler l'avancement*/
            QSlider * slider = new QSlider(Qt::Horizontal);
            slider->setRange ( 0,100 );
            connect (slider,SIGNAL(valueChanged  ( int )),this,SLOT (noucelleIcone(int)));
            l->addWidget(slider,0,0,1,3);

            /*bouton pour charge la premiere image*/
            QPushButton * b = new QPushButton("charger Pix1");
            connect(b,SIGNAL(clicked()),this,SLOT(OpenPix1()));
            /*label qui affiche la premiere image*/
            m_Apix1 =new  QLabel;
            l->addWidget(b,1,0);
            l->addWidget(m_Apix1,2,0);

            /*button qui affiche une l'icon calcule*/
            m_AresultIcone = new QPushButton;
            /*label qui affiche l'image clacule*/
            m_Aresult =new  QLabel;
            l->addWidget(m_AresultIcone,1,1);
            l->addWidget(m_Aresult,2,1);

            /*bouton pour charger la deusieme image*/
            QPushButton * b = new QPushButton("charger Pix2");
            connect(b,SIGNAL(clicked()),this,SLOT(OpenPix2()));
            /*label qui affiche la deusieme image*/
            m_Apix2 =new  QLabel;
            l->addWidget(b,1,2);
            l->addWidget(m_Apix2,2,2);
        }
        setLayout(l);

        m_trayIcon.show();
    }


private slots :

    void noucelleIcone(int percent)
    {

        //pixmap avec un fond transparent
        QPixmap resultImage(200,200);
        resultImage.fill(Qt::transparent);
        {
            QPainter painter(&resultImage);

            //precalcule des pourcentages de la hauteur
            float percent_1 = (100.- percent)/100.;
            float percent_2 =  percent/100.;

            //heut de l'image
            QRect target(0, 0, resultImage.width(), percent_1*resultImage.height());
            QRect source(0, 0, m_pix1.width(), percent_1*m_pix1.height());
```

Compose une image à partir de deux images

```
            painter.drawPixmap(target, m_pix1, source);

            //bas de l'image
            QRect target2(0, percent_1*resultImage.height(), resultImage.width(),
 percent_2*resultImage.height());
            QRect source2(0, percent_1*m_pix2.height(), m_pix2.width(), percent_2*m_pix2.height());
            painter.drawPixmap(target2, m_pix2, source2);

            //pourcentage
            painter.setPen(QPen(Qt::black));
            painter.setFont(QFont("TAHOMA",80,20));
            painter.drawText(10,resultImage.height()/2,QString::number(percent));
        }
        QIcon  iconeCourante(resultImage);
        //affichage dans un label
        m_Aresult->setPixmap(resultImage);
        //affichage sous forme d'icone dans un bouton
        m_AresultIcone->setIcon(iconeCourante);
        //icone de la fenetre
        setWindowIcon (iconeCourante);
        //icone du systray
        m_trayIcon.setIcon(iconeCourante);

    }
    void OpenPix1()
    {
        static QString s;
        s =QFileDialog::getOpenFileName
            (
                this,
                "image 1",
                s,
                "images (*.jpg *.png *.bmp *.gif)\n *.*"
            );
        if (! s.isEmpty())
        {
            QPixmap tmp(s);
            m_pix1 = tmp.scaled(200,200,Qt::KeepAspectRatio);
            m_Apix1->setPixmap(m_pix1);
        }
    }
    void OpenPix2()
    {
        static QString s;
        s =QFileDialog::getOpenFileName
            (
                this,
                "image 2",
                s,
                "images (*.jpg *.png *.bmp *.gif)\n *.*"
            );
        if (! s.isEmpty())
        {
            QPixmap tmp(s);
            m_pix2 = tmp.scaled(200,200,Qt::KeepAspectRatio);
            m_Apix2->setPixmap(m_pix2);
        }
    }
private :
     QPixmap m_pix1;
     QPixmap m_pix2;

     QLabel * m_Apix1;
     QLabel * m_Apix2;

     QSystemTrayIcon m_trayIcon;
     QPushButton * m_AresultIcone;
```

Compose une image à partir de deux images

```
        QLabel * m_Aresult;
};

#include "main.moc"

int main(int argc, char *argv[])
{

    QApplication app(argc,argv);
    myWindows w;
    w.show();

    return app.exec();
}
```

Quel est le rapport entre Item View et Graphics View ?

Auteurs : Benjamin Poulain ,

Il n'y a pas de liens de parenté entre ces deux là. Au début il arrive de se tromper entre Item View et Graphics View simplement car il y a "View" dans les deux noms. Graphics View permet de dessiner et gérer des objets 2D. Item View permet de gérer des éléments (les items), et de les représenter visuellement (en table, liste et arbre). Rien n'empêche de combiner ces deux infrastructures, mais il vaut mieux ne pas les confondre pour lire la documentation ou pour communiquer.

Comment valider des entrées utilisateurs dans une zone d'édition ?

Auteurs : François Jaffré ,

Qt fournit un mécanisme de gestion de la validation et de vérification des entrées utilisateurs dans une zone d'édition. La classe de base est la classe QValidator dont dérive les classes QDoubleValidator, QIntValidator, et QRegExpValidator. L'utilisation de ces classes est très simple au moins pour ce qui est des deux premières.

Exemple d'utilisation: On veut limiter l'entrée dans une zone d'édition de nombres entiers compris entre 0 et 100 et uniquement ce type de nombre.

```
//On définit un objet de type QIntValidator acceptant uniquement des nombre entier entre 0 et 100
QIntValidator* validator =  new QIntValidator (0, 100, this);
//Création d'une zone d'édition : ici une QLineEdit
QLineEdit *edit= new QLineEdit(this);
//On applique l'objet QIntValidator à la zone d'edition
edit->setValidator(validator);
```

Le même principe est utilisable pour contrôler par exemple une valeur double avec QDoubleValidator. Si l'on souhaite un motif spécifique par exemple une adresse IP, un numéro de téléphone ou autre il faut utiliser la classe QRegExpValidator ou créer soit même un validator à partir d'une classe dérivant de QValidator.

lien : QValidator

Comment créer un programme dans la zone de notification ?

Auteurs : François Jaffré ,

Qt depuis sa version 4.2 permet grâce à la classe QSystemTrayIcon de créer facilement des programmes ayant une entrée dans la zone de notification. Cela se fait à l'aide des quelques méthodes suivantes :

- **void setIcon (const QIcon & icon) -> Permet d'appliquer une image qui servira d'icon dans la zone de notification**
- **void setContextMenu (QMenu * menu) -> Permet l'ajout d'un menu quand on fait un click droit sur l'icone se trouvant dans la zone de notification.**
- **void showMessage (const QString & title, const QString & message, MessageIcon icon = Information, int millisecondsTimeoutHint = 10000) -> Permet l'affichage d'un popup dans la zone de notification.**

Exemple

```
#include <QApplication>
#include <QSystemTrayIcon>
#include <QMenu>
#include <QPushButton>
#include <QHBoxLayout>

class SysTray : public QWidget
{
    Q_OBJECT

private :
    QSystemTrayIcon* m_SystIcon;
    QPushButton* m_Bouton_Message;


public :
    SysTray()
    {
        //Bontou qui nou servira pour l'affichage de l'info bulle
        this->m_Bouton_Message = new QPushButton("Affiche text",this);

        //Gestion du Layout pour l'affichage
        QHBoxLayout *layout = new QHBoxLayout();
        layout->addWidget(m_Bouton_Message);
        this->setLayout(layout);

        //Création de l'objet gérant l'entré dans la zone de notification
        m_SystIcon = new QSystemTrayIcon(this);

        //Création du menu qui apparaitera apres un click droit sur l'icon
        QMenu* sysTrayMenu = new QMenu(this);

        //Création des differents actions pour le menu de la zone de notification
        QAction* hide = new QAction("Hide",this);
        QAction* show = new QAction("Show",this);
        QAction* quit = new QAction("Quit",this);


        //On ajoute nos actions au menu system
        sysTrayMenu->addAction(hide);
        sysTrayMenu->addAction(show);
        sysTrayMenu->addAction(quit);

        //On lie le menu avec l'icon
        m_SystIcon->setContextMenu(sysTrayMenu);

        //On charge l'image dans l'object icon
        QIcon unIcon("windows.png");

        //On place notre image dans la zone de notification
        m_SystIcon->setIcon(unIcon);

        //Affichage de l'icone dans la zone de notification
        m_SystIcon->show();

        //On ferme si la personne click sur "Quit"
        connect(quit, SIGNAL(triggered()), this, SLOT(close()));
```

Exemple

```
        //On affiche la fenetre si la personne click sur "Show"
        connect(show, SIGNAL(triggered()), this, SLOT(show()));
        //On cache la fenetre si la personne click sur "Hide"
        connect(hide, SIGNAL(triggered()), this, SLOT(hide()));
        //On affiche une info bulle dans la zone de notification
        connect(m_Bouton_Message, SIGNAL(clicked(bool)), this, SLOT(Affiche(bool)));
    }

private slots:
    void Affiche(bool valid)
    {
        //Affichage de l'info bulle
        m_SystIcon->showMessage
                (
                "Bavo !",
                "Vous venez de créer une application dans la zone de notification !"
                );
    }

};

#include "main.moc"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    SysTray w;
    w.show();
    return a.exec();
}
```

Remarque : La méthode showMessage() (arrivée à partir de Qt 4.3) qui fait apparaitre une info bulle est dépendante des paramètres de l'OS.

lien : QSystemTrayIcon

lien : QSystemTrayIcon::showMessage

lien : Tutoriel sur QSystemTrayIcon

Comment réaliser des fenêtres modales et amodales ?

Auteurs : François Jaffré ,

Qt permet comme tous les framworks gérant les GUI, la création de fenêtre modale (fenêtre bloquant l'accès aux autres fenêtres du programme) et amodale (fenêtre indépendante des autres fenêtres du programme). La classe de base gérant les boites de dialogues est QDialog. Celle-ci gère le mode modale ou amodale de la fenêtre au partir des méthodes suivantes :

- **bool isModal () const -> Permet de savoir si la boite de dialogue est modale ou pas.**
- **void setModal (bool modal) -> Permet de définir si la boite de dialogue est modale ou pas (Par défaut une boite de dialogue Qt est amodale).**
- **int exec() -> Force l'affichage de la boite de dialogue en mode modale (la valeur définie par setModal() est ignorée).**
- **void show() -> Force l'affichage de la boite de dialogue.**
- **void hide() -> Permet de cacher la boite de dialogue.**

Création de boites de dialogues modales et amodales

```
#include <QApplication>
#include <QHBoxLayout>
#include <QPushButton>
#include <QDialog>
#include <QLabel>

class AmodaleDial : public QDialog
{
    Q_OBJECT

private :
    QLabel * m_Label;

public :
    AmodaleDial(QWidget *parent): QDialog(parent)//Contructeur de la fenêtre amodale
    {
        //Création du label contenu dans la fenêtre
        m_Label = new QLabel("Cette fenêtre n'est pas modal", this);
        m_Label->resize(100,50);
        //Gestion du layout
        QHBoxLayout* layout = new QHBoxLayout();
        layout->addWidget(m_Label);
        setLayout(layout);
    }

};

class ModaleDial : public QDialog
{
    Q_OBJECT

private :
QLabel* m_Label;

public :
    //Contructeur de notre fenêtre
    ModaleDial(QWidget *parent): QDialog(parent)
    {
        //Création du label contenu dans la fenêtre
        m_Label = new QLabel ("Cette fenêtre est modale", this);
        m_Label->resize(100,50);
        QHBoxLayout* layout = new QHBoxLayout();
        layout->addWidget(m_Label);
        setLayout(layout);
    }

};

class Window : public QDialog
{
    Q_OBJECT

private :
    QPushButton* m_Bouton_ModaleFrame;
    QPushButton* m_Bouton_AmodaleFrame;
```

Création de boites de dialogues modales et amodales

```
    //Pointeur sur notre fenêtre qui sera amodale
    AmodaleDial* m_amodaleDial;

public :
    Window()
    {
        //Création des differents bouton
        m_Bouton_ModaleFrame = new QPushButton("Modale Frame", this);
        m_Bouton_AmodaleFrame = new QPushButton("Amodale Frame", this);
        //Gestion du layout
        QHBoxLayout* layout = new QHBoxLayout();
        layout->addWidget(m_Bouton_ModaleFrame);
        layout->addWidget(m_Bouton_AmodaleFrame);
        setLayout(layout);
        //Instanciation de notre fenêtre qui sera amodale
        m_amodaleDial = new AmodaleDial(this);
        //Connections des different signaux slots pour gérer le click
        connect(this->m_Bouton_ModaleFrame, SIGNAL(clicked(bool)), this, SLOT(clickModale(bool)));
        connect(this->m_Bouton_AmodaleFrame, SIGNAL(clicked(bool)), this, SLOT(clickAmodale(bool)));
    }

private slots:
    void clickModale(bool valid)
    {
        //Création de notre fenêtre modale
        ModaleDial modaleDial(this);
        //On rend visible et modale notre fenêtre avec la methode exec()
        modaleDial.exec();
    }

    void clickAmodale(bool valid)
    {
        //On rend visible notre fenêtre amodale qui par default est amodale
        m_amodaleDial->show();
    }

};

#include "main.moc"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Window w;
    w.show();
    return a.exec();
}
```

Remarque : Tout les QWidgets et par conséquent leurs classes dérivé peuvent être modal en modifiant leur propriétés windowModality à l'aide de la méthode setWindowModality(). Cependant il est préférable de gérer ce type de boite de dialogue à partir d'une QDialog et nom d'un QWidget.

lien : QDialog

lien : QWidget::setWindowModality

Comment manipuler du texte sélectionné avec QTextCursor ?

Auteurs : Louis du Verdier ,

Les curseurs sont très utilisés dans les classes pouvant les accueillir (comme l'autorisent par défaut QTextEdit, QTextBrowser et d'autres classes encore).

Pour retrouver la sélection courante qu'à faite l'utilisateur d'un programme, il suffit d'utiliser la fonction selectedText() de la manière qui suit :

```
// QString QTextCursor::selectedText () const
widget->textCursor().selectedText();
```

Notes : Cette fonction, retournant un QString, peut parfaitement être vide (ou empty en anglais).

Il est de même possible de supprimer la sélection courant à l'aide de removeSelectedText() :

```
// void QTextCursor::removeSelectedText ()
widget->textCursor().removeSelectedText();
```

Note : La suppression du texte sélectionné ne se produira que s'il y a lieu d'une sélection.

L'insertion de texte avec l'assistance des curseurs s'effectue avec insertText() qui entre du texte à partir de la position du curseur.

```
// void QTextCursor::insertText ( const QString & text, const QTextCharFormat & format )
widget->textCursor().insertText("Chaine de type QString");
```

Voici un exemple, reprenant ce qui est inscrit ci-dessus, permettant d'effectuer une modification sur un texte sélectionné sans pour autant modifier plusieurs chaines identiques :

```
 // ...
zoneTexte = new QTextEdit(this);
connect(zoneTexte, SIGNAL(selectionChanged()), this, SLOT(transformer()));
// ...

void XXXXXXXX::transformer()
{
    // On garde en mémoire le texte sélectionné
    QString selectionTexte(zoneTexte->textCursor().selectedText());
    // On supprime celui-ci sans pour autant supprimer les chaines identiques
    zoneTexte->textCursor().removeSelectedText();
    // Et on effectue l'opération voulue au départ
    zoneTexte->textCursor().insertText("<italique>" + selectionTexte + "</italique>");
}
```

lien : QTextCursor

Comment utiliser les icônes par défaut de Qt ?

Auteurs : François Jaffré ,

Qt fournit dans son framework des icônes standards comme flèches, fichier, dossiers, lecteur DVD, corbeille, etc. La récupération de ces icônes se fait à l'aide de la classe QStyle avec la méthode suivante :

```
QIcon QStyle::standardIcon (StandardPixmap standardIcon, const QStyleOption * option = 0, const
 QWidget * widget = 0) const
```

Qui permet de récupérer l'icône choisi à partir de l'énumération StandardPixmap.

Exemple d'utilisation en appliquant un icone flèche sur un bouton :

```
//Création d'un contrôle ici un bouton
```

```
QPushButton button;
//On récupère un pointeur sur l'objet QStyle correspondant à l'application
QStyle* style =  QApplication::style();
//On récupère l'icône désiré ici une flèche pointant vers le haut
QIcon iconFleche = style->standardIcon(QStyle::SP_ArrowUp);
//On applique notre icônes à notre contrôle
button.setIcon(iconFleche);
```

lien : QStyle::StandardIcon

lien : Liste des icons standard

Sommaire > Le contenu de Qt4 > QtGui > Graphics View

Qu'est-ce que Graphics View ?

Auteurs : Benjamin Poulain ,

Graphics View est une infrastructure qui permet de dessiner des scènes 2D complexes.

Graphics View permet de:
- **dessiner des éléments en 2D**
- **faire interagir l'utilisateur avec ces éléments**
- **d'appliquer des transformations avancées sur les éléments**
- **d'animer les éléments**

Voici un exemple minimaliste pour montrer quelques possibilités de Graphics View.

L'exemple insère une boîte de dialogue dans la scène, et la déforme:

```
#include <QApplication >
#include <QGraphicsScene>
#include <QGraphicsProxyWidget>
#include <QGraphicsView>
#include <QFileDialog>

int main(int argc, char **argv)
{
    QApplication app(argc, argv);
    QGraphicsScene scene;
    QGraphicsView view(&scene);

    QFileDialog fileDialog;
    QGraphicsProxyWidget *item = scene.addWidget(&fileDialog);
    item->rotate(-15);
    item->shear(-0.5, 0);

    view.resize(800, 600);
    view.show();
    return app.exec();
}
```

Comment accélérer Graphics View ?

Auteurs : Benjamin Poulain ,

Si vous avez un problème de vitesse avec Grahics View, commencez par vérifiez que le problème de vitesse ne se situe pas dans un de vos items. Graphics View est extrêmement performant, il n'est pas inhabituel d'y dessiner plusieurs milliers d'éléments (essayez la demo "chip" par exemple). Les problèmes de vitesse sont généralement dû à un graphics item qui se dessine trop lentement, ou à des rafraichissements abusifs.

Si le problème ne vient pas de là, vous pouvez utiliser un backend plus performant pour dessiner. Dessiner avec les backends de la plateforme est assez lent, Qt contourne le problème en dessinant sur un format interne optimisé. Cela peut être configuré en ajoutant "-graphicssystem raster" ou "-graphicssystem opengl" (expérimental dans Qt 4.5) dans les options de l'application. Notez que l'option raster ne change rien pour Windows car c'est déjà le backend par défaut.

```
view.setViewport(new QGLWidget);
```

Si cela n'est toujours pas suffisant, il vous faut optimiser précisément votre vue à l'aide des options de QGraphicsView. Une première option à considérer est renderHints. Cette option permet de configurer les paramètres du QPainter utilisé pour dessiner la scène. Si vous utilisez l'antiAliasing par exemple, cela ralenti le dessin des items.

```
view.setRenderHint(QPainter::Antialiasing, false);
view.setRenderHint(QPainter::TextAntialiasing, false);
```

Un autre paramètre utile est optimizationFlags. Cette propriété permet de changer les paramètres de bas niveau du dessin de la scène. Par exemple, concevoir les items pour pouvoir utiliser QGraphicsView::DontSavePainterState est important si les items sont très nombreux. À la différence des autres options, les paramètres de optimizationFlags impliquent des effets de bords sur le rendu de la scène. Ne changez ce paramètre que si vos items sont conçus en conséquence. Finalement, il existe le paramètre cacheMode pour accélérer le rendu du fond de la vue. Le cacheMode permet de demander à la vue de créer un cache pour l'image de fond. Cette option n'est utile que si le fond est long à dessiner, par exemple s'il s'agit d'une image, d'un dégradé ou si la transparence est utilisée.

```
view.setCacheMode(QGraphicsView::CacheBackground);
```

Sommaire > Le contenu de Qt4 > QtGui > Model View

Qu'est ce que Model View ?

Auteurs : Benjamin Poulain ,

Model View est l'infrastructure de Qt qui permet de gérer les vues arborescentes, les listes et les tables. Il se base sur un pattern Modèle-View-Delegate.

Le modèle est une interface vers les données, ce qui permet d'abstraire la façon dont celle-ci sont représentées (mémoire, base de données, système de fichier, etc).

La vue est la représentation visuelles des données sur l'écran. Qt fourni trois types de représentations: liste (QListView), arbre (QTreeView) et tableaux (QTableView).

Le délégué permet de faire le lien entre la vue et le modèle. Le délégué se charge de dessiner les éléments du modèle dans la vue, et de gérer l'édition de ces éléments.

Par rapport à Modèle-Vue-Controleur, le délégué est une sorte de super contrôleur. Il génère un éditeur à la demande de la vue, et se charge de faire parvenir les informations au modèle, c'est son rôle de contrôleur. En plus de cela, le délégué se charge de dessiner les éléments du modèle, ce qui permet de pouvoir complètement personnaliser la vue.

Voici un exemple d'utilisation de Model View en utilisant le système de fichier comme modèle

```
#include <QApplication>
#include <QFileSystemModel>
#include <QTreeView>
#include <QListView>
#include <QTableView>

int main(int argc, char **argv)
{
   QApplication app(argc, argv);

   QFileSystemModel model; // le modèle est ici le système de fichier
   model.setRootPath(QDir::rootPath());

   QTreeView treeView;
   treeView.setModel(&model);
   QListView listView;
   listView.setModel(&model);
   QTableView tableView;
   tableView.setModel(&model);

   // partageons la selection pour plus de cohérence
   QItemSelectionModel *selection = treeView.selectionModel();
   listView.setSelectionModel(selection);
   tableView.setSelectionModel(selection);

   // changeons la racine en fonction de la selection

  // (car les tables et listes en sont pas fait pour naviguer, on pourra toujours naviguer avec l'arbre)
   QObject::connect(selection, SIGNAL(currentChanged(const QModelIndex, const
QModelIndex)), &listView, SLOT(setRootIndex(const QModelIndex)));
   QObject::connect(selection, SIGNAL(currentChanged(const QModelIndex, const
QModelIndex)), &tableView, SLOT(setRootIndex(const QModelIndex)));

   treeView.show();
   listView.show();
   tableView.show();

   return app.exec();
```

Voici un exemple d'utilisation de Model View en utilisant le système de fichier comme modèle

```
}
```

Comment trier un QTreeWidget ?

Auteurs : Yan Verdavaine ,

Les QTreeWidget trient leurs items grâce à l'opérateur < des QTreeWidgetItem. Par défaut, cet opérateur compare deux QString. Malheureusement, ceci ne correspond pas toujours à ce que l'on souhaite.

Pour y remédier, il suffit de créer sa propre classe d'item qui hérite de QTreeWidgetItem et qui redéfinit cet opérateur.

Tout fonctionnera grâce au polymorphisme !

Exemple : différence entre le tri lexical de QTreeWidgetItem et le tri numérique implémenté par notre classe item

Trier un QTreeWidget

```
#include <QtGui>
#include <cstdlib>  //srand, rand et RAND_MAX
#include <ctime>    //time

class MyQTreeWidgetItem :public QTreeWidgetItem
{
public :
    //Constructeur de MyQTreeWidgetItem.
    MyQTreeWidgetItem ( QTreeWidgetItem * parent, int type = Type ):QTreeWidgetItem(parent,type){};
    MyQTreeWidgetItem ( QTreeWidget * parent, int type = Type ):QTreeWidgetItem(parent,type){};

    //Reimplemente le test
    bool operator< ( const QTreeWidgetItem & other ) const
    {
        //On recupere la colonne utilisee pour le test
        const QTreeWidget * pTree =treeWidget ();
        //Si aucun TreeWidget n'est associe, on utilise la colonne 0
        int colonne   = pTree ? pTree->sortColumn() : 0;

        //On recupere les donnees sous forme de variant situé dans la colone
        QVariant  q1= data ( colonne , 0 );
        QVariant  q2= other.data ( colonne , 0 );
        //On verifie que les deux variants sont de même type
        if (q1.type() == q2.type())
        {
            //Si se sont des int
            if (q1.type()==QVariant::Int)     return q1.toInt()<q2.toInt();
            //Si se sont des double
            if (q1.type()==QVariant::Double)     return q1.toDouble()<q2.toDouble();
        }
        //Sinon on appele le comparateur de la class mere "QTreeWidgetItem"
        return QTreeWidgetItem::operator<(other);
    }

};


int main(int argc, char* argv[])
{
    srand(time(NULL));
    QApplication app(argc, argv);

    QTreeWidget tree;
    //On remplit l'entete
```

Trier un QTreeWidget

```
    QStringList entete;
    entete << "int" << "double" ;
    tree.setHeaderItem(new QTreeWidgetItem((QTreeWidgetItem*)0,entete));
    tree.setSortingEnabled(true);

    //On cree une premiere branche avec des QTreeWidgetItem
    //Les items seront tries de manière lexicale
    QTreeWidgetItem *parentitem = new QTreeWidgetItem(&tree);
    parentitem->setText(0,"QTreeWidgetItem");
    for (int i=0;i<20;++i)
    {
        QTreeWidgetItem*item = new QTreeWidgetItem(parentitem);
        item ->setData (0, 0, QVariant(i));
        item ->setData (1, 0, QVariant(rand()/(1.+RAND_MAX)*i));
    }

    //On cree une deuxieme branche avec notre nouvelle classe
    //Les items seront tries de maniere numerique
    parentitem = new MyQTreeWidgetItem(&tree);
    parentitem->setText(0,"MyQTreeWidgetItem ");
    for (int i=0;i<20;++i)
    {
        MyQTreeWidgetItem *item = new MyQTreeWidgetItem(parentitem);
        item ->setData (0, 0, QVariant(i));
        item ->setData (1, 0, QVariant(rand()/(1.+RAND_MAX)*i));
    }

    tree.resize(400, 650);
    tree.expandAll();
    tree.show();
    return app.exec();
}
```

Comment trier un QListWidget ?

Auteurs : Yan Verdavaine ,

Les QListWidget trient leurs items avec l'opérateur < des QListWidgetItem. Par défaut, cet opérateur compare deux QString. Malheureusement, ceci ne correspond pas toujours à ce que l'on souhaite. Pour y remédier, il suffit de créer sa propre classe d'item qui hérite de QListWidgetItem et qui redéfinit cet opérateur. Tout fonctionnera simplement grâce au polymorphisme.

Exemple : différence entre le tri par ordre croissant numérique implémenté par notre classe item (vue de gauche) et le tri par ordre croissant lexical des QListWidgetItem (vue de droite)

Trier un QListWidget

```
#include <QtGui>

class MyQListWidgetItem :public QListWidgetItem
{
public :
    //Constructeur de MyQListWidgetItem.
    MyQListWidgetItem ( QListWidget * parent, int type = Type ):QListWidgetItem(parent,type){};

   //Reimplemente le teste
   bool operator< ( const QListWidgetItem & other ) const
       {
       //On recupere les donnees sous forme de variant
       QVariant  q1= data ( 0 );
       QVariant  q2= other.data (  0 );
```

Trier un QListWidget

```
        //On verifie que les deux variants sont de même type
        if (q1.type() ==q2.type())
         {
            //Si ce sont des int
            if (q1.type()==QVariant::Int)     return q1.toInt()<q2.toInt();
            //Si ce sont des double
            if (q1.type()==QVariant::Double)     return q1.toDouble()<q2.toDouble();
         }
        //Sinon on appelle le comparateur de la classe mere "QListWidgetItem"
        return QListWidgetItem::operator<(other);
        }

};


int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    //Viewer  de gauche
    QListWidget  * pList1 = new QListWidget;
    //On active le tri. Par defaut, le tri est par ordre croissant
    pList1->setSortingEnabled(true);
    //On utilise nos items. Le tri se fera par ordre numerique
     for (int i=0;i<20;++i)
    {
       QListWidgetItem*item = new MyQListWidgetItem(pList1);
       item ->setData ( 0, QVariant(i));
    }

    //Viewer  de droite
    QListWidget  * pList2 = new QListWidget;
    //On active le tri. Par defaut, le tri est par ordre croissant
    pList2->setSortingEnabled(true);
    //On utilise les QListWidgetItem. Le tri se fera par ordre lexicale
    for (int i=0;i<20;++i)
    {
       QListWidgetItem*item = new QListWidgetItem(pList2);
       item ->setData ( 0, QVariant(i));
    }

    QWidget w;
    QHBoxLayout  hl(&w);
    hl.addWidget(pList1);
    hl.addWidget(pList2);
    w.resize(400, 400);
    w.show();
    return app.exec();
}
```

Comment effacer le contenu d'un QTableWidget ?

Auteurs : François Jaffré ,

Pour effacer le contenu d'un QTableWidget c'est très simple, il faut tout d'abord effacer les QTableWidgetItem qu'il contient puis effacer toutes les lignes du tableau.

```
//On efface les QTableWidgetItem contenu dans le tableau
tableWidget->clear();
//On efface toutes les lignes du tableau
tableWidget->setRowCount(0);
```

Sommaire > Le contenu de Qt4 > QtXml

Comment lire un fichier XML avec QXmlStreamReader ?

Auteurs : François Jaffré ,

Qt fourni depuis sa version 4.3 la classe QXmlStreamReader qui permet de lire rapidement des fichiers XML un peu à la manière de SAX mais en non événementiel. Comme SAX on parcourt l'arbre XML et l'on ne peut le remonter pendant son parcours. QXmlStreamReader repose sur le principe d'une boucle dans laquelle on va parcourir le fichier à l'aide de la methode readNext() et vérifier sur quel type de token on est positionné.

Exemple avec le fichier XML

```
<developpez>
    <contributeur>Superjaja</contributeur>
</developpez>
```

Si l'on parcourt le fichier avec la méthode readNext() les différents types de token renvoyés sont:

- **StartDocument**
- **StartElement (name() == "developpez")**
- **StartElement (name() == "contributeur")**
- **Characters (text() == "Superjaja")**
- **EndElement (name() == "contributeur")**
- **EndElement (name() == "developpez")**
- **EndDocument**

Pour connaitre tout les types de tokens possible se référer à la documentation Qt ici.

Voici un exemple un peu plus complexe

Fichier XML

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <site>
        <nom>developpez</nom>
        <url>www.developpez.com</url>
    </site>
    <contributeur>
        <nom faq="c++">Alp</nom>
    </contributeur>
    <contributeur>
        <nom faq="Qt">Mongaulois</nom>
    </contributeur>
</root>
```

Parcours de ce fichier avec QXmlStreamReader

```
QXmlStreamReader reader;//Objet servant à la navigation
QString fileName = "developpez.xml";
QFile file(fileName);
file.open(QFile::ReadOnly |
 QFile::Text); //Ouverture du fichier XML en lecture seul et en mode texte
reader.setDevice(&file);//Initialise l'instance reader avec le flux XML venant de file


//Le but de cette boucle est de parcourir le fichier et de vérifier si l'on est au debut d'un element.
reader.readNext();
while (!reader.atEnd())
{
    if (reader.isStartElement())
    {
```

Parcours de ce fichier avec QXmlStreamReader

```
        if (reader.name() == "root")
        {
            reader.readNext();//On va au prochain token
            //Tant que celui-ci n'est pas un element de depart on avance au token suivant.
            while(reader.isStartElement()==false)
                reader.readNext();

            if(reader.name() == "site")
            {
                reader.readNext();
                while(reader.isStartElement()==false)
                reader.readNext();
                if(reader.name() == "nom")
                {
                    QString strNom = reader.readElementText();
                    reader.readNext();
                    while(reader.isStartElement()==false)
                    reader.readNext();
                }
                if(reader.name() == "url")
                {
                    QString strUrl = reader.readElementText();
                }

            }
        }

        if(reader.name() == "contributeur")
        {
            reader.readNext();
            while(reader.isStartElement()==false)
                reader.readNext();
            if(reader.name() == "nom")
            {
                QString attrFAQ = reader.attributes().value("faq").toString();
                QString strNameContrib = reader.readElementText();
            }
        }
    }
    reader.readNext();//On va au prochain Token
}
file.close();
```

Comment écrire un fichier XML avec QXmlStreamWriter ?

Auteurs : François Jaffré ,

Qt depuis sa version 4.3 fourni la classe QXmlStreamWriter pour écrire des fichiers XML. Celle-ci est relativement simple et fonctionne sur le même principe que QXmlStreamReader pour la lecture.
On souhaite écrire le fichier XML suivant :

Fichier XML

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <site>
        <nom>developpez</nom>
        <url>www.developpez.com</url>
    </site>
    <contributeur>
        <nom faq="c++">AAA</nom>
    </contributeur>
    <contributeur>
        <nom faq="Qt">BBB</nom>
```

Fichier XML
```
      </contributeur>
  </root>
```

avec

Réalisation avec QXmlStreamWriter
```
  QString fileName = "developpez.xml";
  QFile file(fileName);
  //Ouverture du fichier en écriture et en texte.
  file.open(QFile::WriteOnly | QFile::Text);
  QXmlStreamWriter writer(&file);

  //Permet l'indentation du fichier XML
  writer.setAutoFormatting(true);

  //Ecrit l'entête du fichier XML <?xml version="1.0" encoding="UTF-8" ?>
  writer.writeStartDocument();

  //Élément racine du fichier XML
  writer.writeStartElement("root");

  //Ajoute l'élément site
  writer.writeStartElement("site");

  //Ajoute l'élément nom et lui applique le texte "developpez" et ferme l'élément nom
  writer.writeTextElement("nom", "developpez");

  //Ajoute l'élément url et lui applique le texte "www.developpez.com" et ferme l'élément url
  writer.writeTextElement("url", "www.developpez.com");

  //Ferme l'élément site
  writer.writeEndElement();

  //Ajoute l'élément contributeur
  writer.writeStartElement("contributeur");

  //Ajoute l'élément nom
  writer.writeStartElement("nom");

  //Ajoute un attribut a l'élément nom
  writer.writeAttribute("faq", "c++");

  //Ajout du texte à l'élément nom
  writer.writeCharacters("AAA");

  //Ferme l'élément nom
  writer.writeEndElement();

  //Ferme l'élément contributeur
  writer.writeEndElement();

  //Ajoute l'élément contributeur
  writer.writeStartElement("contributeur");

  //Ajoute l'élément nom
  writer.writeStartElement("nom");

  //Ajoute un attribut à l'élément nom
  writer.writeAttribute("faq", "Qt");
  writer.writeCharacters("BBB");

  //Ferme l'élément nom
  writer.writeEndElement();
  //Ferme l'élément contributeur
  writer.writeEndElement();
```

Réalisation avec QXmlStreamWriter

```
//Finalise le document XML
writer.writeEndDocument();

//Fermer le fichier pour bien enregistrer le document et ferme l'élément root
file.close();
```

Comment lire un fichier XML avec DOM ?

Auteurs : François Jaffré ,

Qt permet de lire facilement un fichier XML avec l'API DOM. DOM permet de naviguer dans un document XML comme dans un arbre avec toujours une relation parent enfant entre les noeuds qui composent l'arbre. La classe de base permettant de gérer les document avec DOM sous Qt est QDomDocument et sa documentation se trouve ici Exemple de parcours d'un fichier XML simple :

fichier xml

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <site>
        <nom>developpez</nom>
        <url>www.developpez.com</url>
    </site>
    <contributeur>
        <nom faq="c++">AAA</nom>
    </contributeur>
    <contributeur>
        <nom faq="Qt">BBB</nom>
    </contributeur>
</root>
```

Parcours du fichier en utilisant l'API DOM

```
QString fileName = "developpez.xml";
QFile file(fileName);
//Ouverture du fichier en lecture seul et en mode texte
file.open(QFile::ReadOnly | QFile::Text);

QDomDocument doc;
//Ajoute le contenu du fichier XML dans un QDomDocument et dit au QDomDocument de ne pas tenir coompte des namsp
doc.setContent(&file, false);

//ici racine pointe sur l'element <root> de notre document
QDomElement racine = doc.documentElement();

//ici racine pointe sur une fils de <root> c'est à dire <site>
racine = racine.firstChildElement();

//Boucle permettant la navigation dans le fichier XML
while(!racine.isNull())
{
    //Si on pointe sur un element de type <site>
    if(racine.tagName() == "site")
    {
        //On recupere le première enfant de l'element site c'est a dire <nom> ou <url>
        QDomElement unElement = racine.firstChildElement();

        //On parcours tous les enfants de l'element <site>
        while(!unElement.isNull())
        {
            //Si l'enfant de l'element site est l'element <nom>
            if(unElement.tagName() == "nom")
            {
```

Parcours du fichier en utilisant l'API DOM

```
                //On recupère le text contenu entre les balise <nom> </nom>
                QString strNom = unElement.text();
            }
            //Si l'enfant de l'element <site> est <url>
            else if(unElement.tagName() == "url")
            {
                //On recupère le text contenu entre les balise <url> </url>
                QString strURL = unElement.text();
            }
            //Permet d'aller au prochaine enfant de <site> et de poursuivre la boucle
            unElement = unElement.nextSiblingElement();
        }
    }

    //ici racine pointe sur un element <contributeur>
    if(racine.tagName() == "contributeur")
    {
        //On pointe sur l'element <nom>
        QDomElement elem = racine.firstChildElement();

        //On recupere le texte de l'element <nom>
        QString strNameContrib = elem.text();

        //Recuperation de l'attribut faq de l'elment <nom>
        QString attFaq = elem.attribute("faq");
    }

    //On va a l'element fils de <root> suivant
    racine = racine.nextSiblingElement();
}
```

Comment écrire un fichier XML avec DOM ?

Auteurs : François Jaffré ,

Qt fourni la classe QDomDocument qui permet de gérer les fichiers XML à la manière de DOM c'est à dire que le fichier XML est représenté par un arbre dans lequel on peut se déplacer grâce à une relation de parent enfant entre les noeuds qui composent l'arbre. Avec QDomDocument vous pouvez soit créer un fichier XML en partant de rien mais vous pouvez aussi éditer un fichier existant (ajout de noeuds, modification de texte...).

Exemple dont le but est de créer le fichier XML ci-dessous à l'aide de l'API DOM de Qt :

fichier xml

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <site>
        <nom>developpez</nom>
        <url>www.developpez.com</url>
    </site>
    <contributeur>
        <nom faq="c++">AAA</nom>
    </contributeur>
    <contributeur>
        <nom faq="Qt">BBB</nom>
    </contributeur>
</root>
```

écriture avec DOM

```
//Objet de base servant à la création de notre fichier XML
```

écriture avec DOM

```
QDomDocument doc;

//creation de l'entête du fichier XML <?xml version="1.0" encoding="UTF-8"?>
QDomNode xmlNode = doc.createProcessingInstruction("xml","version=\"1.0\" encoding=\"UTF-8\"");

//On insert cette entête avant le première enfant
doc.insertBefore(xmlNode, doc.firstChild());

 //Création de l'element <root>
QDomElement root = doc.createElement("root");

//On ajoute l'element <root> en tant que première enfant de notre document
doc.appendChild(root);

//Création de l'element <site>
QDomElement site = doc.createElement("site");

//On ajoute l'element <site> en tant que première enfant de l'element <root>
root.appendChild(site);

//création de l'element <nom>
QDomElement nom = doc.createElement("nom");

//On ajoute l'element <nom> en tant que première enfant de l'element <site>
site.appendChild(nom);

 //Creation du texte qui sera entre les balises <nom> </nom>
QDomText nomText = doc.createTextNode("developpez");

//On ajoute ce texte à l'element <nom>
nom.appendChild(nomText);

//Création de l'element <url>
QDomElement url = doc.createElement("url");

//On ajoute l'element <url> en tant que deuxième enfant de l'element <site>
site.appendChild(url);

//Creation du texte qui sera entre les balises <url> </url>
QDomText urlText = doc.createTextNode("www.developpez.com");

//On ajoute ce texte à l'element <url>
url.appendChild(urlText);

//Création de l'element <contributeur>
QDomElement contributeur1 = doc.createElement("contributeur");

//Création de l'element <nom>
QDomElement nomContrib1 = doc.createElement("nom");

//Ajout de l'attibut "faq" qui aura la valeur "C++" à l'element <nom>
nomContrib1.setAttribute( "faq","C++");

//On ajoute l'element <contributeur> en tant qu'enfant de l'element <root>
root.appendChild(contributeur1);

//On ajoute l'elemnt <nom> en tant qu'enfant de l'element <contributeur>
contributeur1.appendChild(nomContrib1);

//Creation du texte qui sera entre les balises <nom> </nom>
QDomText nameContrib1 = doc.createTextNode("AAA");

//On ajoute ce texte à l'element <nom>
nomContrib1.appendChild(nameContrib1);

//Création de l'element <contributeur>
QDomElement contributeur2 = doc.createElement("contributeur");
```

écriture avec DOM

```
//Création de l'element <nom>
QDomElement nomContrib2 = doc.createElement("nom");

//Ajout de l'attibut "faq" qui aura la value "Qt" à l'element <nom>
nomContrib2.setAttribute("faq","Qt");

//On ajoute l'element <contributeur> en tant qu'enfant de l'element <root>
root.appendChild(contributeur2);

//On ajoute l'elemnt <nom> en tant qu'enfant de l'element <contributeur>
contributeur2.appendChild(nomContrib2);

//Creation du texte qui sera entre les balises <nom> </nom>
QDomText nameContrib2 = doc.createTextNode("BBB");

//On ajoute ce texte à l'element <nom>
nomContrib2.appendChild(nameContrib2);

//Creation du fichier XML de sortie
QFile file( "developpez.xml" );

//Ouverture de ce fichier en lecture seule
file.open(QIODevice::WriteOnly);

//On crée une QTextStream a partir de ce fichier
QTextStream ts(&file);

//Valeur servant au nombre de caractere pour l'indentation du fichier XML
int indent = 2;

//On sauvegarde le fichier XML en precisant l'indentation desirée
doc.save(ts, indent);
```

Comment télécharger une page web avec QHttp ?

Auteurs : François Jaffré ,

Qt a encapsulé le protocole HTTP au sein de la classe QHttp. Celle-ci possède notamment les méthodes suivantes :

- **int post(const QString & path, QIODevice * data, QIODevice * to = 0) -> Permet l'envoie de données**
- **int get(const QString & path, QIODevice * to = 0) -> Permet de réceptionner des données**
- **int close() -> Ferme la connexion**

Téléchargement d'une page web

```
#include <QApplication>
#include <QPushButton>
#include <QGridLayout>
#include <QProgressBar>
#include <QLabel>
#include <QLineEdit>
#include <QHBoxLayout>
#include <QHBoxLayout>
#include <QHttp>
#include <QFile>
#include <QUrl>
#include <QMessageBox>
#include <QFileDialog>

class FAQHttp : public QWidget
{
    Q_OBJECT
    QPushButton     *m_Bouton_Dowload;//Bouton "Download"
    QProgressBar    *m_ProgressBar;//Sert a afficher la progression du telechargement
    QLineEdit       *m_Edit;//Chemin de l'URL de telechargment
    QLabel          *m_Label;
    QHttp           *m_http;//Permet la gestion du telechargement via HTTP
    QFile           *m_file;//Fichier de fin du telechargement
    int             m_httpId;//Contient l'ID de notre requette HTTP
public :
    FAQHttp()
    {
        //Creation des contrôles de types boutons, Edit, Label, ProgressBar
        this->m_Bouton_Dowload = new QPushButton("Télécharger",this);
        this->m_ProgressBar = new QProgressBar(this);
        this->m_Edit = new QLineEdit(this);
        this->m_Label = new QLabel("URL:", this);
        m_Label->setBuddy(m_Edit);

        //Gestion du layout pour le placement des boutons
        QHBoxLayout *firstLayout = new QHBoxLayout;
        firstLayout->addWidget(m_Label);
        firstLayout->addWidget(m_Edit);

        QHBoxLayout *Layout = new QHBoxLayout;
        Layout->addLayout(firstLayout);
        Layout->addWidget(m_Bouton_Dowload);

        QVBoxLayout *secondeLayout = new QVBoxLayout();
        secondeLayout->addLayout(Layout);
        secondeLayout->addWidget(m_ProgressBar);
        setLayout(secondeLayout);

        //On connecte les differents signaux et slots pour les boutons
        connect(this->m_Bouton_Dowload, SIGNAL(clicked(bool)), this, SLOT(click_Download(bool)));
        resize(400, 150);//Gestion de la taille de la fenêtre
    }

private slots:

    void DownloadFile(QString source , QString fichier)
    {
        QUrl adresse(source);
        this->m_http = new QHttp(adresse.host(),80,this);
        m_file = new QFile(fichier);
```

Téléchargement d'une page web

```
    if(!m_file->open(QIODevice::WriteOnly))//On vérifie qu'il n'y a pas de problème lors de l'ouverture du fichier
          {
              QMessageBox::information(this, "HTTP Problemes"," Probleme lors du telechargement");
              //On fait le ménage en memoire
              delete this->m_http;
              delete this->m_file;
              return;
          }
          //On fait la requette et on en récupère l'Id
          m_httpId = m_http->get(adresse.toString(), m_file);

          //On connecte les signaux slots pour avoir l'état d'avancement du telechargement
          connect(m_http,SIGNAL(requestFinished(int, bool)),this,SLOT(finish_dowload(int ,bool)));
          connect(m_http,SIGNAL(dataSendProgress(int,int)),this,SLOT(progress_dowload(int, int)));
          this->m_ProgressBar->reset();
          this->m_ProgressBar->setMinimum(0);
      }

      void finish_dowload(int httpId, bool error)//Est déclanché à la fin du telechargement
      {
          if (this->m_httpId != httpId)
          {
              return;
          }
          else
          {
              if(error)
                  QMessageBox::information(this, "HTTP Problemes"," Probleme lors du telechargement");
              else
                  QMessageBox::information(this, "HTTP"," Le telechargement a réussi");

              this->m_file->close();//On ferme le fichier une fois le telechargement terminé
              //On fait le ménage au niveau memoire
              delete this->m_http;
              delete this->m_file;
          }

      }

      void click_Download(bool valid)//Bouton "Start" "Stop"
      {
          QString saveFile = QFileDialog::getSaveFileName(this,"Download Files","C:\
\", "All Files (*.*)");
          if(!saveFile.isEmpty())
              DownloadFile(this->m_Edit->text(),saveFile);
      }

      void progress_dowload(int done , int total)//Permet de suivre l'avancé du telechargement
      {
          this->m_ProgressBar->setMaximum(total);

    this->m_ProgressBar->setValue(done);//On incremente la progress bar en fonction des données telechargées
      }

  };

  #include "main.moc"


  int main(int argc, char *argv[])
  {
      QApplication a(argc, argv);
      FAQHttp w;
      w.show();
      return a.exec();
  }
```

Remarque : Cette classe possède de nombreux signaux qui permettrent de suivre l'état de la connexion avec le serveur. Référez-vous à la doc pour plus d'information.
Remarque: Depuis Qt 4.4, QHttp est déprécié au profit des classes QNetworkAccessManager et QNetworkReply qui offrent plus de souplesse que QHttp.

lien : QHttp

Comment retrouver l'IP d'un domaine et vice versa ?

Auteurs : François Jaffré ,

Qt fournit dans son module réseau la classe QHostInfo qui permet de retrouver l'adresse ip correspondant à un nom de domaine mais aussi, si possible, de retrouver le domaine correspondant à l'ip. Dans ce but QHostInfo possède les méthodes static suivantes :

- **int QHostInfo::lookupHost (const QString & name, QObject * receiver, const char * member) -> Permet retrouver le nom de domaine à partir d'une IP ou vice versa de manière asynchrone (non bloquante).**
- **QHostInfo QHostInfo::fromName (const QString & name) -> Permet de retrouver l'ip appartenant au domaine et vice versa de manière synchrone (bloquante).**

Exemple

```
#include <QApplication>
#include <Qlabel>
#include <QVBoxLayout>
#include <QPushButton>
#include <QLabel>
#include <QLineEdit>

class ResolveDNS : public QWidget
{
    Q_OBJECT
    QLabel* m_Label_Domain;
    QLabel* m_Label_Ip;
    QLineEdit* m_Line_Domain;
    QLineEdit* m_Line_Ip;
    QPushButton* m_Button_Ip;
    QPushButton* m_Button_Domain;

public :
    ResolveDNS()//Constructeur
    {
        //Création des differents contôles
        m_Label_Domain = new QLabel("&Domain :", this);
        m_Label_Ip = new QLabel("&Ip :", this);
        m_Line_Domain = new QLineEdit(this);
        m_Line_Ip = new QLineEdit(this);

        m_Label_Domain->setBuddy(m_Line_Domain);
        m_Label_Ip->setBuddy(m_Line_Ip);
        m_Button_Ip = new QPushButton("Resolve IP",this);
        m_Button_Domain = new QPushButton("Resolve Domaine",this);

        //Gestion du layout
        QHBoxLayout* Layout = new QHBoxLayout();
        Layout->addWidget(m_Label_Domain);
        Layout->addWidget(m_Line_Domain);
        Layout->addWidget(m_Label_Ip);
        Layout->addWidget(m_Line_Ip);
        Layout->addWidget(m_Button_Domain);
        Layout->addWidget(m_Button_Ip);
        setLayout(Layout);

        //Connexions des differents signaux slot pour la gestions des boutons
        connect(this->m_Button_Ip, SIGNAL(clicked(bool)), this, SLOT(resolveIp(bool)));
        connect(this->m_Button_Domain, SIGNAL(clicked(bool)), this, SLOT(resolveDomain(bool)));
    }
private slots :
    void resolveDomain(bool valid)//Click sur "Resolve Domaine"
    {
        //Résolution domaine -> IP (mode synchrone)
        QHostInfo infoAdresse = QHostInfo::fromName(m_Line_Domain->text());
        QList<QHostAddress> result_adresses = infoAdresse.addresses();
        QString Ip;
        //Si plusieurs adresses correspondent à un même domaine
        for(int i=0; i < result_adresses.size(); i++)
        {
            //On crée une chaine qui contient toutes les adresses ip
            Ip = Ip + result_adresses.at(i).toString() +"; ";
        }
        //Affichage dans l'édit
```

Exemple

```
        m_Line_Ip->setText(Ip);
    }

    //Click sur "Resolve IP"
    void resolveIp(bool valid)
    {
        //Résolution IP -> domaine si possible de manière asynchrone
        QHostInfo::lookupHost(m_Line_Ip->text(),this, SLOT(resolveIpFinish(QHostInfo)));
    }

    //Slot appellé à la fin de la resolution IP -> domaine (mode asynchrone)
    void resolveIpFinish(const QHostInfo& infoAdresse)
    {
        //Affichage dans l'édit
        m_Line_Domain->setText(infoAdresse.hostName());
    }
};

#include "main.moc"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    ResolveDNS w;
    w.show();
    return a.exec();
}
```

lien : **QHostInfo**

Sommaire > Les bibliothèques complémentaires

Comment gérer le port série ?

Auteurs : Matthieu Brucher ,

Qt ne gère pas de manière native le port série. En revanche, des projets externes existent, en particulier QExtSerialPort.

lien : Le site officiel de QExtSerialPort

Sommaire > Les bibliothèques complémentaires > Qwt

Qu'est ce que Qwt ?

Auteurs : gulish ,

Qwt (Qt Widgets for Technical applications) est une librairie annexe à Qt, qui utilise cette dernière, et qui simplifie la création de widgets de mesure (Courbes, Graphiques, Jauges, Cadrans, ...). Cette librairie, ainsi que toutes les informations la concernant, sont disponibles sur la page web du projet : La librairie Qwt.

lien : Qwt

Quels types de composants puis je créer avec Qwt ?

Auteurs : gulish ,

Quelques images des exemples Qwt sont disponibles dans le chapitre "Screenshots" de la page web de la librairie. Elles sont assez représentatives du type de composants pouvant être crées, même si une personnalisation est toujours possible.

lien : Qwt

Est il difficile d'utiliser Qwt ?

Auteurs : gulish ,

Les classes Qwt sont basées sur celles de Qt, et leur utilisation est relativement simple. De plus, les classes disponibles sont assez complètes, et permettent facilement de créer l'objet que l'on désire. Cependant, cette librairie est moins intuitive que Qt, et la documentation la concernant est plutôt succincte. Il est nécessaire de connaître les bases de l'utilisation de Qt pour utiliser Qwt.

Comment installer Qwt ?

Auteurs : gulish ,

Qwt est basé sur Qt, et il faut donc préalablement avoir installé la librairie Qt. Il suffit ensuite de décompresser l'archive Qwt, modifier le fichier .pri (similaire au fichier .pro) selon vos besoins, puis d'exécuter qmake, puis make ou équivalents (nmake, cmake ...).

Comment insérer un composant Qwt dans mon application Qt ?

Auteurs : gulish ,

Les repères des courbes (QwtPlot) et les éléments de type "jauges" (QwtCompass, QwtDial, QwtKnob, ...) héritent tous de la classe QWidget. Il est donc facile de les ajouter, comme tout autre widget, dans le layout de votre application.

Comment créer un repère dans mon interface ?

Auteurs : gulish ,

La classe principale pour la gestion des repères est QwtPlot. Son utilisation est très simple. Par défaut, seule la partie positive des axes x et y apparaît, mais cela est modifiable.

```
#include <QApplication>
#include <qwt_plot.h>

int main(int argc, char** argv)
{
    QApplication app(argc,argv);
    QwtPlot myPlot;
    myPlot.show();
    return app.exec();
}
```

Comment changer la couleur de fond de mon repère ?

Auteurs : gulish ,

On utilise la méthode setCanvasBackground(const QColor &c) :

```
#include <QApplication>
#include <qwt_plot.h>

int main(int argc, char** argv)
{
    QApplication app(argc,argv);
    QwtPlot myPlot;
    myPlot.setCanvasBackground(Qt::white);
    myPlot.show();
    return app.exec();
}
```

Comment ajouter une courbe sur mon repère ?

Auteurs : gulish ,

Les courbes se créent grâce à la classe QwtPlotCurve. Cette classe permet une grande personnalisation de l'affichage (symboles des points, couleur, traçage ou non de ligne, etc.). Une courbe étant une suite de points, il faut renseigner une suite de points à QwtPlotCurve. Pour cela, on utilise la méthode setData(...) qui peut accepter différents arguments. La plus triviale est setData (const double *xData, const double *yData, int size), ou xData représente un tableau de double, contenant toutes les abscisses des points de notre courbe, yData contient lui les ordonnées. Ainsi, notre premier point aura comme coordonnées [xData[0],yData[0]]. Le troisième paramètre, size, renseigne sur le nombre de points à afficher, en partant de zéro. Bien sûr, xData et yData doivent au moins contenir "size" éléments.

```
#include <QApplication>
#include <qwt_plot.h>
#include <qwt_plot_curve.h>

int main(int argc, char** argv)
{
    QApplication app(argc,argv);
    QwtPlot myPlot;
    QwtPlotCurve myCurve;

    //indique à myPlot qu'il doit afficher myCurve
    myCurve.attach(&myPlot);

    //QVector peut être facilement manipulé, et peut être convertit en double[];
    QVector<double> x(5);
    QVector<double> y(5);
```

```
    //On entre des valeurs
    for(int i=0;i<5;i++)
    {
        x.append((double)i);
        y.append((double)(5-i));
    }
    myCurve.setData(x.data(),y.data(),x.size());

    //Il faut rafraîchir l'affichage grâce à la méthode replot(), quand les données ont changé.
    myPlot.replot();
    myPlot.show();
    return app.exec();
}
```

Vous remarquerez que les axes se sont automatiquement ajustés aux données.

Comment changer la couleur de ma courbe ?

Auteurs : gulish ,

Il faut utiliser la méthode QwtPlotCurve::setPen().

```
#include <QApplication>

#include <qwt_plot.h>
#include <qwt_plot_curve.h>

int main(int argc, char** argv)
{
    QApplication app(argc,argv);
    QwtPlot myPlot;
    QwtPlotCurve myCurve;
    //indique à myPlot qu'il doit afficher myCurve
    myCurve.attach(&myPlot);

    //QVector peut être facilement manipulé, et peut être convertit en double[];
    QVector<double> x(5);
    QVector<double> y(5);
    //On entre des valeurs
    for(int i=0;i<5;i++)
    {
        x.append((double)i);
        y.append((double)(5-i));
    }
    myCurve.setData(x.data(),y.data(),x.size());
    //courbe en rouge
    myCurve.setPen(QPen(Qt::red));
    //Il faut rafraîchir l'affichage grâce à la méthode replot(), quand les données ont changé.
    myPlot.replot();
    myPlot.show();
    return app.exec();
}
```

Comment ajouter une grille à mon repère ?

Auteurs : gulish ,

Il est inutile d'ajouter une multitude de courbes pour faire apparaître une grille sur notre repère. QwtPlotGrid permet d'ajouter facilement une grille à notre repère. De plus, cette grille s'adapte automatiquement aux nouvelles échelles, et peut être personnalisée.

```
#include <QApplication>
#include <qwt_plot.h>
#include <qwt_plot_curve.h>
#include <qwt_plot_grid.h>

int main(int argc, char** argv)
{
    QApplication app(argc,argv);
    QwtPlot myPlot;
    QwtPlotCurve myCurve;
    QwtPlotGrid myGrid;
    //indique à myPlot qu'il doit afficher myCurve
    myCurve.attach(&myPlot);

    //QVector peut être facilement manipulé, et peut être convertit en double[];
    QVector<double> x(5);
    QVector<double> y(5);

    //On entre des valeurs
    for(int i=0;i<5;i++)
    {
        x.append((double)i);
        y.append((double)(5-i));
    }

    myCurve.setData(x.data(),y.data(),x.size());

    //indique à myPlot qu'il doit afficher myGrid
    myGrid.attach(&myPlot);

    //Il faut rafraîchir l'affichage grâce à la méthode replot(), quand les données ont changé.
    myPlot.replot();
    myPlot.show();
    return app.exec();
}
```

Comment ajouter des grilles seulement horizontales/verticales ?

Auteurs : gulish ,

On peut facilement réaliser ce type de grille grâce aux méthodes QwtPlotGrid::enableX() et QwtPlotGrid::enableY().

```
#include <QApplication>
#include <QHBoxLayout>
#include <qwt_plot.h>
#include <qwt_plot_curve.h>
#include <qwt_plot_grid.h>

int main(int argc, char** argv)
{
    QApplication app(argc,argv);
    QWidget myWidget;

    QwtPlot myLeftPlot;
    QwtPlotGrid myLeftGrid;
    myLeftGrid.enableX(false);//On affiche seulement les lignes horizontales
    myLeftGrid.attach(&myLeftPlot);

    QwtPlot myRightPlot;
    QwtPlotGrid myRightGrid;
    myRightGrid.enableY(false);//On affiche seulement les lignes verticales
    myRightGrid.attach(&myRightPlot);
```

```
    QHBoxLayout myLayout(&myWidget);
    myWidget.setLayout(&myLayout);
    myLayout.addWidget(&myLeftPlot);
    myLayout.addWidget(&myRightPlot);

    myLeftPlot.replot();
    myRightPlot.replot();
    myWidget.show();
    return app.exec();
}
```

Comment spécifier des valeurs à mes axes ?

Auteurs : gulish ,

Les axes peuvent simplement se personnaliser grâce aux fonctions setAxisScale(axe,min,max,step) et setAxisTitle(axe,titre). Dans la majorité des cas, la grille s'ajuste automatique sur l'échelle du repère.

```
#include <QApplication>

#include <qwt_plot.h>
#include <qwt_plot_curve.h>
#include <qwt_plot_grid.h>

int main(int argc, char** argv)
{
    QApplication app(argc,argv);

    QwtPlot myPlot;
    QwtPlotGrid myGrid;
    myGrid.attach(&myPlot);

    //l'axe x ira de -15 à +35, en affichant les valeurs de 5 en 5
    myPlot.setAxisScale(QwtPlot::xBottom,-15.0,35.0,5.0);
    myPlot.setAxisScale(QwtPlot::yLeft,-70.0,-35.0,5.0);

    myPlot.setAxisTitle(QwtPlot::xBottom,QString("My x values"));
    //On peut bien sûr utiliser les mêmes propriétés que QString
    myPlot.setAxisTitle(QwtPlot::yLeft,QString("<u>My y values</u>"));

    myPlot.replot();
    myPlot.show();

    return app.exec();
}
```

Comment ajouter un outil de zoom sur mon repère ?

Auteurs : gulish ,

Il existe une classe dédiée au zoom nommée QwtPlotZoomer. Elle permet de créer très simplement un outil de zoom, et d'autres fonctionnalités. On peut zoomer successivement sur une zone. Le zoom s'effectue en faisant un cliquer-glisser sur une zone du repère, le clic sur la molette permet de revenir au zoom précédent, et le clic droit permet de revenir au repère initial. La grille et les axes s'ajustent avec le zoom.

```
#include <QApplication>
#include <qwt_plot.h>
#include <qwt_plot_curve.h>
```

```
#include <qwt_plot_grid.h>
#include <qwt_plot_zoomer.h>
int main(int argc, char** argv)
{
    QApplication app(argc,argv);

    QwtPlot myPlot;
    QwtPlotGrid myGrid;
    myGrid.attach(&myPlot);
    QwtPlotZoomer myZoomer(myPlot.canvas());

    myPlot.replot();
    myPlot.show();

    return app.exec();
}
```