eBook Gratuit

APPRENEZ Qt

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#qt

Table des matières

À propos

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: qt

It is an unofficial and free Qt ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Qt.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapitre 1: Démarrer avec Qt

Remarques

Comme indiqué dans la documentation officielle , Qt est un framework de développement d'applications multi plates-formes pour ordinateurs de bureau, embarqués et mobiles. Les plates-formes prises en charge incluent Linux, OS X, Windows, VxWorks, QNX, Android, iOS, BlackBerry, Sailfish OS et autres.

Cette section fournit une vue d'ensemble de ce qu'est Qt et de la raison pour laquelle un développeur peut vouloir l'utiliser.

Il devrait également mentionner tous les sujets importants de Qt et les relier aux sujets connexes. Étant donné que la documentation de qt est nouvelle, vous devrez peut-être créer des versions initiales de ces rubriques connexes.

Versions

Version	Date de sortie
Qt 3.0	2001-10-16
Qt 3.3	2004-02-05
Qt 4.1	2005-12-20
Qt 4.8	2011-12-15
Qt 5.0	2012-12-19
Qt 5.6	2016-03-16
Qt 5.7	2016-06-16
Qt 5.8	2017-01-23
Qt 5.9	2017-05-31

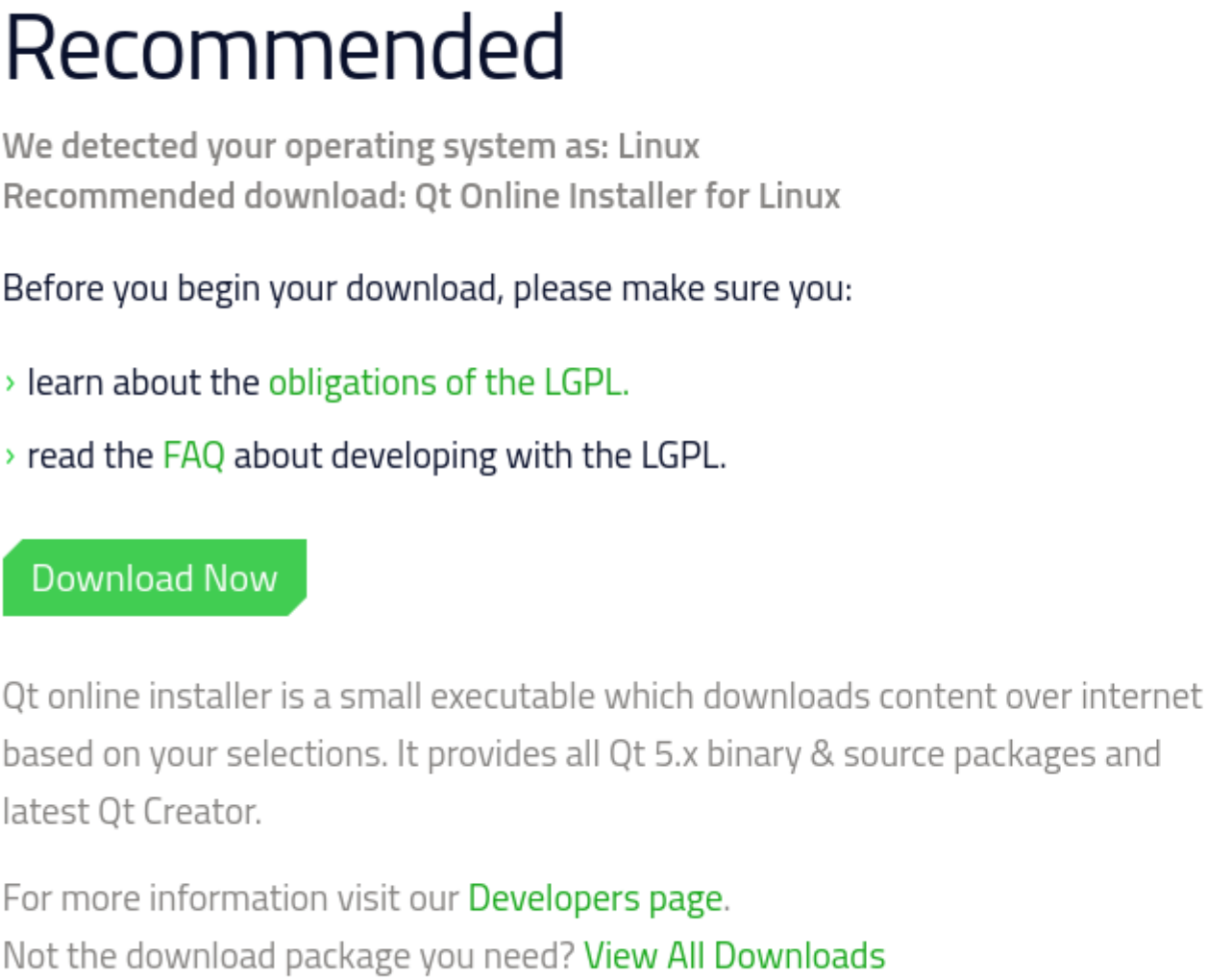
Examples

Installation et configuration sous Windows et Linux

Téléchargez la version Open Source de Qt pour Linux

Allez sur https://www.qt.io/download-open-source/ et cliquez sur Download Now, assurez-vous de

télécharger le programme d'installation de Qt pour Linux.



Un fichier nommé qt-unified-linux-x-online.run sera téléchargé, puis ajoutera une autorisation d'exécution

```
chmod +x qt-unified-linux-x-online.run
```

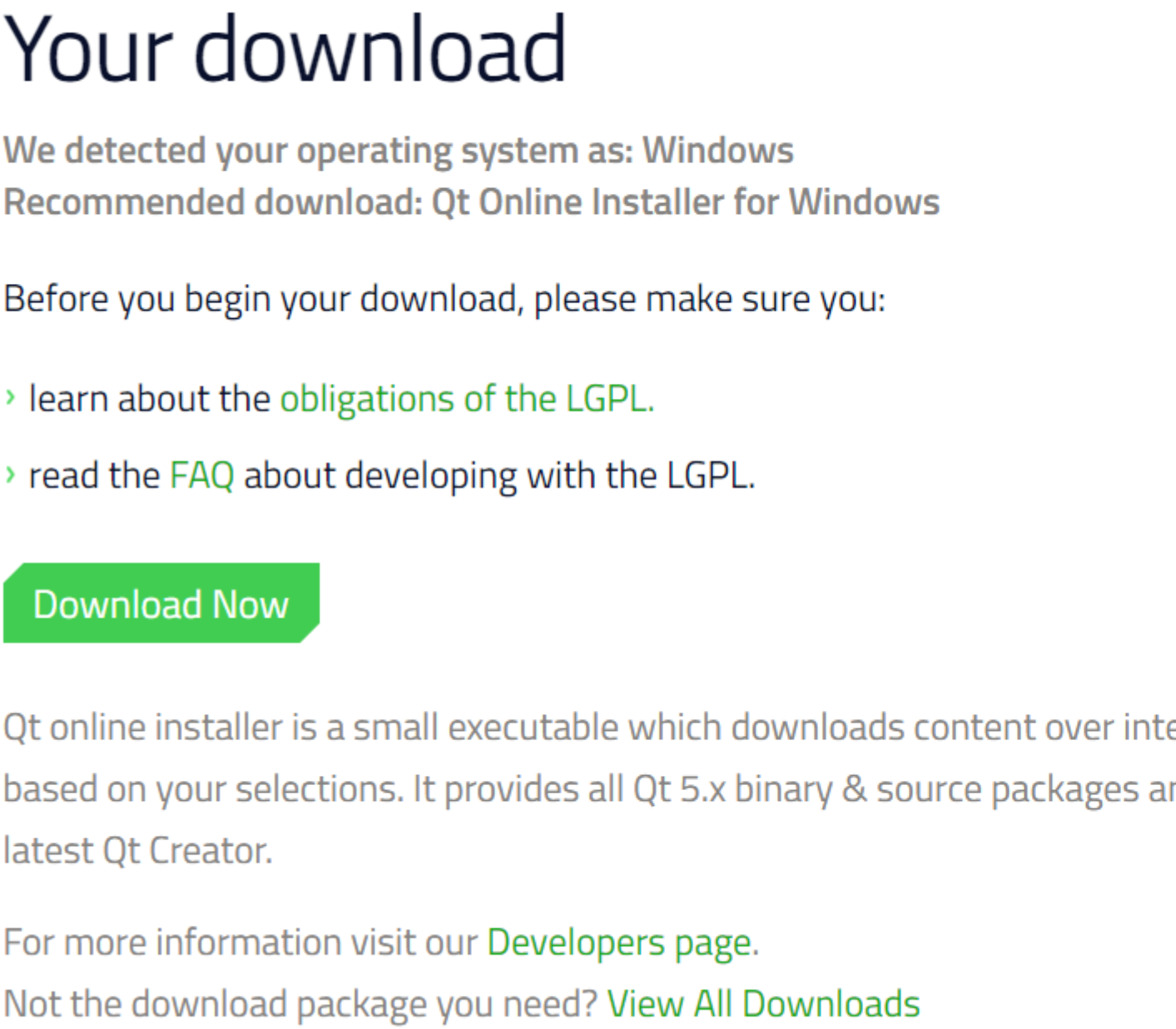
N'oubliez pas de changer 'x' pour la version actuelle du programme d'installation. Puis lancez l'installateur

```
./qt-unified-linux-x-online.run
```

Télécharger la version Qt pour Windows Open Source

Allez sur https://www.qt.io/download-open-source/ . La capture d'écran suivante montre la page

de téléchargement sous Windows:



Ce que vous devez faire maintenant dépend de l'EDI que vous allez utiliser. Si vous envisagez d'utiliser Qt Creator, qui est inclus dans le programme d'installation, cliquez simplement sur Télécharger maintenant et lancez l'exécutable.

Si vous utilisez Qt dans Visual Studio, le bouton Télécharger maintenant devrait également fonctionner. Assurez-vous que le fichier téléchargé s'appelle qt-opensource-windows-x86-msvc2015_64-xxxexe ou qt-opensource-windows-x86-msvc2015_32-xxxexe (où xxx est la version de Qt, par exemple 5.7.0). Si ce n'est pas le cas, cliquez sur Afficher tous les téléchargements et sélectionnez l'une des quatre premières options sous Windows Host.

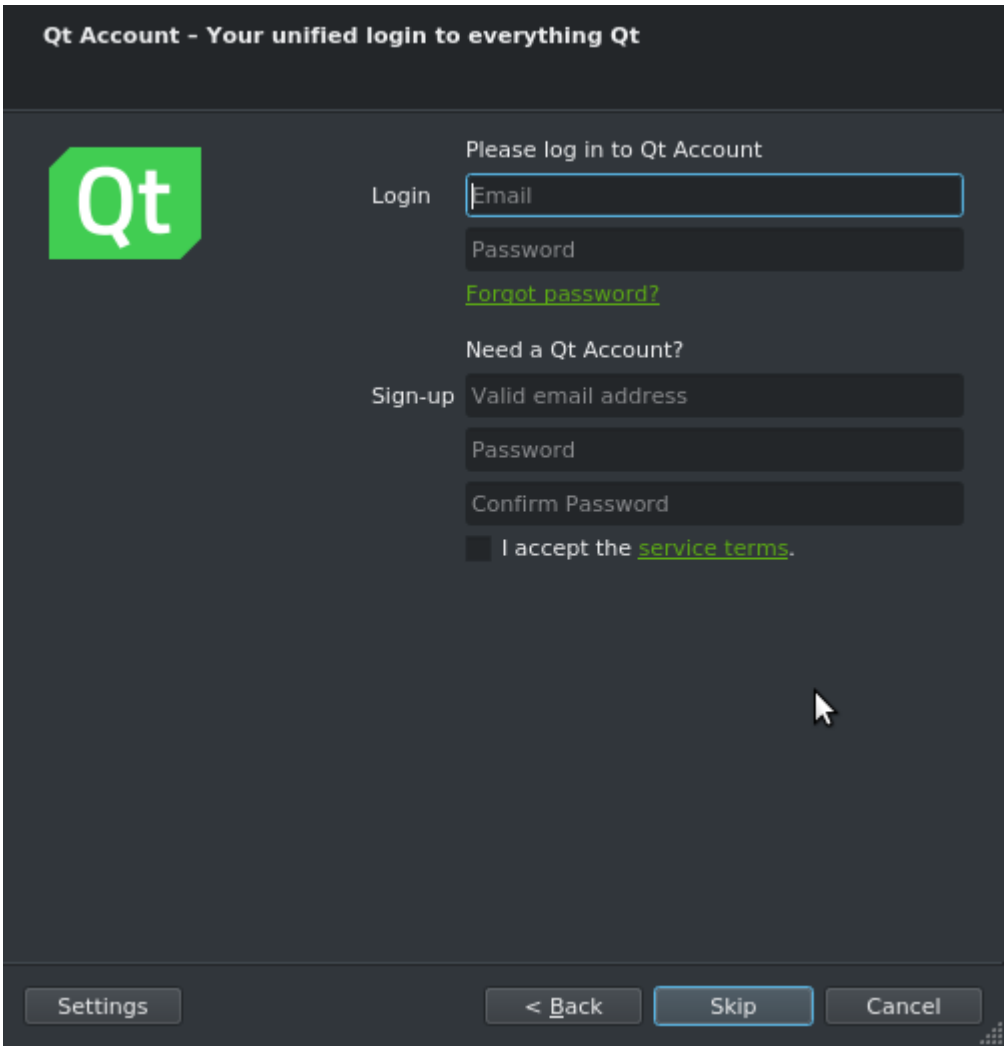
Si vous envisagez d'utiliser Qt dans Code :: Blocks, cliquez sur Afficher tous les téléchargements et sélectionnez Qt xxx pour Windows 32 bits (MinGW xxx, 1,2 Go) sous hôte Windows.

Une fois que vous avez téléchargé le fichier d'installation approprié, exécutez le fichier exécutable et suivez les instructions ci-dessous. Notez que vous devez être un administrateur pour installer Qt. Si vous n'êtes pas administrateur, vous pouvez trouver plusieurs solutions alternatives ici .

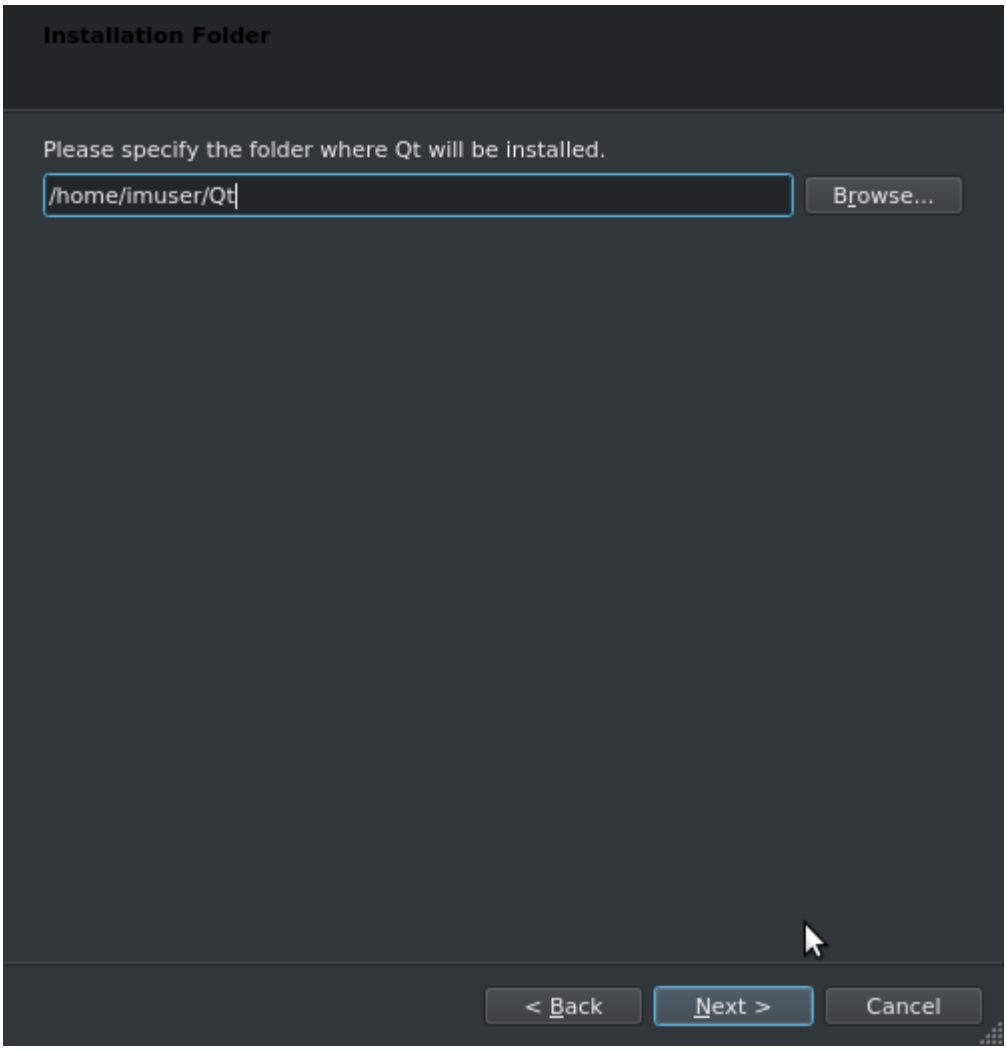
Installez Qt dans n'importe quel système opérationnel

Une fois que vous avez téléchargé Qt et ouvert le programme d'installation, la procédure d'installation est la même pour tous les systèmes opérationnels, bien que les captures d'écran puissent être un peu différentes. Les captures d'écran fournies ici proviennent de Linux.

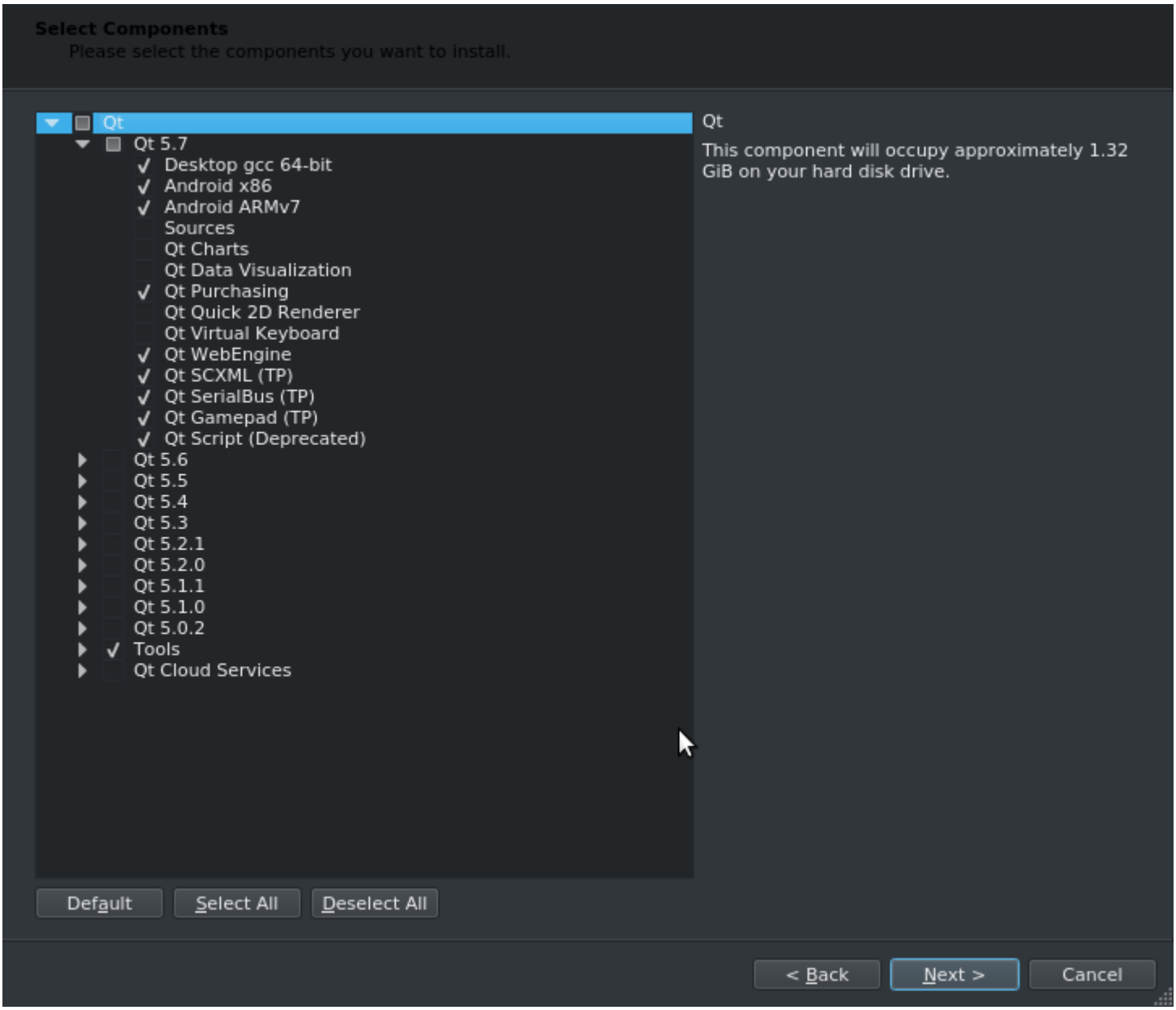
Connectez-vous avec un compte Qt existant ou créez-en un nouveau:



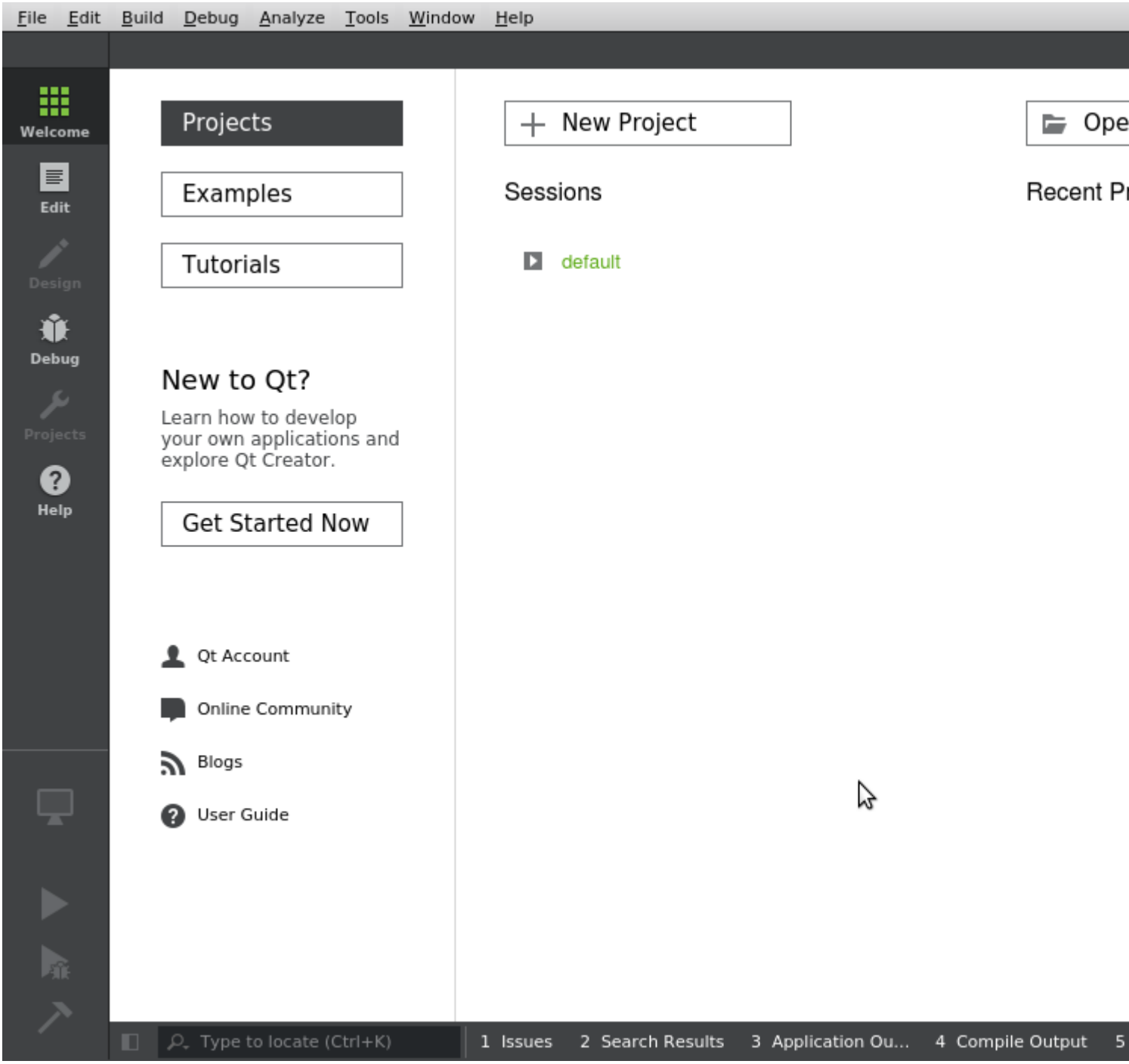
Sélectionnez un chemin pour installer les bibliothèques et les outils Qt



Sélectionnez la version de la bibliothèque et les fonctionnalités souhaitées



Une fois le téléchargement terminé et l'installation terminée, accédez au répertoire d'installation de Qt et lancez Qt Creator ou exécutez-le directement à partir de la ligne de commande.



Bonjour le monde

Dans cet exemple, nous créons et affichons simplement un bouton-poussoir dans un cadre de fenêtre sur le bureau. Le bouton poussoir aura le label `Hello world!`

Cela représente le programme Qt le plus simple possible.

Tout d'abord, nous avons besoin d'un fichier de projet:

helloworld.pro

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
```

```
TARGET = helloworld
TEMPLATE = app

SOURCES += main.cpp
```

- QT est utilisé pour indiquer quelles bibliothèques (modules Qt) sont utilisées dans ce projet. Comme notre première application est une petite interface graphique, nous aurons besoin de QtCore et de QtGui. Comme Qt5 sépare QtWidgets de QtGui, nous avons besoin d'ajouter `greaterThan` ligne plus grande afin de la compiler avec Qt5.
- CIBLE est le nom de l'application ou de la bibliothèque.
- TEMPLATE décrit le type à construire. Il peut s'agir d'une application (app), d'une bibliothèque (lib) ou simplement de sous-répertoires (sous-répertoires).
- SOURCES est une liste de fichiers de code source à utiliser lors de la création du projet.

Nous avons également besoin du fichier main.cpp contenant une application Qt:

main.cpp

```
#include <QApplication>
#include <QPushButton>

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

    QPushButton button ("Hello world!");
    button.show();

    return a.exec(); // .exec starts QApplication and related GUI, this line starts 'event
loop'
}
```

- QApplication object. Cet objet gère les ressources à l'échelle de l'application et est nécessaire pour exécuter tout programme Qt doté d'une interface graphique. Il nécessite argv et args car Qt accepte quelques arguments de ligne de commande. Lors de l'appel de `a.exec()` la boucle d'événement Qt est lancée.
- Objet QPushButton. Le bouton poussoir avec l'étiquette `Hello world!` . La ligne suivante, `button.show()` , montre le bouton-poussoir sur l'écran dans son propre cadre de fenêtre.

Enfin, pour exécuter l'application, ouvrez une invite de commande et entrez le répertoire dans lequel vous avez le fichier .cpp du programme. Tapez les commandes shell suivantes pour générer le programme.

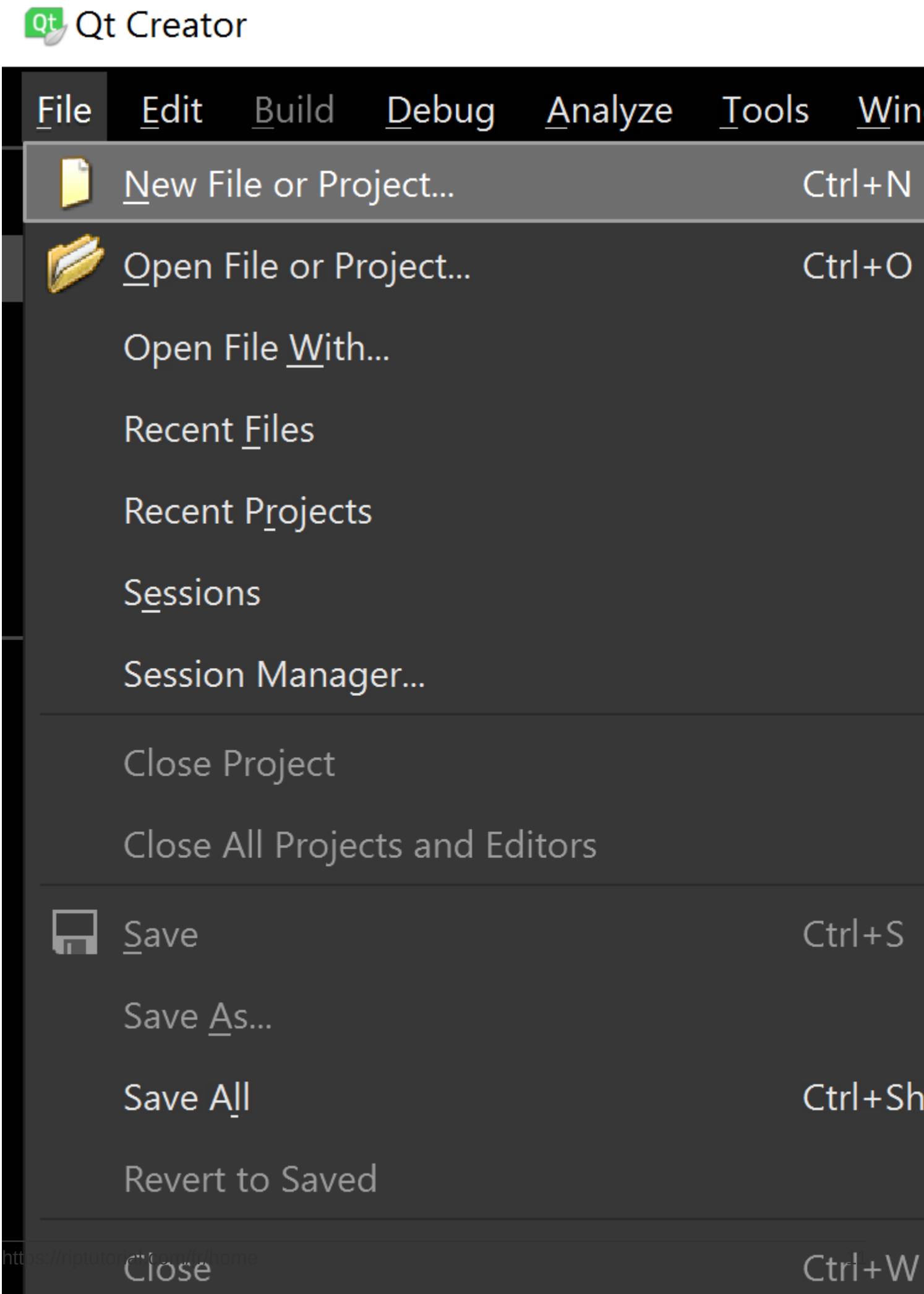
```
qmake -project
qmake
make
```

Application de base avec QtCreator et QtDesigner

QtCreator est actuellement le meilleur outil pour créer une application Qt. Dans cet exemple, nous

allons voir comment créer une application Qt simple qui gère un bouton et écrit du texte.

Pour créer une nouvelle application, cliquez sur Fichier-> Nouveau fichier ou projet:


Qt Creator
File
Edit
Build
Debug
Analyze
Tools
Win
New File or Project...
Ctrl+N
Open File or Project...
Ctrl+O
Open File With...
Recent Files
Recent Projects
Sessions
Session Manager...
Close Project
Close All Projects and Editors
Save
Ctrl+S
Save As...
Save All
Ctrl+Sh
Revert to Saved
Close
Ctrl+W

qui initialise l'interface utilisateur.

Ensuite, nous pouvons créer le MainWindow :: whenButtonIsClicked () dans notre classe .cpp qui pourrait changer le texte de l'étiquette comme ceci:

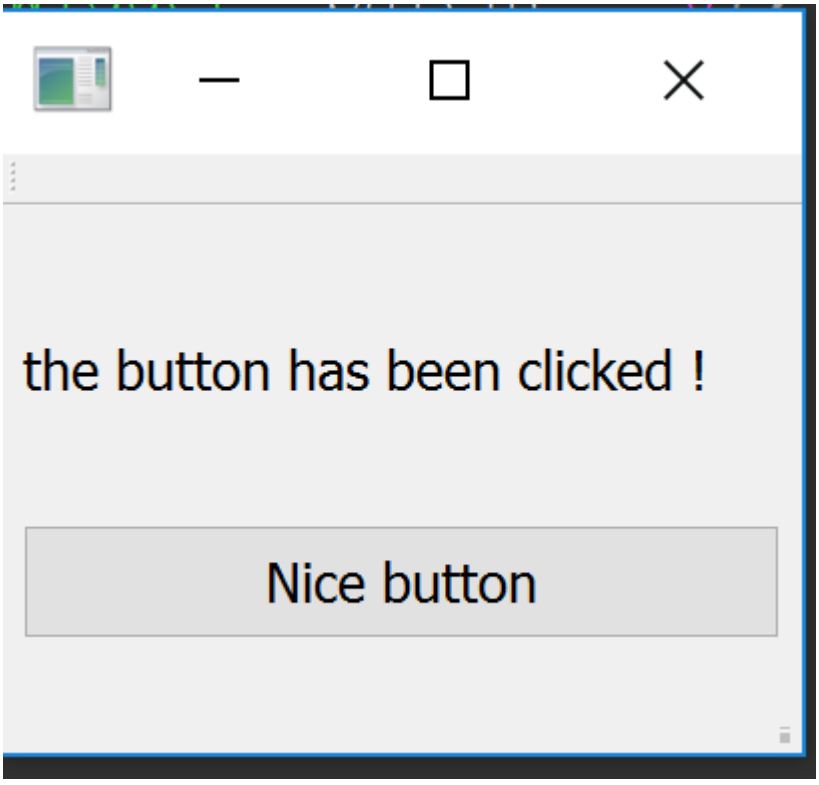
```
void MainWindow::whenButtonIsClicked()
{
    ui->label->setText("the button has been clicked !");
}
```

Et dans notre mainwindow.h, nous devons ajouter:

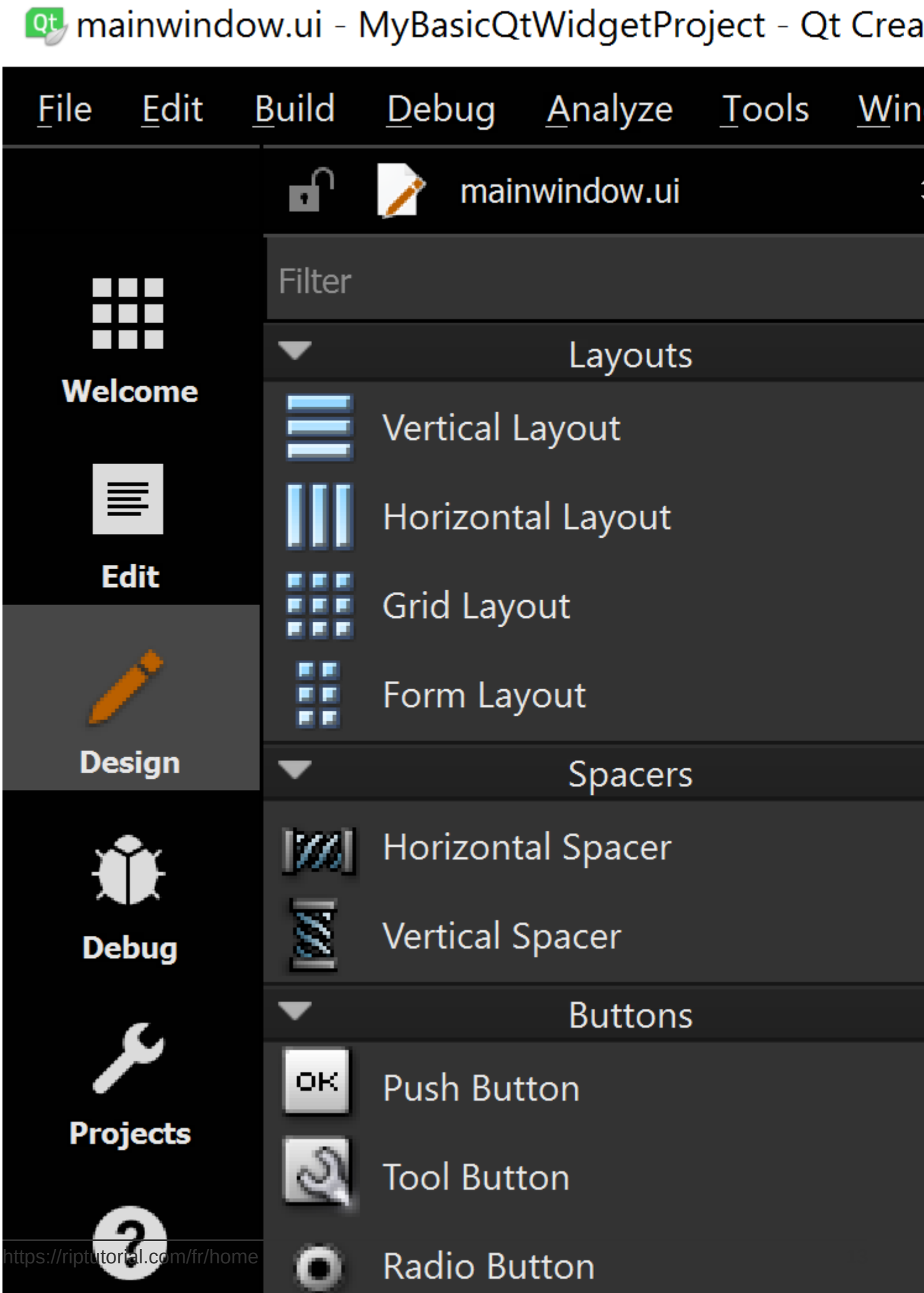
```
public slots:
    void whenButtonIsClicked();
```

Les slots publics signifient que cette méthode peut être appelée lorsqu'un signal est reçu. connecter le signal lorsque nous cliquons sur le bouton et une méthode pour appeler.

Alors maintenant, si nous exécutons notre application et cliquez sur le bouton, nous obtenons:



Ce qui signifie que notre connexion fonctionne. Mais avec Qt Designer, nous avons un moyen encore plus simple de le faire. Si vous voulez faire autrement, retirez la connexion pour déconnecter le bouton (parce que nous allons le connecter différemment), revenez à mainwindow.ui et cliquez avec le bouton droit sur le bouton. Cliquez sur Aller à l'emplacement ..., sélectionnez cliqué () et appuyez sur OK.


mainwindow.ui - MyBasicQtWidgetProject - Qt Crea
File
Edit
Build
Debug
Analyze
Tools
Win
mainwindow.ui
Filter
Welcome
Edit
Design
Debug
Projects
Layouts
Vertical Layout
Horizontal Layout
Grid Layout
Form Layout
Spacers
Horizontal Spacer
Vertical Spacer
Buttons
OK
Push Button
Tool Button
Radio Button
https://riptutorial.com/fr/home

qui est une classe géniale qui peut convertir beaucoup de choses dans beaucoup d'autres choses. Donc, ajoutez un int qui augmente lorsque vous appuyez sur le bouton.

Donc le .h:

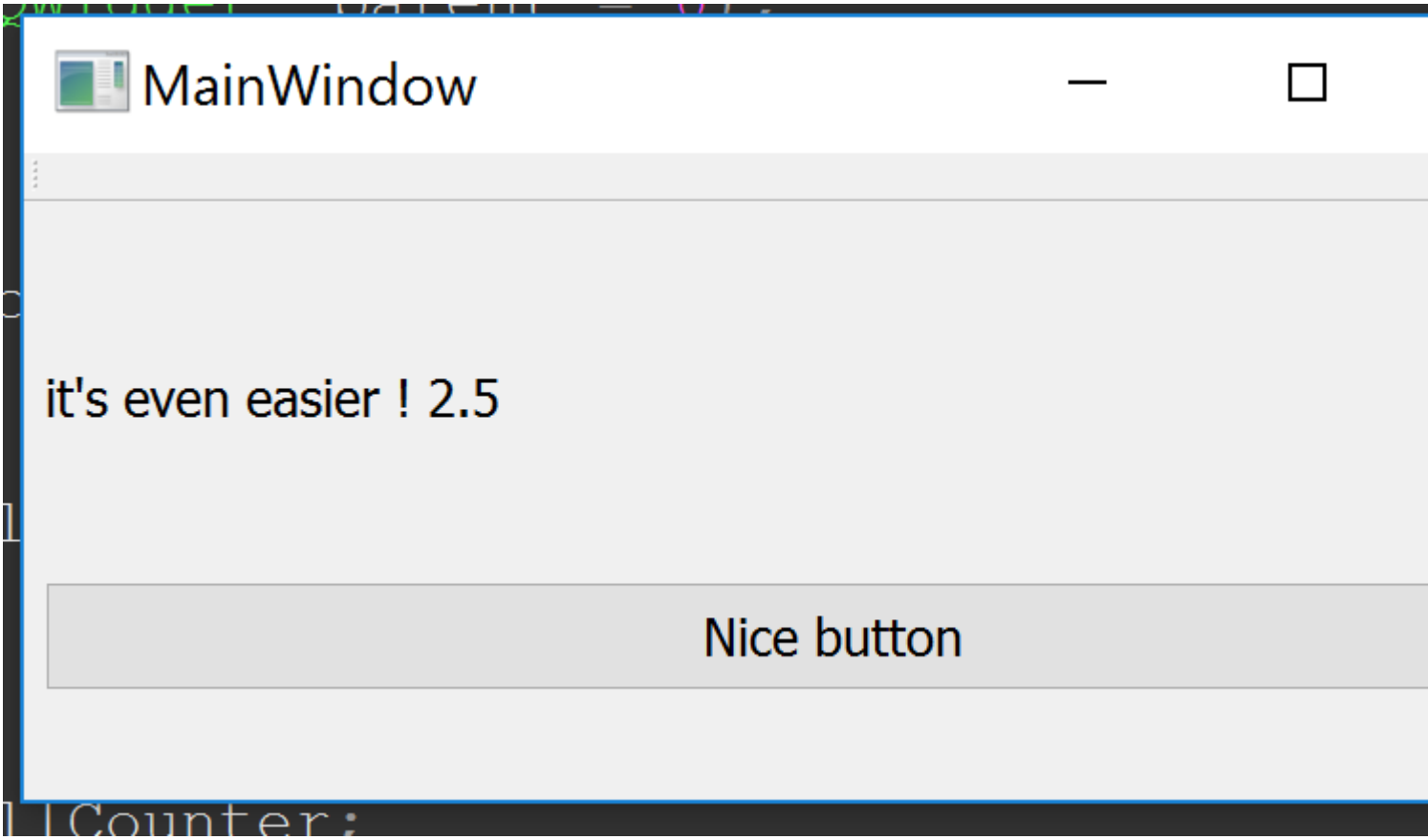
```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

public slots:
    void whenButtonIsClicked();

private slots:
    void on_pushButton_clicked();

private:
    Ui::MainWindow  *ui;
    double          _smallCounter;
};

#endif // MAINWINDOW_H
```

Le .cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
//    connect(ui->pushButton, SIGNAL(clicked(bool)), this, SLOT(whenButtonIsClicked()));
    _smallCounter = 0.0f;
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::whenButtonIsClicked()
{
    ui->label->setText("the button has been clicked !");
}
```

```
void MainWindow::on_pushButton_clicked()
{
    _smallCounter += 0.5f;
    ui->label->setText("it's even easier ! " + QVariant(_smallCounter).toString());
}
```

Et maintenant, nous pouvons enregistrer et exécuter à nouveau. Chaque fois que vous cliquez sur le bouton, il affiche "c'est encore plus facile!" Avec la valeur de _smallCounter. Donc, vous devriez avoir quelque chose comme:



Ce tutoriel est terminé. Si vous voulez en savoir plus sur Qt, voyons les autres exemples et la documentation de Qt sur la documentation StackOverflow ou la documentation Qt

Lire Démarrer avec Qt en ligne: https://riptutorial.com/fr/qt/topic/902/demarrer-avec-qt

Chapitre 2: A propos de l'utilisation des mises en page, du widget

Introduction

Les mises en page sont nécessaires dans chaque application Qt. Ils gèrent l'objet, leur position, leur taille, comment ils sont redimensionnés.

Remarques

De la documentation de mise en page Qt :

> Lorsque vous utilisez une présentation, vous n'avez pas besoin de transmettre un parent lors de la construction des widgets enfants. La mise en page va automatiquement réparer les widgets (en utilisant QWidget :: setParent ()) pour qu'ils soient des enfants du widget sur lequel la disposition est installée.

Alors faites :

```
QGroupBox *box = new QGroupBox("Information:", widget);
layout->addWidget(box);
```

ou faire:

```
QGroupBox *box = new QGroupBox("Information:", nullptr);
layout->addWidget(box);
```

est exactement la même chose.

Examples

Mise en page horizontale de base

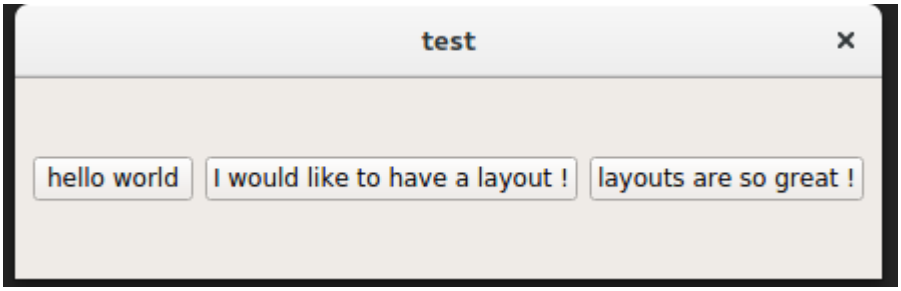
La disposition horizontale configure l'objet à l'intérieur horizontalement.

code de base:

```
#include <QApplication>

#include <QMainWindow>
#include <QWidget>
#include <QHBoxLayout>
#include <QPushButton>

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

```
    QApplication a(argc, argv);

    QMainWindow window;
    QWidget *widget = new QWidget(&window);
    QHBoxLayout *layout = new QHBoxLayout(widget);

    window.setCentralWidget(widget);
    widget->setLayout(layout);



    layout->addWidget(new QPushButton("hello world", widget));
    layout->addWidget(new QPushButton("I would like to have a layout !", widget));
    layout->addWidget(new QPushButton("layouts are so great !", widget));

    window.show();

    return a.exec();
}
```

cela va sortir:



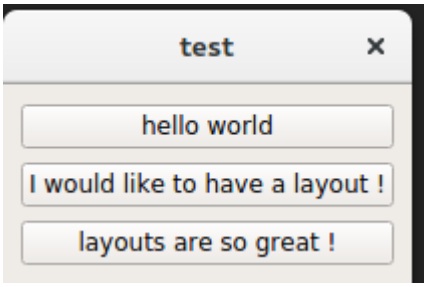
Mise en page verticale de base

La disposition verticale configure l'objet à l'intérieur verticalement.

```
#include "mainwindow.h"
#include <QApplication>

#include <QMainWindow>
#include <QWidget>
#include <QVBoxLayout>
#include <QPushButton>

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

    QMainWindow window;
    QWidget *widget = new QWidget(&window);
    QVBoxLayout *layout = new QVBoxLayout(widget);

    window.setCentralWidget(widget);
    widget->setLayout(layout);



    layout->addWidget(new QPushButton("hello world", widget));
    layout->addWidget(new QPushButton("I would like to have a layout !", widget));
```

```
    layout->addWidget(new QPushButton("layouts are so great !", widget));

    window.show();

    return a.exec();
}
```

sortie:



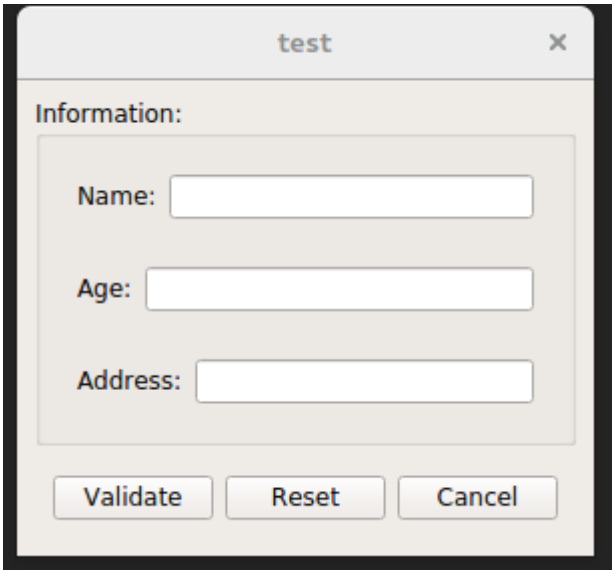
Combinaison de dispositions

Vous pouvez combiner plusieurs agencements grâce à d'autres QWidgets dans votre mise en page principale pour effectuer des effets plus spécifiques, comme un champ d'information: par exemple:

```
#include <QApplication>

#include <QMainWindow>
#include <QWidget>
#include <QVBoxLayout>
#include <QPushButton>
#include <QLabel>
#include <QLineEdit>
#include <QGroupBox>

#include <QTextEdit>

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

    QMainWindow window;
    QWidget *widget = new QWidget(&window);
    QVBoxLayout *layout = new QVBoxLayout(widget);

    window.setCentralWidget(widget);
    widget->setLayout(layout);

    QGroupBox *box = new QGroupBox("Information:", widget);
    QVBoxLayout *boxLayout = new QVBoxLayout(box);

    layout->addWidget(box);


    QWidget* nameWidget = new QWidget(box);
    QWidget* ageWidget = new QWidget(box);
    QWidget* addressWidget = new QWidget(box);
```

```
    boxLayout->addWidget(nameWidget);
    boxLayout->addWidget(ageWidget);
    boxLayout->addWidget(addressWidget);

    QHBoxLayout *nameLayout = new QHBoxLayout(nameWidget);
    nameLayout->addWidget(new QLabel("Name:"));
    nameLayout->addWidget(new QLineEdit(nameWidget));


    QHBoxLayout *ageLayout = new QHBoxLayout(ageWidget);
    ageLayout->addWidget(new QLabel("Age:"));
    ageLayout->addWidget(new QLineEdit(ageWidget));

    QHBoxLayout *addressLayout = new QHBoxLayout(addressWidget);
    addressLayout->addWidget(new QLabel("Address:"));
    addressLayout->addWidget(new QLineEdit(addressWidget));



    QWidget* validateWidget = new QWidget(widget);
    QHBoxLayout *validateLayout = new QHBoxLayout(validateWidget);
    validateLayout->addWidget(new QPushButton("Validate", validateWidget));
    validateLayout->addWidget(new QPushButton("Reset", validateWidget));
    validateLayout->addWidget(new QPushButton("Cancel", validateWidget));

    layout->addWidget(validateWidget);

    window.show();

    return a.exec();
}
```

va sortir:



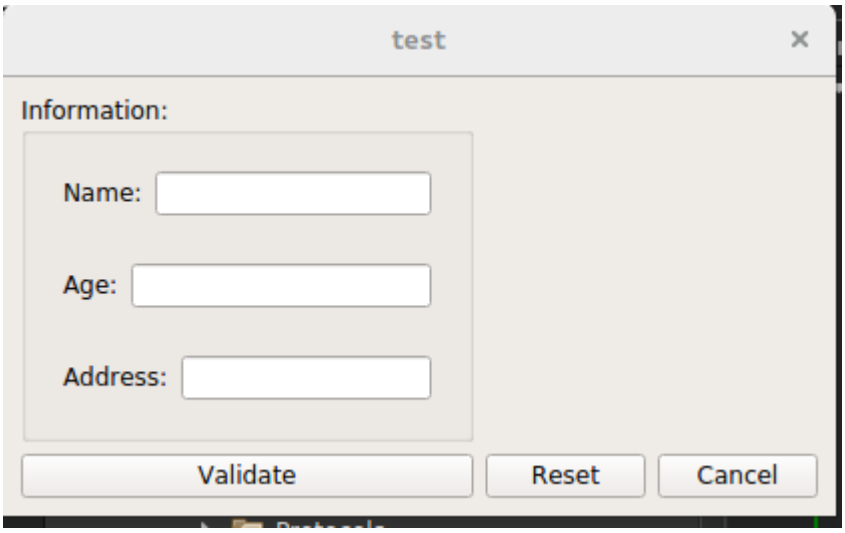
Exemple d'agencement de grille

La disposition de la grille est une mise en page puissante avec laquelle vous pouvez faire une disposition horizontale et verticale une fois.

Exemple:

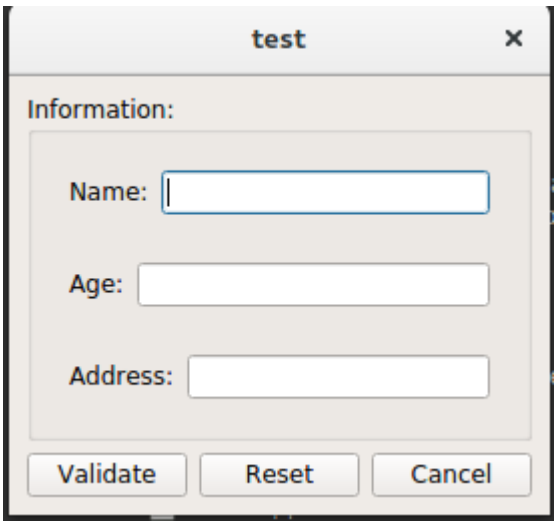
```
#include "mainwindow.h"
#include <QApplication>

#include <QMainWindow>
#include <QWidget>
#include <QVBoxLayout>
#include <QPushButton>
#include <QLabel>
#include <QLineEdit>
#include <QGroupBox>

#include <QTextEdit>

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

    QMainWindow window;
    QWidget *widget = new QWidget(&window);
    QGridLayout *layout = new QGridLayout(widget);

    window.setCentralWidget(widget);
    widget->setLayout(layout);

    QGroupBox *box = new QGroupBox("Information:", widget);
    layout->addWidget(box, 0, 0);

    QVBoxLayout *boxLayout = new QVBoxLayout(box);

    QWidget* nameWidget = new QWidget(box);
    QWidget* ageWidget = new QWidget(box);
    QWidget* addressWidget = new QWidget(box);

    boxLayout->addWidget(nameWidget);
    boxLayout->addWidget(ageWidget);
    boxLayout->addWidget(addressWidget);

    QHBoxLayout *nameLayout = new QHBoxLayout(nameWidget);
    nameLayout->addWidget(new QLabel("Name:"));
    nameLayout->addWidget(new QLineEdit(nameWidget));


    QHBoxLayout *ageLayout = new QHBoxLayout(ageWidget);
    ageLayout->addWidget(new QLabel("Age:"));
    ageLayout->addWidget(new QLineEdit(ageWidget));

    QHBoxLayout *addressLayout = new QHBoxLayout(addressWidget);
    addressLayout->addWidget(new QLabel("Address:"));
    addressLayout->addWidget(new QLineEdit(addressWidget));



    layout->addWidget(new QPushButton("Validate", widget), 1, 0);
    layout->addWidget(new QPushButton("Reset", widget), 1, 1);
    layout->addWidget(new QPushButton("Cancel", widget), 1, 2);


    window.show();

    return a.exec();
}
```

donner :



ainsi, vous pouvez voir que la boîte de groupe se trouve uniquement dans la première colonne et la première ligne car l'addWidget était `layout->addWidget(box, 0, 0);`

Cependant, si vous le modifiez en `layout->addWidget(box, 0, 0, 1, 3);` , les nouveaux 0 et 3 représentent le nombre de lignes et de colonnes que vous souhaitez pour votre widget.



exactement comme vous avez créé une disposition horizontale puis verticale dans un sous-widget.

Lire A propos de l'utilisation des mises en page, du widget en ligne: https://riptutorial.com/fr/qt/topic/9380/a-propos-de-l-utilisation-des-mises-en-page--du-widget

Chapitre 3: CMakeLists.txt pour votre projet Qt

Examples

CMakeLists.txt pour Qt 5

Un fichier de projet CMake minimal utilisant Qt5 peut être:

```
cmake_minimum_required(VERSION 2.8.11)

project(myproject)

find_package(Qt5 5.7.0 REQUIRED COMPONENTS
    Core
)

set(CMAKE_AUTOMOC ON)

add_executable(${PROJECT_NAME}
    main.cpp
)

target_link_libraries(${PROJECT_NAME}
    Qt5::Core
)
```

`cmake_minimum_required` est appelée pour définir la version minimale requise pour CMake. La version minimale requise pour que cet exemple fonctionne est `2.8.11` - les versions précédentes de CMake ont besoin d'un code supplémentaire pour qu'une cible utilise Qt.

`find_package` est appelé pour rechercher une installation de Qt5 avec une version donnée - 5.7.0 dans l'exemple - et les composants souhaités - le module Core dans l'exemple. Pour obtenir une liste des modules disponibles, voir Qt Documentation . Qt5 est marqué comme `REQUIRED` dans ce projet. Le chemin d'accès à l'installation peut être indiqué en définissant la variable `Qt5_DIR` .

`AUTOMOC` est un booléen spécifiant si CMake traitera automatiquement le préprocesseur Qt `moc` , c'est-à-dire sans avoir à utiliser la macro `QT5_WRAP_CPP()` .

Les autres variables "AUTOMOC-like" sont:

- `AUTOUIC` : un booléen spécifiant si CMake gérer le Qt `uic` générateur de code automatique, donc sans avoir à utiliser la `QT5_WRAP_UI()` macro.
- `AUTORCC` : un booléen spécifiant si CMake manipulera automatiquement le générateur de code Qt `rcc` , sans avoir à utiliser la macro `QT5_ADD_RESOURCES()` .

`add_executable` est appelé pour créer une cible exécutable à partir des fichiers sources donnés. La cible est alors liée aux modules de Qt répertoriés avec la commande `target_link_libraries` . À

partir de CMake 2.8.11, `target_link_libraries` avec les cibles importées de Qt gère les paramètres de l'éditeur de liens, ainsi que les répertoires et les options du compilateur.

Lire CMakeLists.txt pour votre projet Qt en ligne: https://riptutorial.com/fr/qt/topic/1991/cmakelists-txt-pour-votre-projet-qt

Chapitre 4: Communication entre QML et C ++

Introduction

Nous pouvons utiliser QML pour créer des applications hybrides, car il est beaucoup plus simple que C ++. Nous devrions donc savoir comment ils communiquent les uns avec les autres.

Examples

Appelez C ++ dans QML

Enregistrer les classes C ++ dans QML

Du côté du C ++, imaginez que nous avons une classe nommée `QmlCppBridge` , elle implémente une méthode appelée `printHello()` .

```
class QmlCppBridge : public QObject
{
    Q_OBJECT
public:
    Q_INVOKABLE static void printHello() {
        qDebug() << "Hello, QML!";
    }
};
```

Nous voulons l'utiliser dans le côté QML. Nous devrions enregistrer la classe en appelant `qmlRegisterType()` :

```
// Register C++ class as a QML module, 1 & 0 are the major and minor version of the QML module
qmlRegisterType<QmlCppBridge>("QmlCppBridge", 1, 0, "QmlCppBridge");
```

Dans QML, utilisez le code suivant pour l'appeler:

```
import QmlCppBridge 1.0    // Import this module, so we can use it in our QML script

QmlCppBridge {
    id: bridge
}
bridge.printHello();
```

Utilisation de `QQmlContext` pour injecter des classes ou des variables C ++ dans QML

Nous utilisons toujours la classe C ++ dans l'exemple précédent:

```
QQmlApplicationEngine engine;
QQmlContext *context = engine.rootContext();
```

```
// Inject C++ class to QML
context->setContextProperty(QStringLiteral("qmlCppBridge"), new QmlCppBridge(&engine));

// Inject C++ variable to QML
QString demoStr = QStringLiteral("demo");
context->setContextProperty(QStringLiteral("demoStr"), demoStr);
```

Côté QML:

```
qmlCppBridge.printHello();    // Call to C++ function
str: demoStr                  // Fetch value of C++ variable
```

> **Remarque:** Cet exemple est basé sur Qt 5.7. Je ne sais pas si cela convient aux versions précédentes de Qt.

Appelez QML en C ++

Pour appeler les classes QML en C ++, vous devez définir la propriété objectName.

Dans votre Qml:

```
import QtQuick.Controls 2.0

Button {
    objectName: "buttonTest"
}
```

Ensuite, dans votre C ++, vous pouvez obtenir l'objet avec `QObject.FindChild<QObject*>(QString)`

Comme ça:

```
QQmlApplicationEngine engine;
QQmlComponent component(&engine, QUrl(QLatin1String("qrc:/main.qml")));

QObject *mainPage = component.create();
QObject* item = mainPage->findChild<QObject *>("buttonTest");
```

Maintenant, vous avez votre objet QML dans votre C ++. Mais cela peut paraître inutile puisque nous ne pouvons pas vraiment obtenir les composants de l'objet.

Cependant, nous pouvons l'utiliser pour envoyer des **signaux** entre le QML et le C ++. Pour ce faire, vous devez ajouter un signal dans votre fichier QML comme ceci: `signal buttonClicked(string str)` . Une fois que vous créez cela, vous devez émettre le signal. Par exemple:

```
import QtQuick 2.0
import QtQuick.Controls 2.1

    Button {
        id: buttonTest
        objectName: "buttonTest"
```

```
        signal clickedButton(string str)
        onClicked: {
            buttonTest.clickedButton("clicked !")
        }
    }
```

Nous avons ici notre bouton qml. Quand on clique dessus, on passe à la méthode **onClicked** (une méthode de base pour les boutons qui est appelée quand on appuie sur le bouton). Ensuite, nous utilisons l'id du bouton et le nom du signal pour émettre le signal.

Et dans notre cpp, nous devons connecter le signal avec un slot. comme ça:

main.cpp

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QQmlComponent>

#include "ButtonManager.h"

int main(int argc, char *argv[])
{
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);
    QGuiApplication app(argc, argv);

    QQmlApplicationEngine engine;
    QQmlComponent component(&engine, QUrl(QLatin1String("qrc:/main.qml")));

    QObject *mainPage = component.create();
    QObject* item = mainPage->findChild<QObject *>("buttonTest");

    ButtonManager buttonManager(mainPage);
    QObject::connect(item, SIGNAL(clickedButton(QString)), &buttonManager,
SLOT(onButtonClicked(QString)));

    return app.exec();
}
```

Comme vous pouvez le voir, nous obtenons notre bouton qml avec `findChild` comme précédemment et nous connectons le signal à un gestionnaire de boutons qui est une classe créée et qui ressemble à ça. ButtonManager.h

```
#ifndef BUTTONMANAGER_H
#define BUTTONMANAGER_H

#include <QObject>

class ButtonManager : public QObject
{
    Q_OBJECT
public:
    ButtonManager(QObject* parent = nullptr);
public slots:
    void onButtonClicked(QString str);
};
```

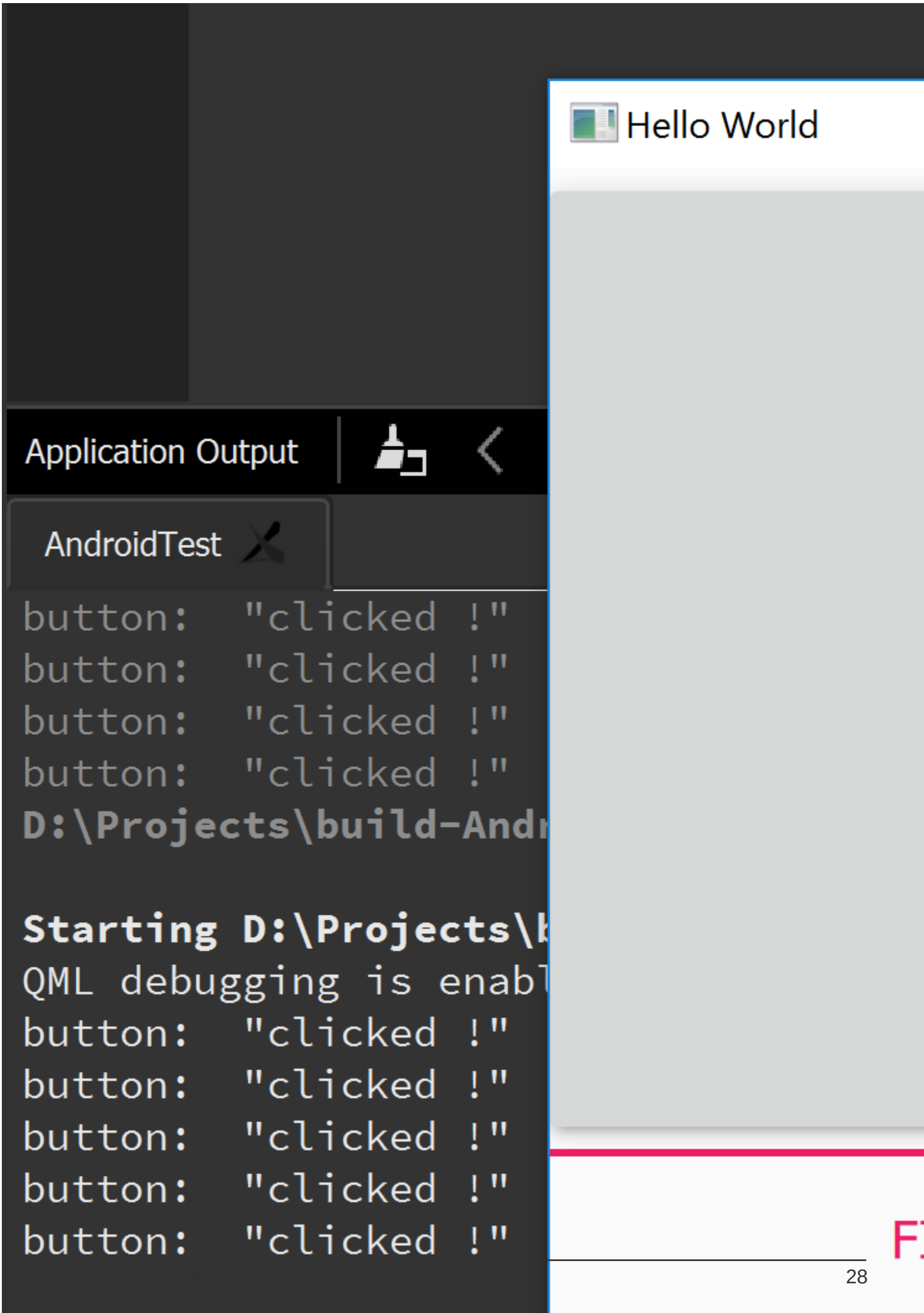
```
#endif // BUTTONMANAGER_H
```

ButtonManager.cpp

```
#include "ButtonManager.h"
#include <QDebug>

ButtonManager::ButtonManager(QObject *parent)
    : QObject(parent)
{

}

void ButtonManager::onButtonClicked(QString str)
{
    qDebug() << "button: " << str;
}
```

Donc, quand le signal sera reçu, il appellera la méthode `onButtonClicked` qui écrira `"button: clicked !"`

sortie:


Hello World
Application Output
AndroidTest
button: "clicked !"
button: "clicked !"
button: "clicked !"
button: "clicked !"
D:\Projects\build-Andr
Starting D:\Projects\b
QML debugging is enabl
button: "clicked !"
button: "clicked !"
button: "clicked !"
button: "clicked !"
button: "clicked !"

https://riptutorial.com/fr/qt/topic/8735/communication-entre-qml-et-c-plusplus

Chapitre 5: Construire QtWebEngine depuis le source

Introduction

Parfois, nous devons construire QtWebEngine à partir de sources pour une raison quelconque, par exemple pour le support mp3.

Examples

Construire sous Windows

Exigences

- Windows 10, veuillez **définir les paramètres régionaux de votre système en anglais** , sinon des erreurs peuvent survenir
- Visual Studio 2013 ou 2015
- QtWebEngine 5.7 code source (peut être téléchargé ici)
- Qt 5.7 installer la version, installez-le et ajoutez le dossier `qmake.exe` au chemin système
- Python 2, ajoutez le dossier `python.exe` au chemin du système
- Git, ajoute le dossier `git.exe` au chemin du système
- gperf, ajoute le dossier `gperf.exe` au chemin du système
- flex-bison, ajoutez le dossier `win_bison.exe` au chemin système et renommez-le en `bison.exe`

  Note: Je n'ai pas testé les versions de Visual Studio, toutes les versions de Qt. Prenons juste un exemple ici, les autres versions devraient être à peu près les mêmes.

Étapes pour construire

1. Décompressez le code source dans un dossier, appelons-le `ROOT`
2. Ouvrez l' `Developer Command Prompt for VS2013` et accédez au dossier `ROOT`
3. Exécutez `qmake WEBENGINE_CONFIG+=use_proprietary_codecs qtwebengine.pro` . Nous ajoutons ce drapeau pour activer le support mp3.
4. Exécuter `nmake`

   Remarque: Mp3 n'est pas pris en charge par QtWebEngine par défaut, en raison d'un problème de licence. Veuillez vous assurer d'obtenir une licence pour le codec que vous avez ajouté.

Lire Construire QtWebEngine depuis le source en ligne:
https://riptutorial.com/fr/qt/topic/8718/construire-qtwebengine-depuis-le-source

Chapitre 6: Déploiement d'applications Qt

Examples

Déploiement sur Windows

Qt fournit un outil de déploiement pour Windows: `windeployqt` . L'outil inspecte un exécutable d'application Qt pour ses dépendances aux modules Qt et crée un répertoire de déploiement avec les fichiers Qt nécessaires pour exécuter le fichier exécutable inspecté. Un script possible peut ressembler à:

```
set PATH=%PATH%;<qt_install_prefix>/bin
windeployqt --dir /path/to/deployment/dir /path/to/qt/application.exe
```

La commande `set` est appelée pour ajouter le répertoire `bin` de Qt à la variable d'environnement `PATH` . `windeployqt` s'appelle alors:

- Le chemin d'accès au répertoire de déploiement reçoit un argument facultatif fourni avec le paramètre `--dir` (par défaut, le chemin d'accès à `windeployqt`).
- Le chemin d'accès à l'exécutable à inspecter est donné en dernier argument.

Le répertoire de déploiement peut ensuite être fourni avec l'exécutable.

REMARQUE:

Si vous utilisez Qt5.7.0 pré-compilé avec vs2013 sous Windows *(ne savez pas si toutes les versions ont ce problème)* , il est possible que vous ayez besoin de copier manuellement `<QTDIR>\5.7\msvc2015\qml` dans votre répertoire bin de votre programme Sinon, le programme s'arrêtera automatiquement après le démarrage.

Voir aussi la documentation Qt .

Intégration avec CMake

Il est possible de lancer `windeployqt` et `macdeployqt` depuis CMake, mais il faut d'abord trouver le chemin d'accès aux exécutables:

```
# Retrieve the absolute path to qmake and then use that path to find
# the binaries
get_target_property(_qmake_executable Qt5::qmake IMPORTED_LOCATION)
get_filename_component(_qt_bin_dir "${_qmake_executable}" DIRECTORY)
find_program(WINDEPLOYQT_EXECUTABLE windeployqt HINTS "${_qt_bin_dir}")
find_program(MACDEPLOYQT_EXECUTABLE macdeployqt HINTS "${_qt_bin_dir}")
```

Pour que `windeployqt` trouve les bibliothèques Qt dans leur emplacement installé, le dossier doit être ajouté à `%PATH%` . Pour ce faire, pour une cible nommée `myapp` après avoir été construite:

```
add_custom_command(TARGET myapp POST_BUILD
    COMMAND "${CMAKE_COMMAND}" -E
        env PATH="${_qt_bin_dir}" "${WINDEPLOYQT_EXECUTABLE}"
            "$<TARGET_FILE:myapp>"
    COMMENT "Running windeployqt..."
)
```

Pour exécuter `macdeployqt` sur un bundle, cela se ferait comme `macdeployqt` :

```
add_custom_command(TARGET myapp POST_BUILD
    COMMAND "${MACDEPLOYQT_EXECUTABLE}"
        "$<TARGET_FILE_DIR:myapp>/../.."
        -always-overwrite
    COMMENT "Running macdeployqt..."
)
```

Déploiement sur Mac

Qt propose un outil de déploiement pour Mac: l'outil de déploiement Mac.

L'outil de déploiement Mac se trouve dans `QTDIR/bin/macdeployqt` . Il est conçu pour automatiser le processus de création d'un ensemble d'applications déployable contenant les bibliothèques Qt en tant que structures privées.

L'outil de déploiement mac déploie également les plugins Qt, selon les règles suivantes (sauf si l' **option -no-plugins est utilisée**):

- Le plugin de la plateforme est toujours déployé.
- Les versions de débogage des plugins ne sont pas déployées.
- Les plug-ins de concepteur ne sont pas déployés.
- Les plugins de format d'image sont toujours déployés.
- Le plug-in de support d'impression est toujours déployé.
- Les plug-ins de pilote SQL sont déployés si l'application utilise le module SQL Qt.
- Les plugins de script sont déployés si l'application utilise le module Qt Script.
- Le plug-in SVG icon est déployé si l'application utilise le module Qt SVG.
- Le plugin d'accessibilité est toujours déployé.

Pour inclure une bibliothèque tierce dans le regroupement d'applications, copiez la bibliothèque manuellement dans le regroupement, une fois le regroupement créé.

Pour utiliser l'outil `macdeployqt` , vous pouvez ouvrir le terminal et taper:

```
$ QTDIR/bin/macdeployqt <path to app file generated by build>/appFile.app
```

Le fichier d'application contiendra désormais toutes les bibliothèques Qt utilisées comme frameworks privés.

`macdeployqt` prend également en charge les options suivantes

Option	La description
-verbose = <0-3>	0 = pas de sortie, 1 = erreur / avertissement (par défaut), 2 = normal, 3 = débogage
-no-plugins	Ignorer le déploiement du plugin
-dmg	Créer une image de disque .dmg
-no-strip	Ne lancez pas 'strip' sur les binaires
-use-debug-libs	Déployer avec les versions de débogage des frameworks et plugins (implique -no-strip)
-exécutable =	Laisser l'exécutable donné utiliser également les frameworks déployés
-qmldir =	Déployer les importations utilisées par les fichiers .qml dans le chemin donné

Des informations détaillées peuvent être obtenues sur Qt Documentation

Déploiement sur Linux

Il existe un outil de déploiement pour Linux sur GitHub . Bien qu'il ne soit pas parfait, il est lié au wiki Qt. Il repose conceptuellement sur l'outil de déploiement Mac Qt et fonctionne de la même manière en fournissant une AppImage .

Étant donné qu'un fichier de bureau doit être fourni avec AppImage, `linuxdeployqt` peut l'utiliser pour déterminer les paramètres de la génération.

```
linuxdeployqt ./path/to/appdir/usr/share/application_name.desktop
```

Où le fichier de bureau spécifie l'exécutable à exécuter (avec `EXEC=`), le nom de l'application et une icône.

Lire Déploiement d'applications Qt en ligne: https://riptutorial.com/fr/qt/topic/5857/deploiement-d-applications-qt

Chapitre 7: En-tête sur QListView

Introduction

Le widget QListView fait partie des mécanismes de programmation Modèle / Vue de Qt. Fondamentalement, il permet d'afficher les éléments stockés dans un modèle sous la forme d'une liste. Dans cette rubrique, nous ne nous intéresserons pas aux mécanismes Model / View de Qt, mais plutôt à l'aspect graphique d'un widget View: le QListView, et surtout comment ajouter un en-tête au-dessus de cet objet via QPaintEvent. objet.

Examples

Déclaration QListView personnalisée

```
/*!
 * \class MainMenuListView
 * \brief The MainMenuListView class is a QListView with a header displayed
 *        on top.
 */
class MainMenuListView : public QListView
{
Q_OBJECT

    /*!
     * \class Header
     * \brief The header class is a nested class used to display the header of a
     *        QListView. On each instance of the MainMenuListView, a header will
     *        be displayed.
     */
    class Header : public QWidget
    {
    public:
        /*!
         * \brief Constructor used to defined the parent/child relation
         *        between the Header and the QListView.
         * \param parent Parent of the widget.
         */
        Header(MainMenuListView* parent);

        /*!
         * \brief Overridden method which allows to get the recommended size
         *        for the Header object.
         * \return The recommended size for the Header widget.
         */
        QSize sizeHint() const;

        protected:
        /*!
         * \brief Overridden paint event which will allow us to design the
         *        Header widget area and draw some text.
         * \param event Paint event.
         */
        void paintEvent(QPaintEvent* event);
```

```
    private:
        MainMenuListView* menu;    /*!< The parent of the Header. */
    };

public:
    /*!
     * \brief Constructor allowing to instanciate the customized QListView.
     * \param parent Parent widget.
     * \param header Text which has to be displayed in the header
     *        (Header by default)
     */
    MainMenuListView(QWidget* parent = nullptr, const QString& header = QString("Header"));

    /*!
     * \brief Catches the Header paint event and draws the header with
     *        the specified text in the constructor.
     * \param event Header paint event.
     */
    void headerAreaPaintEvent(QPaintEvent* event);

    /*!
     * \brief Gets the width of the List widget.
     *        This value will also determine the width of the Header.
     * \return The width of the custom QListView.
     */
    int headerAreaWidth();

 protected:
    /*!
     * \brief Overridden method which allows to resize the Header.
     * \param event Resize event.
     */
    void resizeEvent(QResizeEvent* event);

private:
   QWidget*    headerArea;    /*!< Header widget. */
   QString     headerText;    /*!< Header title. */
};
```

Implémentation de la QListView personnalisée

```
QSize MainMenuListView::Header::sizeHint() const
{
    // fontmetrics() allows to get the default font size for the widget.
    return QSize(menu->headerAreaWidth(), fontMetrics().height());
}

void MainMenuListView::Header::paintEvent(QPaintEvent* event)
{
    // Catches the paint event in the parent.
    menu->headerAreaPaintEvent(event);
}

MainMenuListView::MainMenuListView(QWidget* parent, const QString& header) :
QListView(parent), headerText(header)
{
    headerArea = new Header(this);
```

```
    // Really important. The view port margins define where the content
    // of the widget begins.
    setViewportMargins(0, fontMetrics().height(), 0, 0);
}

void MainMenuListView::headerAreaPaintEvent(QPaintEvent* event)
{
    // Paints the background of the header in gray.
    QPainter painter(headerArea);
    painter.fillRect(event->rect(), Qt::lightGray);

    // Display the header title in black.
    painter.setPen(Qt::black);

    // Writes the header aligned on the center of the widget.
    painter.drawText(0, 0, headerArea->width(), fontMetrics().height(), Qt::AlignCenter,
headerText);
}

int MainMenuListView::headerAreaWidth()
{
    return width();
}

void MainMenuListView::resizeEvent(QResizeEvent* event)
{
    // Executes default behavior.
    QListView::resizeEvent(event);

    // Really important. Allows to fit the parent width.
    headerArea->adjustSize();
}
```

Cas d'utilisation: déclaration MainWindow

```
class MainMenuListView;

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget* parent = 0);
    ~MainWindow();

private:
    MainMenuListView* menuA;
    MainMenuListView* menuB;
    MainMenuListView* menuC;
};
```

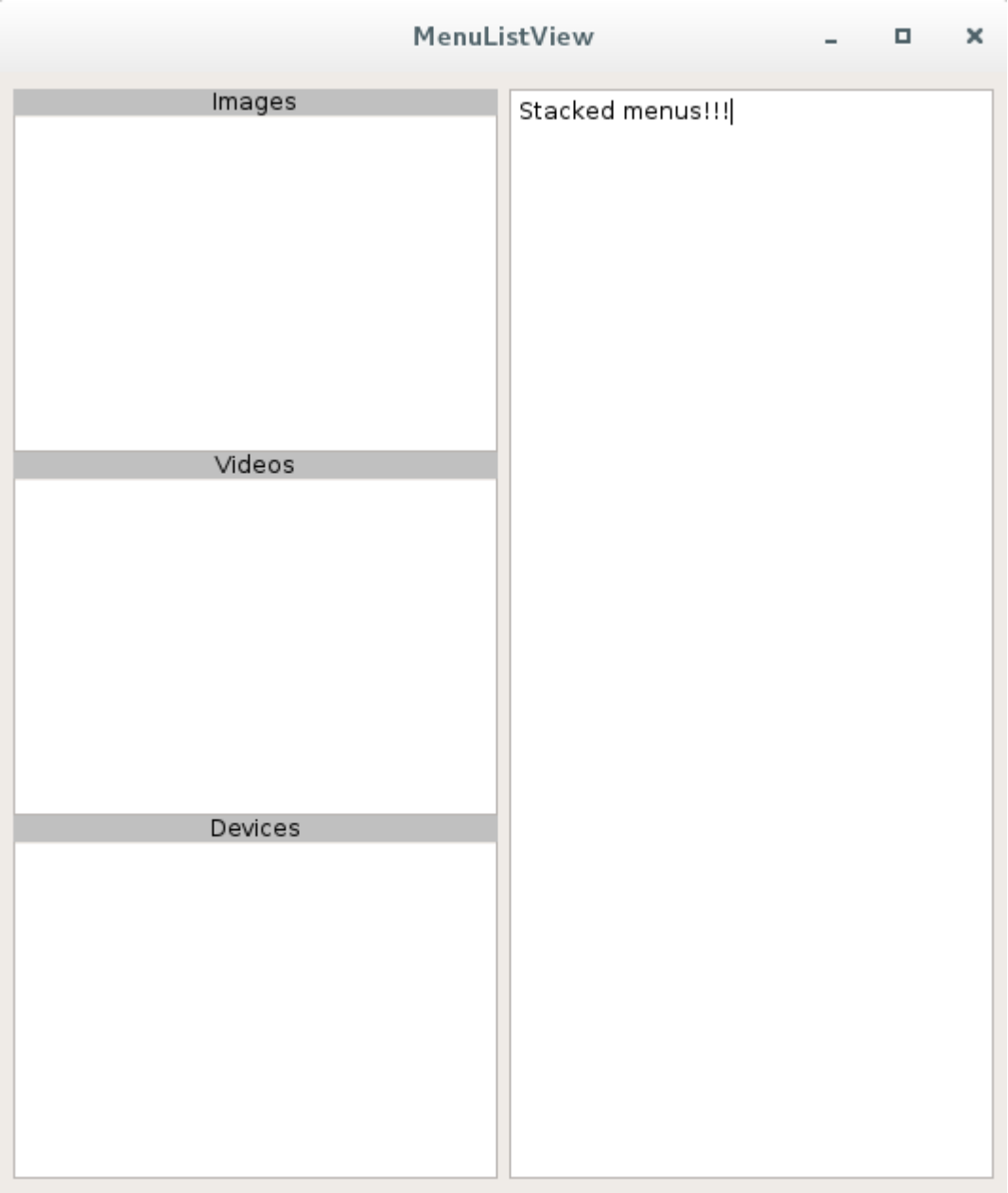
Cas d'utilisation: Implémentation

```
MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent)
{
    QWidget* w = new QWidget(this);
```

```
    QHBoxLayout* hbox = new QHBoxLayout();

    QVBoxLayout* vBox = new QVBoxLayout();
    menuA = new MainMenuListView(w, "Images");
    menuB = new MainMenuListView(w, "Videos");
    menuC = new MainMenuListView(w, "Devices");
    vBox->addWidget(menuA);
    vBox->addWidget(menuB);
    vBox->addWidget(menuC);
    vBox->setSpacing(0);
    hbox->addLayout(vBox);

    QPlainTextEdit* textEdit = new QPlainTextEdit(w);
    hbox->addWidget(textEdit);

    w->setLayout(hbox);
    setCentralWidget(w);

    move((QApplication::desktop()->screenGeometry().width() / 2) - (size().width() / 2),
         (QApplication::desktop()->screenGeometry().height() / 2) - (size().height() / 2));

}

MainWindow::~MainWindow() {}
```

Cas d'utilisation: exemple de sortie

Voici un exemple de sortie:



Comme vous pouvez le voir ci-dessus, cela peut être utile pour créer des menus empilés. Notez que cet exemple est trivial. Les deux widgets ont les mêmes contraintes de taille.

Lire En-tête sur QListView en ligne: https://riptutorial.com/fr/qt/topic/9382/en-tete-sur-qlistview

Chapitre 8: Modèle / vue

Examples

Une table simple en lecture seule pour afficher les données d'un modèle

Il s'agit d'un exemple simple d'affichage de données en lecture seule de nature tabulaire à l'aide du Framework / View Framework de Qt. Plus précisément, les `Qt Objects` QAbstractTableModel (sous-classés dans cet exemple) et QTableView sont utilisés.

Les implémentations des méthodes rowCount () , columnCount () , data () et headerData () sont nécessaires pour permettre à l'objet `QTableView` d'obtenir des informations sur les données contenues dans l'objet `QAbstractTableModel` .

La méthode `populateData()` été ajoutée à cet exemple pour fournir un moyen de renseigner l'objet `QAbstractTableModel` avec des données provenant d'une source arbitraire.

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
#include <QAbstractTableModel>

namespace Ui {
    class MainWindow;
}

class TestModel : public QAbstractTableModel
{
    Q_OBJECT

public:
    TestModel(QObject *parent = 0);

    void populateData(const QList<QString> &contactName,const QList<QString> &contactPhone);

    int rowCount(const QModelIndex &parent = QModelIndex()) const Q_DECL_OVERRIDE;
    int columnCount(const QModelIndex &parent = QModelIndex()) const Q_DECL_OVERRIDE;

    QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const Q_DECL_OVERRIDE;
    QVariant headerData(int section, Qt::Orientation orientation, int role = Qt::DisplayRole)
const Q_DECL_OVERRIDE;

private:
    QList<QString> tm_contact_name;
    QList<QString> tm_contact_phone;

};

class MainWindow : public QMainWindow
{
    Q_OBJECT
```

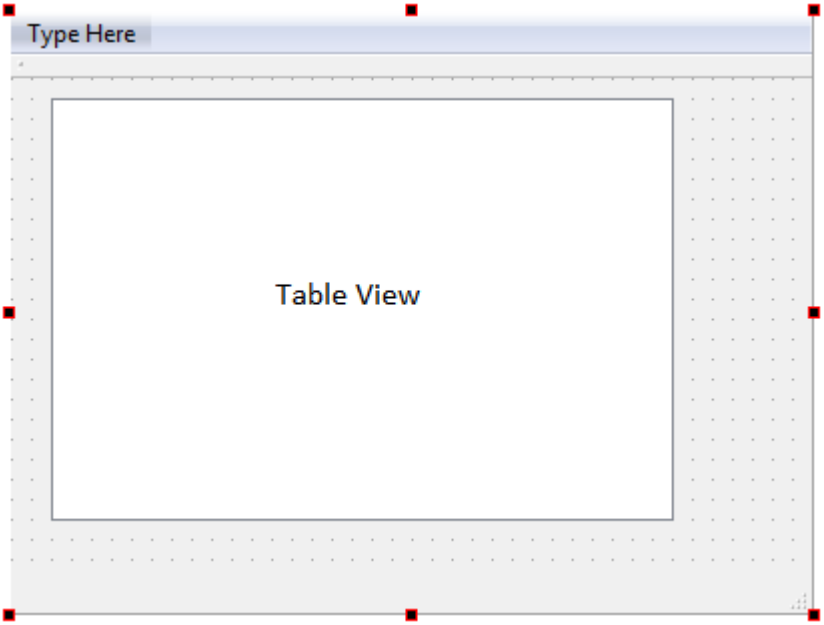
```
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;

};

#endif // MAINWINDOW_H
```

mainwindow.cpp

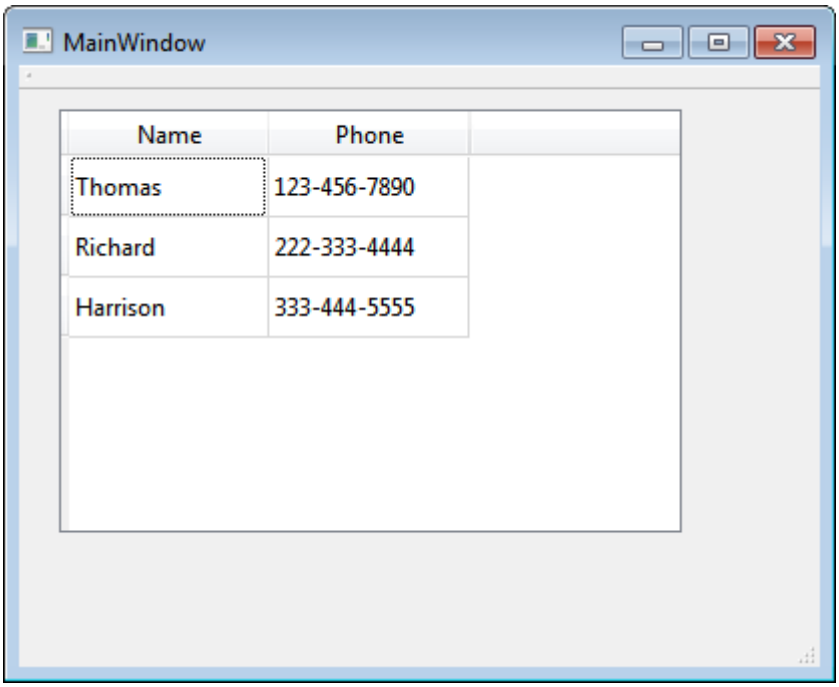
```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    QList<QString> contactNames;
    QList<QString> contactPhoneNums;

    // Create some data that is tabular in nature:
    contactNames.append("Thomas");
    contactNames.append("Richard");
    contactNames.append("Harrison");
    contactPhoneNums.append("123-456-7890");
    contactPhoneNums.append("222-333-4444");
    contactPhoneNums.append("333-444-5555");

    // Create model:
    TestModel *PhoneBookModel = new TestModel(this);

    // Populate model with data:
    PhoneBookModel->populateData(contactNames,contactPhoneNums);

    // Connect model to table view:
    ui->tableView->setModel(PhoneBookModel);

    // Make table header visible and display table:
    ui->tableView->horizontalHeader()->setVisible(true);
    ui->tableView->show();
}

MainWindow::~MainWindow()
{
    delete ui;
}

TestModel::TestModel(QObject *parent) : QAbstractTableModel(parent)
{
}

// Create a method to populate the model with data:
void TestModel::populateData(const QList<QString> &contactName,const QList<QString>
&contactPhone)
{
```

```
    tm_contact_name.clear();
    tm_contact_name = contactName;
    tm_contact_phone.clear();
    tm_contact_phone = contactPhone;
    return;
}

int TestModel::rowCount(const QModelIndex &parent) const
{
    Q_UNUSED(parent);
    return tm_contact_name.length();
}

int TestModel::columnCount(const QModelIndex &parent) const
{
    Q_UNUSED(parent);
    return 2;
}

QVariant TestModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid() || role != Qt::DisplayRole) {
        return QVariant();
    }
    if (index.column() == 0) {
        return tm_contact_name[index.row()];
    } else if (index.column() == 1) {
        return tm_contact_phone[index.row()];
    }
    return QVariant();
}

QVariant TestModel::headerData(int section, Qt::Orientation orientation, int role) const
{
    if (role == Qt::DisplayRole && orientation == Qt::Horizontal) {
        if (section == 0) {
            return QString("Name");
        } else if (section == 1) {
            return QString("Phone");
        }
    }
    return QVariant();
}
```

A l'aide de `Qt Creator/Design` , placez un objet `Table View` , nommé **tableView** dans cet exemple, dans la **fenêtre principale** :



Le programme résultant affiche comme:



Un modèle d'arbre simple

QModelIndex ne connaît pas réellement ses index parent / enfant, il ne contient qu'une **ligne** , une **colonne** et un **pointeur** , et il incombe aux modèles d'utiliser ces données pour fournir des informations aux relations d'un index. Le modèle doit donc effectuer de nombreuses conversions du `void*` stocké dans `QModelIndex` vers un type de données interne et `QModelIndex` .

TreeModel.h:

```
#pragma once

#include <QAbstractItemModel>

class TreeModel : public QAbstractItemModel
```

```
{
    Q_OBJECT
public:
    explicit TreeModel(QObject *parent = nullptr);

    // Reimplementation of QAbstractItemModel methods
    int rowCount(const QModelIndex &index) const override;
    int columnCount(const QModelIndex &index) const override;
    QModelIndex index(const int row, const int column,
        const QModelIndex &parent) const override;
    QModelIndex parent(const QModelIndex &childIndex) const override;
    QVariant data(const QModelIndex &index, const int role) const override;
    bool setData(const QModelIndex &index, const QVariant &value,
        const int role) override;
    Qt::ItemFlags flags(const QModelIndex &index) const override;

    void addRow(const QModelIndex &parent, const QVector<QVariant> &values);
    void removeRow(const QModelIndex &index);

private:
    struct Item
    {
        ~Item();

        // This could individual members, or maybe some other object that
        // contains the data we want to display/edit
        QVector<QVariant> values;

        // It is this information that the model needs to be able to answer
        // questions like "What's the parent QModelIndex of this QModelIndex?"
        QVector<Item *> children;
        Item *parent = nullptr;

        // Convenience method that's used in several places
        int rowInParent() const;
    };
    Item *m_root;
};
```

TreeModel.cpp:

```
#include "TreeModel.h"

// Adapt this to own needs
static constexpr int COLUMNS = 3;

TreeModel::Item::~Item()
{
    qDeleteAll(children);
}
int TreeModel::Item::rowInParent() const
{
    if (parent) {
        return parent->children.indexOf(const_cast<Item *>(this));
    } else {
        return 0;
    }
}

TreeModel::TreeModel(QObject *parent)
```

```
    : QAbstractItemModel(parent), m_root(new Item) {}

int TreeModel::rowCount(const QModelIndex &parent) const
{
    // Parent being invalid means we ask for how many rows the root of the
    // model has, thus we ask the root item
    // If parent is valid we access the Item from the pointer stored
    // inside the QModelIndex
    return parent.isValid()
        ? static_cast<Item *>(parent.internalPointer())->children.size()
        : m_root->children.size();
}
int TreeModel::columnCount(const QModelIndex &parent) const
{
    return COLUMNS;
}

QModelIndex TreeModel::index(const int row, const int column,
    const QModelIndex &parent) const
{
    // hasIndex checks if the values are in the valid ranges by using
    // rowCount and columnCount
    if (!hasIndex(row, column, parent)) {
        return QModelIndex();
    }

    // In order to create an index we first need to get a pointer to the Item
    // To get started we have either the parent index, which contains a pointer
    // to the parent item, or simply the root item

    Item *parentItem = parent.isValid()
        ? static_cast<Item *>(parent.internalPointer())
        : m_root;

    // We can now simply look up the item we want given the parent and the row
    Item *childItem = parentItem->children.at(row);

    // There is no public constructor in QModelIndex we can use, instead we need
    // to use createIndex, which does a little bit more, like setting the
    // model() in the QModelIndex to the model that calls createIndex
    return createIndex(row, column, childItem);
}
QModelIndex TreeModel::parent(const QModelIndex &childIndex) const
{
    if (!childIndex.isValid()) {
        return QModelIndex();
    }

    // Simply get the parent pointer and create an index for it
    Item *parentItem = static_cast<Item*>(childIndex.internalPointer())->parent;
    return parentItem == m_root
        ? QModelIndex() // the root doesn't have a parent
        : createIndex(parentItem->rowInParent(), 0, parentItem);
}

QVariant TreeModel::data(const QModelIndex &index, const int role) const
{
    // Usually there will be more stuff here, like type conversion from
    // QVariant, handling more roles etc.
    if (!index.isValid() || role != Qt::DisplayRole) {
        return QVariant();
```

```
    }
    Item *item = static_cast<Item *>(index.internalPointer());
    return item->values.at(index.column());
}
bool TreeModel::setData(const QModelIndex &index, const QVariant &value,
    const int role)
{
    // As in data there will usually be more stuff here, like type conversion to
    // QVariant, checking values for validity etc.
    if (!index.isValid() || role != Qt::EditRole) {
        return false;
    }
    Item *item = static_cast<Item *>(index.internalPointer());
    item->values[index.column()] = value;
    emit dataChanged(index, index, QVector<int>() << role);
    return true;
}
Qt::ItemFlags TreeModel::flags(const QModelIndex &index) const
{
    if (index.isValid()) {
        return Qt::ItemIsEnabled | Qt::ItemIsSelectable | Qt::ItemIsEditable;
    } else {
        return Qt::NoItemFlags;
    }
}

// Simple add/remove functions to illustrate {begin,end}{Insert,Remove}Rows
// usage in a tree model
void TreeModel::addRow(const QModelIndex &parent,
    const QVector<QVariant> &values)
{
    Item *parentItem = parent.isValid()
        ? static_cast<Item *>(parent.internalPointer())
        : m_root;
    beginInsertRows(parent,
        parentItem->children.size(), parentItem->children.size());
    Item *item = new Item;
    item->values = values;
    item->parent = parentItem;
    parentItem->children.append(item);
    endInsertRows();
}
void TreeModel::removeRow(const QModelIndex &index)
{
    if (!index.isValid()) {
        return;
    }
    Item *item = static_cast<Item *>(index.internalPointer());
    Q_ASSERT(item != m_root);
    beginRemoveRows(index.parent(), item->rowInParent(), item->rowInParent());
    item->parent->children.removeOne(item);
    delete item;
    endRemoveRows();
}
```

Lire Modèle / vue en ligne: https://riptutorial.com/fr/qt/topic/3938/modele---vue

Chapitre 9: Multimédia

Remarques

Qt Multimedia est un module permettant de gérer le multimédia (audio, vidéo) ainsi que les fonctionnalités de caméra et de radio.

Cependant, les fichiers pris en charge de QMediaPlayer dépendent de la plate-forme. En effet, sur Windows, QMediaPlayer utilise DirectShow, sous Linux, il utilise GStreamer. Ainsi, en fonction de la plate-forme, certains fichiers peuvent fonctionner sous Linux mais pas sous Windows ou le contraire.

Examples

Lecture vidéo dans Qt 5

Créons un lecteur vidéo très simple avec le module QtMultimedia de Qt 5.

Dans le fichier .pro de votre application, vous aurez besoin des lignes suivantes:

```
QT += multimedia multimediawidgets
```

Notez que `multimediawidgets` est nécessaire pour utiliser `QVideoWidget` .

```
#include <QtMultimedia/QMediaPlayer>
#include <QtMultimedia/QMediaPlaylist>
#include <QtMultimediaWidgets/QVideoWidget>

QMediaPlayer *player;
QVideoWidget *videoWidget;
QMediaPlaylist *playlist;

player = new QMediaPlayer;

playlist = new QMediaPlaylist(player);
playlist->addMedia(QUrl::fromLocalFile("actualPathHere"));

videoWidget = new QVideoWidget;
player->setVideoOutput(videoWidget);

videoWidget->show();
player->play();
```

C'est tout - après le lancement de l'application (si les codecs nécessaires sont installés dans le système), la lecture du fichier vidéo sera lancée.

De la même manière, vous pouvez lire des vidéos à partir d'URL sur Internet, et pas uniquement des fichiers locaux.

Lecture audio dans Qt5

Comme il s'agit d'un audio, nous n'avons pas besoin d'un QVideoWidget. Nous pouvons donc faire:

```
_player = new QMediaPlayer(this);
QUrl file = QUrl::fromLocalFile(QFileDialog::getOpenFileName(this, tr("Open Music"), "",
tr("")));
if (file.url() == "")
    return ;
_player->setMedia(file);
_player->setVolume(50);
_player->play();
```

dans le .h:

```
QMediaPlayer *_player;
```

Cela ouvrira une boîte de dialogue où vous pourrez choisir votre musique et la jouer.

Lire Multimédia en ligne: https://riptutorial.com/fr/qt/topic/7675/multimedia

Chapitre 10: Partage implicite

Remarques

Les itérateurs de style STL sur Qt Container peuvent avoir des effets secondaires négatifs dus au partage implicite. Il est conseillé d'éviter de copier un conteneur Qt alors que des itérateurs sont actifs dessus.

```
QVector<int> a,b; //2 vectors
a.resize(1000);
b = a; // b and a now point to the same memory internally

auto iter = a.begin(); //iter also points to the same memory a and b do
a[4] = 1; //a creates a new copy and points to different memory.
//Warning 1: b and iter point sill to the same even if iter was "a.begin()"

b.clear(); //delete b-memory
//Warning 2: iter only holds a pointer to the memory but does not increase ref-count.
//           so now the memory iter points to is invalid. UB!
```

Examples

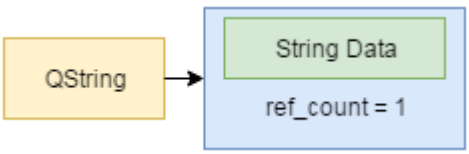
Concept de base

Plusieurs objets et conteneurs Qt utilisent un concept appelé **partage implicite** , qui peut également être appelé **copie sur écriture** .

Le partage implicite signifie que les classes qui utilisent ce concept partagent les mêmes données lors de l'initialisation.

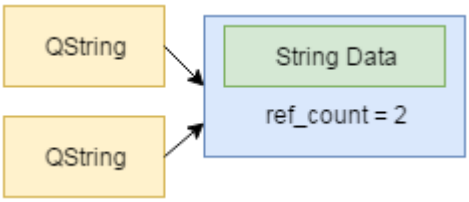
L'une de ces classes pour utiliser le concept est QString.

```
QString s1("Hello World");
```



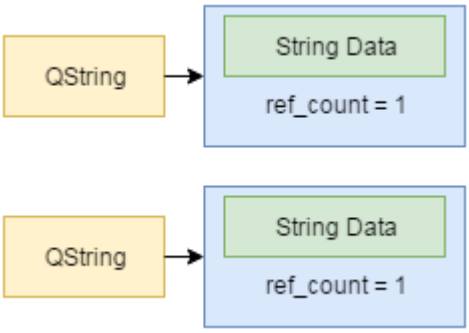
Ceci est un modèle simplifié d'un QString. En interne, il possède un bloc de mémoire, avec les données de chaîne réelles et un compteur de référence.

```
QString s2 = s1;
```



Si nous copions maintenant cette `QString` deux objets pointeront en interne sur le même contenu, évitant ainsi des opérations de copie inutiles. Notez comment le compte de référence a également été augmenté. Donc, si la première chaîne est supprimée, les données partagées savent toujours qu'elle est référencée par une autre `QString` .

```
s2 += " and all the other Worlds!"
```



Maintenant, lorsque la `QString` est réellement modifiée, l'objet "se détache" du bloc mémoire, copiant son contenu et en modifiant le contenu.

Lire Partage implicite en ligne: https://riptutorial.com/fr/qt/topic/6801/partage-implicite

Chapitre 11: Pièges courants

Examples

Utilisation de Qt: DirectConnection lorsqu'un objet récepteur ne reçoit pas de signal

Parfois, vous voyez qu'un signal est émis dans le thread émetteur, mais le connecteur connecté ne l'appelle pas (en d'autres termes, il ne reçoit pas de signal), vous avez demandé à ce sujet le type de connexion Qt :: DirectConnection donc le problème trouvé et tout va bien.

Mais en règle générale, il est déconseillé d'utiliser Qt: DirectConnection jusqu'à ce que vous sachiez vraiment ce que c'est et qu'il n'ya pas d'autre moyen. Expliquons-le davantage. Chaque thread créé par Qt (y compris le thread principal et les nouveaux threads créés par QThread) possède une boucle d'événement, la boucle d'événement est chargée de recevoir les signaux et appelle des emplacements appropriés dans son thread. Généralement, l'exécution d'une opération de blocage dans un logement est une mauvaise pratique, car elle bloque la boucle d'événement de ces threads afin qu'aucun autre emplacement ne soit appelé.

Si vous bloquez une boucle d'événement (en prenant beaucoup de temps ou en bloquant l'opération), vous ne recevrez pas d'événements sur ce thread tant que la boucle d'événement ne sera pas débloquée. Si l'opération de blocage bloque la boucle d'événement pour toujours (comme un temps occupé), les emplacements ne pourront jamais être appelés.

Dans cette situation, vous pouvez définir le type de connexion dans la connexion à Qt :: DirectConnection, maintenant les emplacements seront appelés même la boucle d'événement est bloquée. alors comment cela pourrait faire tout casser? Dans Qt :: DirectConnection, les slots seront appelés dans les threads émetteurs, et non dans les threads récepteurs, et ils pourront briser les synchronisations de données et rencontrer d'autres problèmes. N'utilisez donc jamais Qt :: DirectConnection à moins de savoir ce que vous faites. Si votre problème est résolu à l'aide de Qt :: DirectConnection, vous devez examiner votre code et déterminer pourquoi votre boucle d'événements est bloquée. Ce n'est pas une bonne idée de bloquer la boucle d'événement et ce n'est pas recommandé dans Qt.

Voici un petit exemple qui montre le problème, car vous pouvez voir que le nonBlockingSlot serait appelé même la boucle d'événement bloqué blockingSlot avec while (1), ce qui indique un mauvais codage

```
class TestReceiver : public QObject{
    Q_OBJECT
public:
    TestReceiver(){
         qDebug() << "TestReceiver Constructed in" << QThread::currentThreadId();
    }
public slots:
    void blockingSlot()
    {
```

```
        static bool firstInstance = false;
        qDebug() << "Blocking slot called in thread" << QThread::currentThreadId();
        if(!firstInstance){
            firstInstance = true;
            while(1);
        }
    }
    void nonBlockingSlot(){
        qDebug() << "Non-blocking slot called" << QThread::currentThreadId();
    }
};

class TestSender : public QObject{
    Q_OBJECT
public:
    TestSender(TestReceiver * receiver){
        this->nonBlockingTimer.setInterval(100);
        this->blockingTimer.setInterval(100);

        connect(&this->blockingTimer, &QTimer::timeout, receiver,
&TestReceiver::blockingSlot);
        connect(&this->nonBlockingTimer, &QTimer::timeout, receiver,
&TestReceiver::nonBlockingSlot, Qt::DirectConnection);
        this->nonBlockingTimer.start();
        this->blockingTimer.start();
    }
private:
    QTimer nonBlockingTimer;
    QTimer blockingTimer;
};

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

    TestReceiver TestReceiverInstance;
    TestSender testSenderInstance(&TestReceiverInstance);
    QThread receiverThread;
    TestReceiverInstance.moveToThread(&receiverThread);
    receiverThread.start();

    return a.exec();
}
```

Lire Pièges courants en ligne: https://riptutorial.com/fr/qt/topic/8238/pieges-courants

Chapitre 12: QDialogs

Remarques

La classe QDialog est la classe de **base** des fenêtres de dialogue. Une fenêtre de dialogue est une fenêtre de niveau supérieur principalement utilisée pour les tâches à court terme et les communications brèves avec l'utilisateur. QDialogs peut être **modal** ou sans **modèle** .

Notez que QDialog (et tout autre widget ayant le type Qt :: Dialog) utilise le widget parent légèrement différemment des autres classes de Qt. Un dialogue est **toujours un widget de premier niveau** , mais s'il **a un parent, son emplacement par défaut est centré sur le widget de niveau supérieur du parent** (s'il ne l'est pas lui-même). Il partagera également l'entrée de la barre des tâches du parent.

Une boîte de dialogue **modale** est une boîte de dialogue qui bloque l'entrée dans d'autres fenêtres visibles dans la même application. Les dialogues utilisés pour demander un nom de fichier à l'utilisateur ou pour définir les préférences de l'application sont généralement modaux. Les dialogues peuvent être **modaux** (par défaut) ou **modaux** .

La manière la plus courante d'afficher une boîte de dialogue modale consiste à appeler sa fonction exec (). Lorsque l'utilisateur ferme la boîte de dialogue, exec () fournira une valeur de retour utile.

Un dialogue sans **modèle** est une boîte de dialogue qui fonctionne indépendamment des autres fenêtres de la même application. Les boîtes de dialogue non modélisées sont affichées à l'aide de show (), qui renvoie immédiatement le contrôle à l'appelant.

Examples

MyCompareFileDialog.h

```
#ifndef MYCOMPAREFILEDIALOG_H
#define MYCOMPAREFILEDIALOG_H

#include <QtWidgets/QDialog>

class MyCompareFileDialog : public QDialog
{
    Q_OBJECT

public:
    MyCompareFileDialog(QWidget *parent = 0);
    ~MyCompareFileDialog();
};


#endif // MYCOMPAREFILEDIALOG_H
```

MyCompareFileDialogDialog.cpp

```
#include "MyCompareFileDialog.h"
#include <QLabel>

MyCompareFileDialog::MyCompareFileDialog(QWidget *parent)
: QDialog(parent)
{
    setWindowTitle("Compare Files");
    setWindowFlags(Qt::Dialog);
    setWindowModality(Qt::WindowModal);

    resize(300, 100);
    QSizePolicy sizePolicy(QSizePolicy::Preferred, QSizePolicy::Preferred);
    setSizePolicy(sizePolicy);
    setMinimumSize(QSize(300, 100));
    setMaximumSize(QSize(300, 100));

    QLabel* myLabel = new QLabel(this);
    myLabel->setText("My Dialog!");
}

MyCompareFileDialog::~MyCompareFileDialog()
{ }
```

MainWindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
class MainWindow;
}

class MyCompareFileDialog;

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;
    MyCompareFileDialog* myDialog;
};

#endif // MAINWINDOW_H
```

MainWindow.cpp

```
#include "mainwindow.h"
```

```
#include "ui_mainwindow.h"
#include "mycomparefiledialog.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    myDialog = new MyCompareFileDialog(this);

    connect(ui->pushButton,SIGNAL(clicked()),myDialog,SLOT(exec()));
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

main.cpp

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

mainwindow.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>400</width>
    <height>300</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>MainWindow</string>
  </property>
  <widget class="QWidget" name="centralWidget">
   <widget class="QPushButton" name="pushButton">
    <property name="geometry">
     <rect>
      <x>140</x>
      <y>80</y>
      <width>111</width>
```

```
      <height>23</height>
     </rect>
    </property>
    <property name="text">
     <string>Show My Dialog</string>
    </property>
   </widget>
  </widget>
  <widget class="QMenuBar" name="menuBar">
   <property name="geometry">
    <rect>
     <x>0</x>
     <y>0</y>
     <width>400</width>
     <height>21</height>
    </rect>
   </property>
  </widget>
  <widget class="QToolBar" name="mainToolBar">
   <attribute name="toolBarArea">
    <enum>TopToolBarArea</enum>
   </attribute>
   <attribute name="toolBarBreak">
    <bool>false</bool>
   </attribute>
  </widget>
  <widget class="QStatusBar" name="statusBar"/>
 </widget>
 <layoutdefault spacing="6" margin="11"/>
 <resources/>
 <connections/>
</ui>
```

Lire QDialogs en ligne: https://riptutorial.com/fr/qt/topic/7819/qdialogs

Chapitre 13: QGraphics

Examples

Panoramique, zoom et rotation avec QGraphicsView

`QGraphics` peut être utilisé pour organiser des scènes compliquées d'objets visuels dans un cadre plus facile à manipuler.

Il existe trois principaux types d'objets utilisés dans ce cadre: QGraphicsView , QGraphicsScene et QGraphicsItems . QGraphicsItems sont les éléments visuels de base qui existent dans la scène.

De nombreux types sont préconfigurés et peuvent être utilisés, tels que les ellipses , les lignes , les chemins , les pixels , les polygones , les rectangles et le texte .

Vous pouvez également créer vos propres éléments en héritant de `QGraphicsItem` . Ces éléments sont ensuite placés dans un `QGraphicsScene` qui est fondamentalement le monde que vous envisagez de regarder. Les objets peuvent se déplacer dans la scène, ce qui revient à les faire bouger dans le monde que vous regardez. Le positionnement et l'orientation des éléments sont gérés par des matrices de transformation appelées QTransforms . Qt a de jolies fonctions intégrées, vous n'avez donc pas besoin de travailler directement avec `QTransforms` , mais vous appelez des fonctions telles que pivoter ou redimensionner pour créer les transformations appropriées. La scène est ensuite visualisée selon la perspective définie dans `QGraphicsView` (à nouveau avec `QTransforms`), qui est l'élément que vous `QTransforms` dans un widget de votre interface utilisateur.

Dans l'exemple suivant, il y a une scène très simple avec un seul élément (une pixmap), qui est placé dans une scène et affiché dans une vue. En `DragMode` indicateur `DragMode` , la scène peut être déplacée avec la souris et en utilisant les fonctions de mise à l'échelle et de rotation, elle peut être mise à l'échelle avec le défilement de la souris et pivotée avec les touches fléchées.

Si vous souhaitez exécuter cet exemple, créez une instance de View qui sera affichée et créez un fichier de ressources avec le préfixe / images contenant une image my_image.png.

```
#include <QGraphicsView>
#include <QGraphicsScene>
#include <QGraphicsPixmapItem>
#include <QWheelEvent>
#include <QKeyEvent>

class View : public QGraphicsView
{
  Q_OBJECT
public:
  explicit View(QWidget *parent = 0) :
    QGraphicsView(parent)
  {
    setDragMode(QGraphicsView::ScrollHandDrag);
```

```
    QGraphicsPixmapItem *pixmapItem = new
QGraphicsPixmapItem(QPixmap(":/images/my_image.png"));
    pixmapItem->setTransformationMode(Qt::SmoothTransformation);

    QGraphicsScene *scene = new QGraphicsScene();
    scene->addItem(pixmapItem);
    setScene(scene);
  }

protected Q_SLOTS:
  void wheelEvent(QWheelEvent *event)
  {
    if(event->delta() > 0)
      scale(1.25, 1.25);
    else
      scale(0.8, 0.8);
  }

  void keyPressEvent(QKeyEvent *event)
  {
    if(event->key() == Qt::Key_Left)
      rotate(1);
    else if(event->key() == Qt::Key_Right)
      rotate(-1);
  }
};
```

Lire QGraphics en ligne: https://riptutorial.com/fr/qt/topic/7539/qgraphics

Chapitre 14: qmake

Examples

Profil par défaut.

qmake est un outil d'automatisation de la construction, fourni avec le *framework Qt* . Il fait un travail similaire aux outils tels que *CMake* ou *GNU Autotools* , mais il est conçu pour être utilisé spécifiquement avec *Qt* . En tant que tel, il est bien intégré à l'écosystème *Qt* , notamment *QT Creator* IDE.

Si vous démarrez *Qt Creator* et sélectionnez `File -> New File or Project -> Application -> Qt Widgets` , *Qt Creator* générera un squelette de projet avec un fichier "pro". Le fichier "pro" est traité par *qmake* afin de générer des fichiers, qui sont à leur tour traités par les systèmes de construction sous-jacents (par exemple *GNU Make* ou *nmake*).

Si vous avez nommé votre projet "myapp", le fichier "myapp.pro" apparaîtra. Voici comment ce fichier par défaut ressemble, avec des commentaires, qui décrivent chaque variable *qmake* ajoutée.

```
# Tells build system that project uses Qt Core and Qt GUI modules.
QT       += core gui

# Prior to Qt 5 widgets were part of Qt GUI module. In Qt 5 we need to add Qt Widgets module.
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

# Specifies name of the binary.
TARGET = myapp

# Denotes that project is an application.
TEMPLATE = app

# List of source files (note: Qt Creator will take care about this list, you don't need to
update is manually).
SOURCES += main.cpp\
        mainwindow.cpp

# List of header files (note: Qt Creator will take care about this list).
HEADERS  += mainwindow.h

# List of "ui" files for a tool called Qt Designer, which is embedded into Qt Creator in newer
versions of IDE (note: Qt Creator will take care about this list).
FORMS    += mainwindow.ui
```

Préserver la structure du répertoire source dans une construction (option "object_parallel_to_source" non documentée).

Si vous souhaitez organiser votre projet en conservant les fichiers source dans différents sous-répertoires, vous devez savoir que lors d'une génération, *qmake* ne conservera pas cette structure de répertoires et conservera tous les fichiers ".o" dans un seul répertoire de construction. Cela

peut être un problème si vous aviez des noms de fichiers contradictoires dans différents répertoires, comme la suite.

```
src/file1.cpp
src/plugin/file1.cpp
```

Maintenant, *qmake* décidera de créer deux fichiers "fichier1.o" dans un répertoire de compilation, provoquant le remplacement de l'un d'eux par un autre. Le buld échouera. Pour éviter cela, vous pouvez ajouter l'option de configuration `CONFIG += object_parallel_to_source` à votre fichier "pro". Cela indiquera à *qmake* de générer des fichiers de construction qui préservent votre structure de répertoire source. De cette façon, votre répertoire de construction reflétera la structure du répertoire source et les fichiers objets seront créés dans des sous-répertoires distincts.

```
src/file1.o
src/plugin/file1.o
```

Exemple complet

```
QT += core
TARGET = myapp
TEMPLATE = app

CONFIG += object_parallel_to_source

SOURCES += src/file1.cpp \
           src/plugin/file1.cpp

HEADERS  += src/plugin/file1.h
```

Notez que l'option `CONFIG object_parallel_to_source` n'est **pas officiellement documentée** .

Exemple simple (Linux)

Window.h

```
#include <QWidget>

class Window : public QWidget
{
    Q_OBJECT
public:
    Window(QWidget *parent = Q_NULLPTR) : QWidget(parent) {}
}
```

main.cpp

```
#include <QApplication>
#include "Window.h"

int main()
{
    QApplication app;
```

```
    Window window;
    window.show();
    return app.exec();
}
```

example.pro

```
# The QT variable controls what modules are included in compilation.
# Note that the 'core' and 'gui' modules are included by default.
# For widget-based GUI applications, the 'widgets' module needs to be added.
QT += widgets

HEADERS = Window.h # Everything added to the HEADER variable will be checked
                   # to see if moc needs to run on it, and it will be run if
                   # so.

SOURCES = main.cpp # Everything added to the SOURCES variable will be compiled
                   # and (in the simple example) added to the resulting
                   # executable.
```

Ligne de commande

```
# Assuming you are in a folder that contains the above files.
> qmake         # You can also add the example.pro file if needed
> make          # qmake creates a Makefile, this runs make on it.
> ./example     # The name of the executable defaults to the .pro file name.
```

Exemple SUBDIRS

La capacité SUBDIRS de qmake peut être utilisée pour compiler un ensemble de bibliothèques dont chacune dépend d'une autre. L'exemple ci-dessous est légèrement compliqué pour montrer les variations avec la capacité SUBDIRS.

Structure du répertoire

Certains des fichiers suivants seront omis dans un souci de concision. Ils peuvent être supposés être le format en tant qu'exemples non-subdir.

```
project_dir/
-project.pro
-common.pri
-build.pro
-main.cpp
-logic/
----logic.pro
----some logic files
-gui/
----gui.pro
----gui files
```

project.pro

C'est le fichier principal qui active l'exemple. C'est aussi le fichier qui serait appelé avec qmake

sur la ligne de commande (voir ci-dessous).

```
TEMPLATE = subdirs  # This changes to the subdirs function.  You can't combine
                    # compiling code and the subdirs function in the same .pro
                    # file.

# By default, you assign a directory to the SUBDIRS variable, and qmake looks
# inside that directory for a <dirname>.pro file.
SUBDIRS = logic

# You can append as many items as desired.  You can also specify the .pro file
# directly if need be.
SUBDIRS += gui/gui.pro

# You can also create a target that isn't a subdirectory, or that refers to a
# different file(*).
SUBDIRS += build
build.file = build.pro # This specifies the .pro file to use
# You can also use this to specify dependencies.  In this case, we don't want
# the build target to run until after the logic and gui targets are complete.
build.depends = logic gui/gui.pro
```

(*) Voir la documentation de référence pour les autres options pour une cible de sous-répertoires.

common.pri

```
#Includes common configuration for all subdirectory .pro files.
INCLUDEPATH += . ..
WARNINGS += -Wall

TEMPLATE = lib

# The following keeps the generated files at least somewhat separate
# from the source files.
UI_DIR = uics
MOC_DIR = mocs
OBJECTS_DIR = objs
```

logique / logic.pro

```
# Check if the config file exists
! include( ../common.pri ) {
    error( "Couldn't find the common.pri file!" )
}

HEADERS += logic.h
SOURCES += logic.cpp

# By default, TARGET is the same as the directory, so it will make
# liblogic.so (in linux).  Uncomment to override.
# TARGET = target
```

gui / gui.pro

```
! include( ../common.pri ) {
    error( "Couldn't find the common.pri file!" )
```

```
}

FORMS += gui.ui
HEADERS += gui.h
SOURCES += gui.cpp

# By default, TARGET is the same as the directory, so it will make
# libgui.so (in linux).  Uncomment to override.
# TARGET = target
```

build.pro

```
TEMPLATE = app

SOURCES += main.cpp

LIBS += -Llogic -Lgui -llogic -lgui

# This renames the resulting executable
TARGET = project
```

Ligne de commande

```
# Assumes you are in the project_dir directory
> qmake project.pro # specific the .pro file since there are multiple here.
> make -n2 # This makes logic and gui concurrently, then the build Makefile.
> ./project # Run the resulting executable.
```

Exemple de bibliothèque

Un exemple simple pour créer une bibliothèque (plutôt qu'un exécutable, qui est la valeur par défaut). Variable `TEMPLATE` spécifie le type du projet que vous effectuez. `lib` option `lib` permet à makefile de construire une bibliothèque.

library.pro

```
HEADERS += library.h
SOURCES += library.cpp

TEMPLATE = lib

# By default, qmake will make a shared library.  Uncomment to make the library
# static.
# CONFIG += staticlib

# By default, TARGET is the same as the directory, so it will make
# liblibrary.so or liblibrary.a (in linux).  Uncomment to override.
# TARGET = target
```

Lorsque vous `staticlib` une bibliothèque, vous pouvez ajouter des options `dll` (par défaut), `staticlib` ou `plugin` - `plugin` à `CONFIG` .

Création d'un fichier de projet à partir du code existant

Si vous avez un répertoire avec les fichiers source existants, vous pouvez utiliser `qmake` avec le `-project` -option pour créer un fichier de projet.

Supposons que le dossier *MyProgram* contienne les fichiers suivants:

- main.cpp
- foo.h
- foo.cpp
- bar.h
- bar.cpp
- sous-répertoire / foobar.h
- sous-répertoire / foobar.cpp

Puis en appelant

```
qmake -project
```

un fichier *MyProgram.pro* est créé avec le contenu suivant:

```
#############################################################################
# Automatically generated by qmake (3.0) Mi. Sep. 7 23:36:56 2016
#############################################################################

TEMPLATE = app
TARGET = MyProgram
INCLUDEPATH += .

# Input
HEADERS += bar.h foo.h subdir/foobar.h
SOURCES += bar.cpp foo.cpp main.cpp subdir/foobar.cpp
```

Le code peut alors être construit comme décrit dans cet exemple simple .

Lire qmake en ligne: https://riptutorial.com/fr/qt/topic/4438/qmake

Chapitre 15: QObject

Remarques

`QObject` classe `QObject` est la classe de base pour tous les objets Qt.

Examples

QObjet exemple

`Q_OBJECT` macro `Q_OBJECT` apparaît dans la section privée d'une classe. `Q_OBJECT` exige que la classe soit une sous-classe de `QObject` . Cette macro est nécessaire pour que la classe déclare ses signaux / slots et utilise le système de méta-objets Qt.

Si Meta Object Compiler (MOC) trouve la classe avec `Q_OBJECT` , il le traite et génère un fichier source C ++ contenant le code source du méta-objet.

Voici l'exemple de l'en-tête de classe avec `Q_OBJECT` et signal / slots:

```
#include <QObject>

class MyClass : public QObject
{
    Q_OBJECT

public:

public slots:
    void setNumber(double number);

signals:
    void numberChanged(double number);

private:
}
```

qobject_cast

```
T qobject_cast(QObject *object)
```

Une fonctionnalité ajoutée en dérivant de `QObject` et en utilisant la macro `Q_OBJECT` est la possibilité d'utiliser le `qobject_cast` .

Exemple:

```
class myObject : public QObject
{
    Q_OBJECT
    //...
```

```
};

QObject* obj = new myObject();
```

Pour vérifier si `obj` est un type `myObject` et le `myObject` en C ++, vous pouvez généralement utiliser un `dynamic_cast` . Cela dépend de l'activation de RTTI lors de la compilation.

La macro Q_OBJECT des autres mains génère les contrôles de conversion et le code qui peuvent être utilisés dans le qobject_cast.

```
myObject* my = qobject_cast<myObject*>(obj);
if(!myObject)
{
    //wrong type
}
```

Cela ne dépend pas de RTTI. Et vous permet également de diffuser des limites de bibliothèque dynamiques (via les interfaces / plugins Qt).

Durée de vie et propriété de QObject

Les QObjects sont livrés avec leur propre concept de durée de vie par rapport aux pointeurs bruts, uniques ou partagés de C ++.

Les QObjects ont la possibilité de créer un objecttree en déclarant des relations parent / enfant.

Le moyen le plus simple de déclarer cette relation est de transmettre l'objet parent dans le constructeur. Comme alternative, vous pouvez définir manuellement le parent d'un `QObject` en appelant `setParent` . C'est la seule direction pour déclarer cette relation. Vous ne pouvez pas ajouter un enfant à une classe de parents mais seulement l'inverse.

```
QObject parent;
QObject child* = new QObject(&parent);
```

Lorsque `parent` maintenant supprimé dans stack- `child` , l' `child` sera également supprimé.

Lorsque nous `QObject` un objet `QObject` , " `QObject` " lui-même forme l'objet parent;

```
QObject parent;
QObject child* = new QObject(&parent);
delete child; //this causes no problem.
```

La même chose s'applique aux variables de pile:

```
QObject parent;
QObject child(&parent);
```

`child` sera supprimé avant `parent` pendant le déroulement de la pile et se désinscrire de son parent.

Remarque: vous pouvez appeler manuellement `setParent` avec un ordre inverse de la déclaration qui cassera la destruction automatique.

Lire QObject en ligne: https://riptutorial.com/fr/qt/topic/6304/qobject

Chapitre 16: Qt - Traitement des bases de données

Remarques

- Vous aurez besoin du plug-in Qt SQL correspondant au type donné à `QSqlDatabase::addDatabase`
- Si vous ne disposez pas du plugin SQL requis, Qt vous avertira qu'il ne trouve pas le pilote demandé
- Si vous ne disposez pas du plug-in SQL requis, vous devrez les compiler à partir de la source Qt

Examples

Utiliser une base de données sur Qt

Dans le fichier Project.pro, nous ajoutons:

```
CONFIG += sql
```

dans MainWindow.h nous écrivons:

```
#include <QMainWindow>
#include <QSql>
#include <QDebug>

namespace Ui
{
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:

private:
    Ui::MainWindow *ui;
    QSqlDatabase db;
};
```

Maintenant dans MainWindow.cpp:

```
#include "mainwindow.h"
```

```
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    db = QSqlDatabase::addDatabase("QT SQL DRIVER" , "CONNECTION NAME");
    db.setDatabaseName("DATABASE NAME");
    if(!db.open())
    {
        qDebug() << "Can't Connect to DB !";
    }
    else
    {
        qDebug() << "Connected Successfully to DB !";
        QSqlQuery query;
        query.prepare("QUERY TO BE SENT TO THE DB");
        if(!query.exec())
        {
            qDebug() << "Can't Execute Query !";
        }
        else
        {
            qDebug() << "Query Executed Successfully !";
        }
    }
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

Qt - Traitement des bases de données SQLite

Dans le fichier Project.pro, nous ajoutons: `CONFIG += sql`

dans MainWindow.h nous écrivons:

```
#include <QMainWindow>
#include <QSql>
#include <QDebug>

namespace Ui
{
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
```

```
private:
    Ui::MainWindow *ui;
    QSqlDatabase db;
};
```

Maintenant dans MainWindow.cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    db = QSqlDatabase::addDatabase("QSQLITE" , "CONNECTION NAME");
    db.setDatabaseName("C:\\sqlite_db_file.sqlite");
    if(!db.open())
    {
        qDebug() << "Can't Connect to DB !";
    }
    else
    {
        qDebug() << "Connected Successfully to DB !";
        QSqlQuery query;
        query.prepare("SELECT name , phone , address FROM employees WHERE ID = 201");
        if(!query.exec())
        {
            qDebug() << "Can't Execute Query !";
        }
        else
        {
            qDebug() << "Query Executed Successfully !";
            while(query.next())
            {
                qDebug() << "Employee Name : " << query.value(0).toString();
                qDebug() << "Employee Phone Number : " << query.value(1).toString();
                qDebug() << "Employee Address : " << query.value(1).toString();
            }
        }
    }
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

Qt - Traitement des bases de données ODBC

Dans le fichier Project.pro, nous ajoutons: `CONFIG += sql`

dans MainWindow.h nous écrivons:

```
#include <QMainWindow>
```

```
#include <QSql>
#include <QDebug>

namespace Ui
{
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:

private:
    Ui::MainWindow *ui;
    QSqlDatabase db;
};
```

Maintenant dans MainWindow.cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    db = QSqlDatabase::addDatabase("QODBC" , "CONNECTION NAME");
    db.setDatabaseName("DRIVER={SQL Server};SERVER=localhost;DATABASE=WorkDatabase"); //
"WorkDatabase" is the name of the database we want
    db.setUserName("sa"); // Set Login Username
    db.setPassword(""); // Set Password if required
    if(!db.open())
    {
        qDebug() << "Can't Connect to DB !";
    }
    else
    {
        qDebug() << "Connected Successfully to DB !";
        QSqlQuery query;
        query.prepare("SELECT name , phone , address FROM employees WHERE ID = 201");
        if(!query.exec())
        {
            qDebug() << "Can't Execute Query !";
        }
        else
        {
            qDebug() << "Query Executed Successfully !";
            while(query.next())
            {
                qDebug() << "Employee Name : " << query.value(0).toString();
                qDebug() << "Employee Phone Number : " << query.value(1).toString();
                qDebug() << "Employee Address : " << query.value(1).toString();
```

```
            }
        }
    }
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

Qt - Traitement des bases de données SQL en mémoire

Dans le fichier Project.pro, nous ajoutons: `CONFIG += sql`

dans MainWindow.h nous écrivons:

```
#include <QMainWindow>
#include <QSql>
#include <QDebug>

namespace Ui
{
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:

private:
    Ui::MainWindow *ui;
    QSqlDatabase db;
};
```

Maintenant dans MainWindow.cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    db = QSqlDatabase::addDatabase("QSQLITE" , "CONNECTION NAME");
    db.setDatabaseName(":memory:");
    if(!db.open())
    {
        qDebug() << "Can't create in-memory Database!";
    }
```

```
    else
    {
        qDebug() << "In-memory Successfully created!";
        QSqlQuery query;

        if (!query.exec("CREATE TABLE employees (ID INTEGER, name TEXT, phone TEXT, address
TEXT)"))
        {
            qDebug() << "Can't create table!";
            return;
        }
        if (!query.exec("INSERT INTO employees (ID, name, phone, address) VALUES (201, 'Bob',
'5555-5555', 'Antarctica')"))
        {
            qDebug() << "Can't insert record!";
            return;
        }

        qDebug() << "Database filling completed!";
        if(!query.exec("SELECT name , phone , address FROM employees WHERE ID = 201"))
        {
            qDebug() << "Can't Execute Query !";
            return;
        }
        qDebug() << "Query Executed Successfully !";
        while(query.next())
        {
            qDebug() << "Employee Name : " << query.value(0).toString();
            qDebug() << "Employee Phone Number : " << query.value(1).toString();
            qDebug() << "Employee Address : " << query.value(1).toString();
        }
    }
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

Supprimer la connexion à la base de données correctement

Si nous voulons supprimer une connexion de base de données de la liste des connexions de base de données. nous devons utiliser `QSqlDatabase::removeDatabase()` , mais c'est une fonction statique et le fonctionnement est un peu câblé.

```
// WRONG WAY
  QSqlDatabase db = QSqlDatabase::database("sales");
  QSqlQuery query("SELECT NAME, DOB FROM EMPLOYEES", db);
  QSqlDatabase::removeDatabase("sales"); // will output a warning

  // "db" is now a dangling invalid database connection,
  // "query" contains an invalid result set
```

La manière correcte que Qt Document nous suggère est ci-dessous.

```
  {
      QSqlDatabase db = QSqlDatabase::database("sales");
```

```
    QSqlQuery query("SELECT NAME, DOB FROM EMPLOYEES", db);
}
// Both "db" and "query" are destroyed because they are out of scope
QSqlDatabase::removeDatabase("sales"); // correct
```

Lire Qt - Traitement des bases de données en ligne: https://riptutorial.com/fr/qt/topic/1993/qt---traitement-des-bases-de-donnees

Chapitre 17: Qt Container Classes

Remarques

Qt fournit ses propres classes de conteneur de modèles. Ils sont tous implicitement partagés. Ils fournissent deux types d'itérateurs (style Java et style STL).

Les conteneurs séquentiels Qt incluent: QVector, QList, QLinkedList, QStack, QQueue.

Les conteneurs associatifs Qt incluent: QMap, QMultiMap, QHash, QMultiHash, QSet.

Examples

Utilisation de la pile

`QStack<T>` est une pile fournissant une classe de modèle Qt. Son analogue en STL est `std::stack` . C'est structure dernier entré, premier sorti (LIFO).

```
QStack<QString> stack;
stack.push("First");
stack.push("Second");
stack.push("Third");
while (!stack.isEmpty())
{
    cout << stack.pop() << endl;
}
```

Il va sortir: troisième, deuxième, premier.

`QStack` hérite de `QVector` , son implémentation est donc très différente de celle de STL. En STL, `std::stack` est implémenté en tant que wrapper pour taper pass pour un argument de template (deque par défaut). Les principales opérations restent les mêmes pour `QStack` et pour `std::stack` .

Utilisation de QVector

`QVector<T>` fournit une classe de modèle de tableau dynamique. Il fournit de meilleures performances dans la plupart des cas que `QList<T>` , il devrait donc être le premier choix.

Il peut être initialisé de différentes manières:

```
QVector<int> vect;
vect << 1 << 2 << 3;

QVector<int> v {1, 2, 3, 4};
```

La dernière concerne la liste d'initialisation.

```
QVector<QString> stringsVector;
```

```
stringsVector.append("First");
stringsVector.append("Second");
```

Vous pouvez obtenir le iième élément de vecteur de cette façon:

`v[i]` ou `at[i]`

Assurez-vous que `i` est une position valide, même `at(i)` ne fait pas de vérification, c'est une différence par rapport à `std::vector` .

Utilisation de QLinkedList

Dans Qt, vous devez utiliser QLinkedList si vous devez implémenter une liste chaînée .

Il est rapide d'ajouter, d'ajouter, d'insérer des éléments dans `QLinkedList` -O (1), mais la recherche d'index est plus lente que dans `QList` ou `QVector` -O (n). C'est normal de prendre en compte que vous devez parcourir les nœuds pour trouver quelque chose dans la liste chaînée.

Le tableau complet de la complexité algorithmique peut être trouvé ici .

Juste pour insérer des éléments dans `QLinkedList` vous pouvez utiliser operator `<<()` :

```
QLinkedList<QString> list;
list << "string1" << "string2" << "string3";
```

Pour insérer des éléments au milieu de `QLinkedList` ou modifier tout ou partie de ses éléments, vous pouvez utiliser des itérateurs de style Java ou de style STL. Voici un exemple simple de la façon dont nous multiplions tous les éléments de `QLinkedList` par 2:

```
QLinkedList<int> integerList {1, 2, 3};
QLinkedList<int>::iterator it;
for (it = integerList.begin(); it != integerList.end(); ++it)
{
    *it *= 2;
}
```

QList

La classe `QList` est une classe de modèle qui fournit des listes. Il stocke les éléments dans une liste qui fournit un accès rapide par index et des insertions et des suppressions basées sur des index.

Pour insérer des éléments dans la liste, vous pouvez utiliser `operator<<()` , `insert()` , `append()` ou `prepend()` . Par exemple:

`operator<<()`

```
QList<QString> list;
list << "one" << "two" << "three";
```

insert()

```
QList<QString> list;
list << "alpha" << "beta" << "delta";
list.insert(2, "gamma");
```

append()

```
QList<QString> list;
list.append("one");
list.append("two");
list.append("three");
```

prepend()

```
QList<QString> list;
list.prepend("one");
list.prepend("two");
list.prepend("three");
```

Pour accéder à l'élément à une position d'index particulière, vous pouvez utiliser l' `operator[]()` ou `at()` . `at()` peut être plus rapide que `operator[]()` , ne provoque jamais de copie profonde du conteneur et doit fonctionner en temps constant. Aucun d'entre eux ne vérifie les arguments. Exemples:

```
if (list[0] == "mystring")
    cout << "mystring found" << endl;
```

Ou

```
if (list.at(i) == "mystring")
    cout << "mystring found at position " << i << endl;
```

Pour supprimer des éléments, il existe des fonctions telles que `removeAt()` , `takeAt()` , `takeFirst()` , `takeLast()` , `removeFirst()` , `removeLast()` ou `removeOne()` . Exemples:

takeFirst()

```
// takeFirst() removes the first item in the list and returns it
QList<QWidget *> list;
...
while (!list.isEmpty())
    delete list.takeFirst();
```

removeOne()

```
// removeOne() removes the first occurrence of value in the list
QList<QString> list;
list << "sun" << "cloud" << "sun" << "rain";
list.removeOne("sun");
```

Pour rechercher toutes les occurrences d'une valeur particulière dans une liste, vous pouvez

utiliser `indexOf()` ou `lastIndexOf()` . Exemple:

`indexOf()`

```
int i = list.indexOf("mystring");
if (i != -1)
    cout << "First occurrence of mystring is at position " << i << endl;
```

Lire Qt Container Classes en ligne: https://riptutorial.com/fr/qt/topic/6303/qt-container-classes

Chapitre 18: QTimer

Remarques

QTimer peut également être utilisé pour demander à une fonction de s'exécuter dès que la boucle d'événement a traité tous les autres événements en attente. Pour ce faire, utilisez un intervalle de 0 ms.

```
// option 1: Set the interval to 0 explicitly.
QTimer *timer = new QTimer;
timer->setInterval( 0 );
timer->start();

// option 2: Passing 0 with the start call will set the interval as well.
QTimer *timer = new QTimer;
timer->start( 0 );

// option 3: use QTimer::singleShot with interval 0
QTimer::singleShot(0, [](){
    // do something
});
```

Examples

Exemple simple

L'exemple suivant montre comment utiliser un `QTimer` pour appeler un logement toutes les 1 secondes.

Dans l'exemple, nous utilisons un `QProgressBar` pour mettre à jour sa valeur et vérifier que le minuteur fonctionne correctement.

main.cpp

```
#include <QApplication>

#include "timer.h"

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

    Timer timer;
    timer.show();

    return app.exec();
}
```

timer.h

```
#ifndef TIMER_H
#define TIMER_H

#include <QWidget>

class QProgressBar;

class Timer : public QWidget
{
    Q_OBJECT

public:
    Timer(QWidget *parent = 0);

public slots:
    void updateProgress();

private:
    QProgressBar *progressBar;
};

#endif
```

timer.cpp

```
#include <QLayout>
#include <QProgressBar>
#include <QTimer>

#include "timer.h"

Timer::Timer(QWidget *parent)
    : QWidget(parent)
{
    QHBoxLayout *layout = new QHBoxLayout();

    progressBar = new QProgressBar();
    progressBar->setMinimum(0);
    progressBar->setMaximum(100);

    layout->addWidget(progressBar);
    setLayout(layout);

    QTimer *timer = new QTimer(this);
    connect(timer, &QTimer::timeout, this, &Timer::updateProgress);
    timer->start(1000);

    setWindowTitle(tr("Timer"));
    resize(200, 200);
}

void Timer::updateProgress()
{
    progressBar->setValue(progressBar->value()+1);
}
```

timer.pro

```
QT += widgets
```

```
HEADERS = \
          timer.h
SOURCES = \
           main.cpp \
           timer.cpp
```

Singleshot Timer avec fonction Lambda comme fente

Si un minuteur unique est requis, il est pratique d'avoir le logement comme fonction lambda à l'endroit où la minuterie est déclarée:

```
QTimer::singleShot(1000, []() { /*Code here*/ } );
```

A cause de ce bogue (QTBUG-26406) , c'est bien possible puisque Qt5.4.

Dans les versions précédentes de Qt5, cela devait être fait avec plus de code de plaque de chaudière:

```
  QTimer *timer = new QTimer(this);
  timer->setSingleShot(true);

  connect(timer, &QTimer::timeout, [=]() {
    /*Code here*/
    timer->deleteLater();
  } );
```

Utiliser QTimer pour exécuter du code sur le thread principal

```
void DispatchToMainThread(std::function<void()> callback)
{
    // any thread
    QTimer* timer = new QTimer();
    timer->moveToThread(qApp->thread());
    timer->setSingleShot(true);
    QObject::connect(timer, &QTimer::timeout, [=]()
    {
        // main thread
        callback();
        timer->deleteLater();
    });
    QMetaObject::invokeMethod(timer, "start", Qt::QueuedConnection, Q_ARG(int, 0));
}
```

Ceci est utile lorsque vous devez mettre à jour un élément de l'interface utilisateur à partir d'un thread. Gardez à l'esprit la durée de vie de toutes les références de rappel.

```
DispatchToMainThread([]
{
    // main thread
    // do UI work here
});
```

Le même code pourrait être adapté pour exécuter du code sur tout thread exécutant la boucle d'événement Qt, implémentant ainsi un mécanisme de répartition simple.

Utilisation de base

`QTimer` ajoute la fonctionnalité pour qu'une fonction / un emplacement spécifique soit appelé après un certain intervalle (plusieurs fois ou juste une fois).

Le `QTimer` permet ainsi à une application GUI de "vérifier" les choses régulièrement ou de gérer les délais d'attente **sans** avoir à démarrer manuellement un thread supplémentaire pour cela et à faire attention aux conditions de course, car le timer sera géré dans la boucle d'événement principal.

Une minuterie peut simplement être utilisée comme ceci:

```
QTimer* timer = new QTimer(parent); //create timer with optional parent object
connect(timer,&QTimer::timeout,[this](){ checkProgress(); }); //some function to check
something
timer->start(1000); //start with a 1s interval
```

Le temporisateur déclenche le signal de `timeout` la fin du temps et celui-ci sera appelé dans la boucle d'événement principal.

QTimer :: singleShot utilisation simple

Le **QTimer :: singleShot** est utilisé pour appeler un slot / lambda de **manière asynchrone** après n ms.

La syntaxe de base est la suivante:

```
QTimer::singleShot(myTime, myObject, SLOT(myMethodInMyObject()));
```

avec **myTime** le temps en ms, **myObject** l'objet qui contient la méthode et **myMethodInMyObject** l'emplacement à appeler

Donc, par exemple, si vous voulez avoir une minuterie qui écrit une ligne de débogage "bonjour!" toutes les 5 secondes:

.cpp

```
void MyObject::startHelloWave()
{
    QTimer::singleShot(5 * 1000, this, SLOT(helloWave()));
}

void MyObject::helloWave()
{
    qDebug() << "hello !";
    QTimer::singleShot(5 * 1000, this, SLOT(helloWave()));
}
```

.hh

```
class MyObject : public QObject {
    Q_OBJECT
    ...
    void startHelloWave();

private slots:
    void helloWave();
    ...
};
```

Lire QTimer en ligne: https://riptutorial.com/fr/qt/topic/4309/qtimer

Chapitre 19: Réseau Qt

Introduction

Qt Network fournit des outils pour utiliser facilement de nombreux protocoles réseau dans votre application.

Examples

Client TCP

Pour créer une connexion **TCP** dans Qt, nous utiliserons QTcpSocket . Tout d'abord, nous devons nous connecter avec `connectToHost` .

Par exemple, pour vous connecter à un serveur tcp local:
`_socket.connectToHost(QHostAddress("127.0.0.1"), 4242);`

Ensuite, si nous avons besoin de lire les données du serveur, nous devons connecter le signal readyRead à un emplacement. Comme ça:

```
connect(&_socket, SIGNAL(readyRead()), this, SLOT(onReadyRead()));
```

et enfin, on peut lire les données comme ça:

```
void MainWindow::onReadyRead()
{
    QByteArray datas = _socket.readAll();
    qDebug() << datas;
}
```

Pour écrire des données, vous pouvez utiliser la méthode `write(QByteArray)` :

```
_socket.write(QByteArray("ok !\n"));
```

Ainsi, un client TCP de base peut ressembler à ceci:

main.cpp:

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

mainwindow.h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QTcpSocket>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

public slots:
    void onReadyRead();

private:
    Ui::MainWindow *ui;
    QTcpSocket  _socket;
};

#endif // MAINWINDOW_H
```

mainwindow.cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

#include <QDebug>
#include <QHostAddress>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow),
    _socket(this)
{
    ui->setupUi(this);
    _socket.connectToHost(QHostAddress("127.0.0.1"), 4242);
    connect(&_socket, SIGNAL(readyRead()), this, SLOT(onReadyRead()));
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::onReadyRead()
{
    QByteArray datas = _socket.readAll();
    qDebug() << datas;
    _socket.write(QByteArray("ok !\n"));
}
```

mainwindow.ui: (vide ici)

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
 <class>MainWindow</class>
 <widget class="QMainWindow" name="MainWindow">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>400</width>
    <height>300</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>MainWindow</string>
  </property>
  <widget class="QWidget" name="centralWidget"/>
  <widget class="QMenuBar" name="menuBar">
   <property name="geometry">
    <rect>
     <x>0</x>
     <y>0</y>
     <width>400</width>
     <height>25</height>
    </rect>
   </property>
  </widget>
  <widget class="QToolBar" name="mainToolBar">
   <attribute name="toolBarArea">
    <enum>TopToolBarArea</enum>
   </attribute>
   <attribute name="toolBarBreak">
    <bool>false</bool>
   </attribute>
  </widget>
  <widget class="QStatusBar" name="statusBar"/>
 </widget>
 <layoutdefault spacing="6" margin="11"/>
 <resources/>
 <connections/>
</ui>
```

Serveur TCP

Créer un **serveur TCP** dans Qt est également très simple, en effet, la classe QTcpServer fournit déjà tout ce dont nous avons besoin pour faire le serveur.

Tout d'abord, nous devons écouter n'importe quelle adresse IP, un port aléatoire et faire quelque chose lorsqu'un client est connecté. comme ça:

```
_server.listen(QHostAddress::Any, 4242);
connect(&_server, SIGNAL(newConnection()), this, SLOT(onNewConnection()));
```

Ensuite, quand il y a une nouvelle connexion, nous pouvons l'ajouter à la liste des clients et nous préparer à lire / écrire sur le socket. Comme ça:

```
   QTcpSocket *clientSocket = _server.nextPendingConnection();
   connect(clientSocket, SIGNAL(readyRead()), this, SLOT(onReadyRead()));
   connect(clientSocket, SIGNAL(stateChanged(QAbstractSocket::SocketState)), this,
SLOT(onSocketStateChanged(QAbstractSocket::SocketState)));
   _sockets.push_back(clientSocket);
```

Le `stateChanged(QAbstractSocket::SocketState)` nous permet de supprimer le socket dans notre liste lorsque le client est déconnecté.

Donc, voici un serveur de chat de base:

main.cpp:

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

mainwindow.h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QTcpServer>
#include <QTcpSocket>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

public slots:
    void onNewConnection();
    void onSocketStateChanged(QAbstractSocket::SocketState socketState);
    void onReadyRead();
private:
    Ui::MainWindow *ui;
    QTcpServer  _server;
    QList<QTcpSocket*>  _sockets;
};

#endif // MAINWINDOW_H
```

mainwindow.cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

#include <QDebug>
#include <QHostAddress>
#include <QAbstractSocket>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow),
    _server(this)
{
    ui->setupUi(this);
    _server.listen(QHostAddress::Any, 4242);
    connect(&_server, SIGNAL(newConnection()), this, SLOT(onNewConnection()));
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::onNewConnection()
{
   QTcpSocket *clientSocket = _server.nextPendingConnection();
   connect(clientSocket, SIGNAL(readyRead()), this, SLOT(onReadyRead()));
   connect(clientSocket, SIGNAL(stateChanged(QAbstractSocket::SocketState)), this,
SLOT(onSocketStateChanged(QAbstractSocket::SocketState)));

    _sockets.push_back(clientSocket);
    for (QTcpSocket* socket : _sockets) {
        socket->write(QByteArray::fromStdString(clientSocket-
>peerAddress().toString().toStdString() + " connected to server !\n"));
    }
}

void MainWindow::onSocketStateChanged(QAbstractSocket::SocketState socketState)
{
    if (socketState == QAbstractSocket::UnconnectedState)
    {
        QTcpSocket* sender = static_cast<QTcpSocket*>(QObject::sender());
        _sockets.removeOne(sender);
    }
}

void MainWindow::onReadyRead()
{
    QTcpSocket* sender = static_cast<QTcpSocket*>(QObject::sender());
    QByteArray datas = sender->readAll();
    for (QTcpSocket* socket : _sockets) {
        if (socket != sender)
            socket->write(QByteArray::fromStdString(sender-
>peerAddress().toString().toStdString() + ": " + datas.toStdString()));
    }
}
```

(utilisez le même mainwindow.ui que l'exemple précédent)

Lire Réseau Qt en ligne: https://riptutorial.com/fr/qt/topic/9683/reseau-qt

Chapitre 20: Signaux et Slots

Introduction

Les signaux et les emplacements sont utilisés pour la communication entre objets. Le mécanisme de signaux et de créneaux est une caractéristique centrale de Qt. Dans la programmation par interface graphique, lorsque nous modifions un widget, nous voulons souvent qu'un autre widget soit notifié. Plus généralement, nous souhaitons que les objets de toute nature puissent communiquer entre eux. Les signaux sont émis par les objets lorsqu'ils changent d'état d'une manière qui peut être intéressante pour d'autres objets. Les emplacements peuvent être utilisés pour recevoir des signaux, mais ils sont également des fonctions membres normales.

Remarques

La documentation officielle sur ce sujet peut être trouvée ici .

Examples

Un petit exemple

Les signaux et les emplacements sont utilisés pour la communication entre objets. Le mécanisme des signaux et des slots est une fonctionnalité centrale de Qt et probablement la partie la plus différente des fonctionnalités fournies par les autres frameworks.

L'exemple minimal nécessite une classe avec un signal, un emplacement et une connexion:

counter.h

```
#ifndef COUNTER_H
#define COUNTER_H

#include <QWidget>
#include <QDebug>

class Counter : public QWidget
{
    /*
     * All classes that contain signals or slots must mention Q_OBJECT
     * at the top of their declaration.
     * They must also derive (directly or indirectly) from QObject.
     */
    Q_OBJECT

public:
    Counter (QWidget *parent = 0): QWidget(parent)
    {
            m_value = 0;

            /*
```

```
             * The most important line: connect the signal to the slot.
             */
            connect(this, &Counter::valueChanged, this, &Counter::printvalue);
    }

    void setValue(int value)
    {
        if (value != m_value) {
            m_value = value;
            /*
             * The emit line emits the signal valueChanged() from
             * the object, with the new value as argument.
             */
            emit valueChanged(m_value);
        }
    }

public slots:
    void printValue(int value)
    {
        qDebug() << "new value: " << value;
    }

signals:
    void valueChanged(int newValue);

private:
    int m_value;

};

#endif
```

La `main` définit une nouvelle valeur. Nous pouvons vérifier comment l'appel est appelé, en imprimant la valeur.

```
#include <QtGui>
#include "counter.h"

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

    Counter counter;
    counter.setValue(10);
    counter.show();

    return app.exec();
}
```

Enfin, notre dossier de projet:

```
SOURCES   = \
            main.cpp
HEADERS   = \
            counter.h
```

La nouvelle syntaxe de connexion Qt5

La syntaxe de `connect` conventionnelle qui utilise les macros `SIGNAL` et `SLOT` fonctionne entièrement au moment de l'exécution, ce qui présente deux inconvénients: elle comporte des surcharges d'exécution (entraînant également une surcharge de taille binaire) et aucune vérification de la correction à la compilation. La nouvelle syntaxe aborde les deux problèmes. Avant de vérifier la syntaxe dans un exemple, il est préférable de savoir ce qui se passe en particulier.

Disons que nous construisons une maison et que nous voulons connecter les câbles. C'est exactement ce que fait la fonction de connexion. Les signaux et les slots sont ceux qui ont besoin de cette connexion. Le fait est que si vous effectuez une connexion, vous devez faire attention aux autres connexions qui se chevauchent. Chaque fois que vous connectez un signal à un slot, vous essayez de dire au compilateur qu'à chaque fois que le signal est émis, invoquez simplement la fonction slot. C'est ce qui se passe exactement.

Voici un exemple de **main.cpp** :

```
#include <QApplication>
#include <QDebug>
#include <QTimer>

inline void onTick()
{
   qDebug() << "onTick()";
}

struct OnTimerTickListener {
   void onTimerTick()
   {
       qDebug() << "OnTimerTickListener::onTimerTick()";
   }
};

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

    OnTimerTickListener listenerObject;

    QTimer timer;
    // Connecting to a non-member function
    QObject::connect(&timer, &QTimer::timeout, onTick);
    // Connecting to an object member method
    QObject::connect(&timer, &QTimer::timeout, &listenerObject,
&OnTimerTickListener::onTimerTick);
    // Connecting to a lambda
    QObject::connect(&timer, &QTimer::timeout, [](){
        qDebug() << "lambda-onTick";
    });

    return app.exec();
}
```

Conseil: l'ancienne syntaxe (macros `SIGNAL` / `SLOT`) exige que le métacompilateur Qt (MOC) soit exécuté pour toute classe comportant des slots ou des signaux. Du point de vue du codage, cela

signifie que ces classes doivent avoir la macro `Q_OBJECT` (ce qui indique la nécessité d'exécuter MOC sur cette classe).

La nouvelle syntaxe, quant à elle, nécessite toujours MOC pour que les signaux fonctionnent, mais **pas** pour les slots. Si une classe ne dispose que de slots et de signaux, elle n'a pas besoin de la macro `Q_OBJECT` et ne peut donc pas invoquer la MOC, ce qui réduit non seulement la taille binaire finale mais réduit également le temps de compilation (aucun appel MOC `*_moc.cpp` fichier `*_moc.cpp`).

Connexion de signaux / slots surchargés

Bien qu'elle soit meilleure à bien des égards, la nouvelle syntaxe de connexion de Qt5 présente une grande faiblesse: la connexion de signaux et de logements surchargés. Afin de permettre au compilateur de résoudre les surcharges, nous devons utiliser `static_cast` s pour les pointeurs des fonctions membres, ou (à partir de Qt 5.7) `qOverload` et amis:

```
#include <QObject>

class MyObject : public QObject
{
    Q_OBJECT
public:
    explicit MyObject(QObject *parent = nullptr) : QObject(parent) {}

public slots:
    void slot(const QString &string) {}
    void slot(const int integer) {}

signals:
    void signal(const QString &string) {}
    void signal(const int integer) {}
};

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

    // using pointers to make connect calls just a little simpler
    MyObject *a = new MyObject;
    MyObject *b = new MyObject;

    // COMPILE ERROR! the compiler does not know which overloads to pick :(
    QObject::connect(a, &MyObject::signal, b, &MyObject::slot);

    // this works, now the compiler knows which overload to pick, it is very ugly and hard to
remember though...
    QObject::connect(
        a,
        static_cast<void(MyObject::*)(int)>(&MyObject::signal),
        b,
        static_cast<void(MyObject::*)(int)>(&MyObject::slot));

    // ...so starting in Qt 5.7 we can use qOverload and friends:
    // this requires C++14 enabled:
    QObject::connect(
        a,
```

```
        qOverload<int>(&MyObject::signal),
        b,
        qOverload<int>(&MyObject::slot));

    // this is slightly longer, but works in C++11:
    QObject::connect(
        a,
        QOverload<int>::of(&MyObject::signal),
        b,
        QOverload<int>::of(&MyObject::slot));

    // there are also qConstOverload/qNonConstOverload and QConstOverload/QNonConstOverload,
the names should be self-explanatory
}
```

Connexion multi-fenêtre

Un exemple simple à plusieurs fenêtres utilisant des signaux et des slots.

Il existe une classe MainWindow qui contrôle la vue de la fenêtre principale. Une deuxième fenêtre contrôlée par la classe du site Web.

Les deux classes sont connectées de sorte que lorsque vous cliquez sur un bouton de la fenêtre du site Web, quelque chose se produit dans la fenêtre principale (une étiquette de texte est modifiée).

J'ai fait un exemple simple qui est aussi sur GitHub :

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "website.h"

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

public slots:
    void changeText();

private slots:
    void on_openButton_clicked();

private:
    Ui::MainWindow *ui;
```

```
    //You want to keep a pointer to a new Website window
    Website* webWindow;
};

#endif // MAINWINDOW_H
```

mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::changeText()
{
    ui->text->setText("New Text");
    delete webWindow;
}

void MainWindow::on_openButton_clicked()
{
    webWindow = new Website();
    QObject::connect(webWindow, SIGNAL(buttonPressed()), this, SLOT(changeText()));
    webWindow->show();
}
```

website.h

```
#ifndef WEBSITE_H
#define WEBSITE_H

#include <QDialog>

namespace Ui {
class Website;
}

class Website : public QDialog
{
    Q_OBJECT

public:
    explicit Website(QWidget *parent = 0);
    ~Website();

signals:
    void buttonPressed();

private slots:
```

```
    void on_changeButton_clicked();

private:
    Ui::Website *ui;
};

#endif // WEBSITE_H
```

website.cpp

```
#include "website.h"
#include "ui_website.h"

Website::Website(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Website)
{
    ui->setupUi(this);
}

Website::~Website()
{
    delete ui;
}

void Website::on_changeButton_clicked()
{
    emit buttonPressed();
}
```

Composition du projet:

```
SOURCES += main.cpp\
        mainwindow.cpp \
    website.cpp

HEADERS  += mainwindow.h \
    website.h

FORMS    += mainwindow.ui \
    website.ui
```

Considérez le Uis à composer:

- Fenêtre principale: une étiquette appelée "texte" et un bouton appelé "openButton"
- Fenêtre du site Web: un bouton appelé "changeButton"

Les points-clés sont donc les connexions entre les signaux et les slots et la gestion des pointeurs ou des références de Windows.

Lire Signaux et Slots en ligne: https://riptutorial.com/fr/qt/topic/2136/signaux-et-slots

Chapitre 21: SQL sur Qt

Examples

Connexion de base et requête

La classe QSqlDatabase fournit une interface pour accéder à une base de données via une connexion. Une instance de QSqlDatabase représente la connexion. La connexion permet d'accéder à la base de données via l'un des pilotes de base de données pris en charge. Assurez-vous d'ajouter

```
QT += SQL
```

dans le fichier .pro. Supposons qu'une base de données SQL nommée TestDB avec un countryTable contienne la colonne suivante:

```
| country |
-----------
| USA     |
```

Pour interroger et obtenir des données SQL depuis TestDB:

```
#include <QtGui>
#include <QtSql>

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

    QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL"); // Will use the driver referred to
by "QPSQL" (PostgreSQL Driver)
    db.setHostName("TestHost");
    db.setDatabaseName("TestDB");
    db.setUserName("Foo");
    db.setPassword("FooPass");

    bool ok = db.open();
    if(ok)
    {
        QSqlQuery query("SELECT country FROM countryTable");
        while (query.next())
        {
          QString country = query.value(0).toString();
          qWarning() << country; // Prints "USA"
        }
    }

    return app.exec();
}
```

Paramètres de requête Qt SQL

Il est souvent pratique de séparer la requête SQL des valeurs réelles. Cela peut être fait en utilisant des espaces réservés. Qt prend en charge deux syntaxes d'espace réservé: la liaison nommée et la liaison de position.

liaison nommée:

```
QSqlQuery query;
query.prepare("INSERT INTO employee (id, name, salary) VALUES (:id, :name, :salary)");
query.bindValue(":id", 1001);
query.bindValue(":name", "Thad Beaumont");
query.bindValue(":salary", 65000);
query.exec();
```

liaison de position:

```
QSqlQuery query;
query.prepare("INSERT INTO employee (id, name, salary) VALUES (?, ?, ?)");
query.addBindValue(1001);
query.addBindValue("Thad Beaumont");
query.addBindValue(65000);
query.exec();
```

Notez qu'avant d'appeler `bindValue()` ou `addBindValue()` vous devez appeler QSqlQuery :: prepare () une fois.

Connexion à la base de données MS SQL Server à l'aide de QODBC

Lorsque vous essayez d'ouvrir une connexion de base de données avec QODBC, veuillez vous assurer que

- Vous avez un pilote QODBC disponible
- Votre serveur a une interface ODBC et est activé pour (cela dépend de vos installations de pilote ODBC)
- utiliser l'accès à la mémoire partagée, les connexions TCP / IP ou la connexion par canal nommé.

Toutes les connexions nécessitent uniquement que DatabaseName soit défini en appelant QSqlDatabase :: setDatabaseName.

Ouvrir la connexion en utilisant l'accès à la mémoire partagée

Pour que cette option fonctionne, vous devez avoir accès à la mémoire de la machine et disposer des autorisations nécessaires pour accéder à la mémoire partagée. Pour utiliser une connexion en mémoire partagée, il est nécessaire de définir lpc: devant la chaîne du serveur. La connexion à l'aide de SQL Server Native Client 11 est effectuée en procédant comme suit:

```
QString connectString = "Driver={SQL Server Native Client 11.0};";                    //
Driver is now {SQL Server Native Client 11.0}
connectString.append("Server=lpc:"+QHostInfo::localHostName()+"\\SQLINSTANCENAME;");   //
Hostname,SQL-Server Instance
```

```
connectString.append("Database=SQLDBSCHEMA;");  // Schema
connectString.append("Uid=SQLUSER;");           // User
connectString.append("Pwd=SQLPASS;");           // Pass
db.setDatabaseName(connectString);

if(db.open())
{
    ui->statusBar->showMessage("Connected");
}
else
{
    ui->statusBar->showMessage("Not Connected");
}
```

Connexion ouverte à l'aide d'un tuyau nommé

Cette option nécessite que votre connexion ODBC ait un DSN complet. La chaîne du serveur est configurée à l'aide du Windows Computername et du nom de l'instance du serveur SQL. L'exemple de connexion sera ouvert à l'aide de SQL Server Native Client 10.0

```
QString connectString = "Driver={SQL Server Native Client 10.0};"; // Driver can also be {SQL
Server Native Client 11.0}
connectString.append("Server=SERVERHOSTNAME\\SQLINSTANCENAME;");   // Hostname,SQL-Server
Instance
connectString.append("Database=SQLDBSCHEMA;");  // Schema
connectString.append("Uid=SQLUSER;");           // User
connectString.append("Pwd=SQLPASS;");           // Pass
db.setDatabaseName(connectString);

if(db.open())
{
    ui->statusBar->showMessage("Connected");
}
else
{
    ui->statusBar->showMessage("Not Connected");
}
```

Ouvrir la connexion en utilisant TCP / IP

Pour ouvrir une connexion TCP / IP, le serveur doit être configuré pour autoriser les connexions sur un port fixe, sinon vous devrez d'abord interroger le port actuellement actif. Dans cet exemple, nous avons un port fixe à 5171. Vous pouvez trouver un exemple de configuration du serveur pour autoriser les connexions sur un port fixe à 1 . Pour ouvrir une connexion utilisant TCP / IP, utilisez un tuple des serveurs IP et Port:

```
QString connectString = "Driver={SQL Server};"; // Driver is now {SQL Server}
connectString.append("Server=10.1.1.15,5171;"); // IP,Port
connectString.append("Database=SQLDBSCHEMA;");  // Schema
connectString.append("Uid=SQLUSER;");           // User
connectString.append("Pwd=SQLPASS;");           // Pass
db.setDatabaseName(connectString);

if(db.open())
```

```
{
    ui->statusBar->showMessage("Connected");
}
else
{
    ui->statusBar->showMessage("Not Connected");
}
```

Lire SQL sur Qt en ligne: https://riptutorial.com/fr/qt/topic/10628/sql-sur-qt

Chapitre 22: Système de ressources Qt

Introduction

Le système Qt Resource est un moyen d'intégrer des fichiers dans votre projet. Chaque fichier de ressources peut avoir un ou plusieurs *préfixes* et chaque *préfixe* peut *contenir* des fichiers.

Chaque fichier dans les ressources est un lien vers un fichier du système de fichiers. Lorsque l'exécutable est construit, les fichiers sont regroupés dans l'exécutable, de sorte que le fichier d'origine n'a pas besoin d'être distribué avec le binaire.

Examples

Référencement de fichiers dans le code

Disons que dans un fichier de ressources, vous aviez un fichier appelé /icons/ok.png

L'URL complète de ce fichier dans le code est `qrc:/icons/ok.png` . Dans la plupart des cas, cela peut être raccourci à `:/icons/ok.png`

Par exemple, si vous souhaitez créer un QIcon et le définir comme l'icône d'un bouton de ce fichier, vous pouvez utiliser

```
QIcon icon(":/icons/ok.png"); //Alternatively use qrc:/icons/ok.png
ui->pushButton->setIcon(icon);
```

Lire Système de ressources Qt en ligne: https://riptutorial.com/fr/qt/topic/8776/systeme-de-ressources-qt

Chapitre 23: Threading et concomitance

Remarques

Quelques notes déjà mentionnées dans les docs officiels ici et ici :

- Si un objet a un parent, il doit être dans le même thread que le parent, c.-à-d. Qu'il ne peut pas être déplacé vers un nouveau thread, et que vous ne pouvez pas définir un parent sur un objet si le parent et l'objet vivent dans des threads différents
- Lorsqu'un objet est déplacé vers un nouveau thread, tous ses enfants sont également déplacés vers le nouveau thread
- Vous ne pouvez que *pousser des* objets vers un nouveau thread. Vous ne pouvez pas les *tirer* vers un nouveau thread, c'est-à-dire que vous ne pouvez appeler que `moveToThread` partir du thread où l'objet vit actuellement

Examples

Utilisation basique de QThread

`QThread` est un handle vers un thread de plate-forme. Il vous permet de gérer le thread en surveillant sa durée de vie et en lui demandant de terminer son travail.

Dans la plupart des cas, l'héritage de la classe n'est pas recommandé. La méthode d' `run` par défaut lance une boucle d'événements pouvant envoyer des événements aux objets de la classe. Les connexions d'intervalle de signal entre threads sont implémentées en envoyant un `QMetaCallEvent` à l'objet cible.

Une instance de `QObject` peut être déplacée vers un thread, où elle traitera ses événements, tels que les événements du minuteur ou les appels d'emplacement / méthode.

Pour travailler sur un thread, commencez par créer votre propre classe de travail `QObject` de `QObject` . Puis déplacez-le sur le fil. L'objet peut exécuter son propre code automatiquement, par exemple en utilisant `QMetaObject::invokeMethod()` .

```
#include <QObject>

class MyWorker : public QObject
{
    Q_OBJECT
public:
    Q_SLOT void doWork() {
        qDebug() << "doWork()" << QThread::currentThread();
        // and do some long operation here
    }
    MyWorker(QObject * parent = nullptr) : QObject{parent} {}
};

class MyController : public QObject
```

```
{
    Q_OBJECT
    Worker worker;
    QThread workerThread;
public:
    MyController()  {
        worker.moveToThread(&workerThread);
        // provide meaningful debug output
        workerThread.setObjectName("workerThread");
        workerThread.start();
        // the thread starts the event loop and blocks waiting for events
    }
    ~MyController()  {
        workerThread.quit();
        workerThread.wait();
    }
    void operate() {
        // Qt::QueuedConnection ensures that the slot is invoked in its own thread
        QMetaObject::invokeMethod(&worker, "doWork", Qt::QueuedConnection);
    }
};
```

Si votre agent doit être éphémère et n'existe que pendant son travail, il est préférable de soumettre un foncteur ou une méthode thread-safe pour exécution dans le pool de threads via `QtConcurrent::run` .

QtConcurrent Run

Si vous trouvez que la gestion de QThreads et de primitives de bas niveau telles que les mutex ou les sémaphores est trop complexe, Qt Concurrent est ce que vous recherchez. Il comprend des classes qui permettent une gestion plus poussée des threads.

Regardons Exécution simultanée. `QtConcurrent::run()` permet d'exécuter une fonction dans un nouveau thread. Quand aimeriez-vous l'utiliser? Lorsque vous avez une longue opération et que vous ne voulez pas créer de thread manuellement.

Maintenant le code:

```
#include <qtconcurrentrun.h>

void longOperationFunction(string parameter)
{
    // we are already in another thread
    // long stuff here
}

void mainThreadFunction()
{
    QFuture<void> f = run(longOperationFunction, "argToPass");
    f.waitForFinished();
}
```

Les choses sont donc simples: lorsque nous devons exécuter une autre fonction dans un autre thread, appelez simplement `QtConcurrent::run` , pass function et ses paramètres et voilà!

`QFuture` présente le résultat de notre calcul asynchrone. Dans le cas de `QtConcurrent::run` nous ne pouvons pas annuler l'exécution de la fonction.

Invocation de slots à partir d'autres threads

Lorsqu'une boucle d'événement Qt est utilisée pour effectuer des opérations et qu'un utilisateur non-Qt-saavy doit interagir avec cette boucle d'événement, l'écriture de l'emplacement pour gérer des appels réguliers à partir d'un autre thread peut simplifier les choses pour les autres utilisateurs.

main.cpp:

```
#include "OperationExecutioner.h"
#include <QCoreApplication>
#include <QThread>

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

    QThread thrd;
    thrd.setObjectName("thrd");
    thrd.start();
    while(!thrd.isRunning())
        QThread::msleep(10);

    OperationExecutioner* oe = new OperationExecutioner;
    oe->moveToThread(&thrd);
    oe->doIt1(123,'A');
    oe->deleteLater();
    thrd.quit();
    while(!thrd.isFinished())
        QThread::msleep(10);

    return 0;
}
```

OperationExecutioner.h:

```
#ifndef OPERATION_EXECUTIONER_H
#define OPERATION_EXECUTIONER_H

#include <QObject>

class OperationExecutioner : public QObject
{
    Q_OBJECT
public slots:
    void doIt1(int argi, char argc);
};

#endif  // OPERATION_EXECUTIONER_H
```

OperationExecutioner.cpp:

```
#include "OperationExecutioner.h"
#include <QMetaObject>
#include <QThread>
#include <QDebug>

void OperationExecutioner::doIt1(int argi, char argc)
{
    if (QThread::currentThread() != thread()) {
        qInfo() << "Called from thread" << QThread::currentThread();
        QMetaObject::invokeMethod(this, "doIt1", Qt::QueuedConnection,
                                  Q_ARG(int,argi), Q_ARG(char,argc));
        return;
    }

    qInfo() << "Called from thread" << QThread::currentThread()
            << "with args" << argi << argc;
}
```

OperationExecutioner.pro:

```
HEADERS += OperationExecutioner.h
SOURCES += main.cpp OperationExecutioner.cpp
QT -= gui
```

Lire Threading et concomitance en ligne: https://riptutorial.com/fr/qt/topic/5022/threading-et-concomitance

Chapitre 24: Utilisation efficace des feuilles de style

Examples

Définition de la feuille de style d'un widget d'interface utilisateur

Vous pouvez définir la feuille de style du widget d'interface utilisateur souhaité à l'aide de tout code CSS valide. L'exemple ci-dessous définira une couleur de texte QLabel comme une bordure.

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    QString style = "color: blue; border: solid black 5px;";
    ui->myLabel->setStylesheet(style); //This can use colors RGB, HSL, HEX, etc.
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

Lire Utilisation efficace des feuilles de style en ligne:
https://riptutorial.com/fr/qt/topic/5931/utilisation-efficace-des-feuilles-de-style

Crédits

S. No	Chapitres	Contributeurs
1	Démarrer avec Qt	agilob, Christopher Aldama, Community, demonplus, devbean, Dmitriy, Donald Duck, fat, Gabriel de Grimouard, Kamalpreet Grewal, Maxito, Tarod, thiagofalcao
2	A propos de l'utilisation des mises en page, du widget	Gabriel de Grimouard
3	CMakeLists.txt pour votre projet Qt	Athena, demonplus, Robert, Velkan, wasthishelpful
4	Communication entre QML et C ++	Gabriel de Grimouard, Martin Zhai
5	Construire QtWebEngine depuis le source	Martin Zhai
6	Déploiement d'applications Qt	Luca Angioloni, Martin Zhai, Nathan Osman, TriskalJM, wasthishelpful
7	En-tête sur QListView	Papipone
8	Modèle / vue	Jan, KernelPanic, Tim D
9	Multimédia	demonplus, Gabriel de Grimouard
10	Partage implicite	Hayt
11	Pièges courants	e.jahandar
12	QDialogs	Wilmort
13	QGraphics	Chris, demonplus
14	qmake	Caleb Huitt - cjhuitt, demonplus, doc, Gregor, Jon Harper
15	QObject	demonplus, Hayt
16	Qt - Traitement des bases de données	Jan, Rinat, Shihe Zhang, Zylva

17	Qt Container Classes	demonplus, Tarod
18	QTimer	avb, Caleb Huitt - cjhuitt, Eugene, Gabriel de Grimouard, Hayt, Rinat, Tarod, thuga, tpr, Victor Tran
19	Réseau Qt	Gabriel de Grimouard
20	Signaux et Slots	Athena, devbean, fat, immerhart, Jan, Luca Angioloni, Robert, Tarod, Violet Giraffe
21	SQL sur Qt	Noam M
22	Système de ressources Qt	Victor Tran
23	Threading et concomitance	demonplus, gmabey, Kuba Ober, Nathan Osman, RamenChef, thuga
24	Utilisation efficace des feuilles de style	Nicholas Johnson