EBook Gratis

APRENDIZAJE Qt

Free unaffiliated eBook created from **Stack Overflow contributors.**

#qt

Tabla de contenido

Acerca de 1

Capítulo 1: Empezando con Qt 2

Observaciones 2

Versiones 2

Examples 2

Instalación y configuración en Windows y Linux 2

Hola Mundo 8

Aplicación básica con QtCreator y QtDesigner 9

Capítulo 2: Clases de Contenedores Qt 16

Observaciones 16

Examples 16

Uso de QStack 16

Uso de QVector 16

Uso de QLinkedList 17

QList 17

Capítulo 3: CMakeLists.txt para su proyecto Qt 20

Examples 20

CMakeLists.txt para Qt 5 20

Capítulo 4: Comunicación entre QML y C ++. 22

Introducción 22

Examples 22

Llama a C ++ en QML 22

Llame a QML en C ++ 23

Capítulo 5: Construye QtWebEngine desde la fuente 28

Introducción 28

Examples 28

Construir en Windows 28

Capítulo 6: Despliegue de aplicaciones Qt 29

Examples 29

Despliegue en windows 29

Integración con CMake 29
Implementación en Mac 30
Despliegue en linux 31
Capítulo 7: Encabezado en QListView 32
Introducción 32
Examples 32
Declaración personalizada QListView 32
Implementación del QListView personalizado. 33
Caso de uso: declaración de MainWindow 34
Caso de uso: Implementación 34
Caso de uso: Salida de muestra 35
Capítulo 8: Errores comunes 37
Examples 37
Uso de Qt: DirectConnection cuando el objeto receptor no recibe señal 37
Capítulo 9: Intercambio implícito 39
Observaciones 39
Examples 39
Concepto basico 39
Capítulo 10: Modelo / Vista 41
Examples 41
Una tabla simple de solo lectura para ver datos de un modelo 41
Un modelo de árbol simple. 44
Capítulo 11: Multimedia 48
Observaciones 48
Examples 48
Reproducción de video en Qt 5 48
Reproducción de audio en Qt5 48
Capítulo 12: QDialogs 50
Observaciones 50
Examples 50
MyCompareFileDialog.h 50
MyCompareFileDialogDialog.cpp 50

MainWindow.h 51
MainWindow.cpp 51
main.cpp 52
mainwindow.ui 52
Capítulo 13: Qgraphics 54
Examples 54
Panorámica, zoom y rotación con QGraphicsView 54
Capítulo 14: qmake 56
Examples 56
Perfil por defecto. 56
Preservar la estructura del directorio de origen en una opción de compilación (no document 56
Ejemplo simple (Linux) 57
Ejemplo de SUBDIRS 58
Ejemplo de biblioteca 60
Creando un archivo de proyecto a partir de código existente 60
Capítulo 15: QObject 62
Observaciones 62
Examples 62
Ejemplo de QObject 62
qobject_cast 62
QObject vida y propiedad 63
Capítulo 16: Qt - Tratar con bases de datos 65
Observaciones 65
Examples 65
Usando una base de datos en Qt 65
Qt - Tratar con bases de datos Sqlite 66
Qt - Tratar con bases de datos ODBC 67
Qt - Tratar con las bases de datos Sqlite en memoria 69
Eliminar la conexión de la base de datos correctamente 70
Capítulo 17: QTimer 72
Observaciones 72
Examples 72

Ejemplo simple 72
Temporizador Singleshot con función Lambda como ranura 74
Usando QTimer para ejecutar código en el hilo principal 74
Uso básico 75
QTimer :: singleShot uso simple 75
Capítulo 18: Red qt 77
Introducción 77
Examples 77
Cliente TCP 77
Servidor TCP 79
Capítulo 19: Roscado y concurrencia 83
Observaciones 83
Examples 83
Uso básico de QThread 83
QtConcurrent Run 84
Invocando ranuras de otros hilos 85
Capítulo 20: Señales y Slots 87
Introducción 87
Observaciones 87
Examples 87
Un pequeño ejemplo 87
La nueva sintaxis de conexión Qt5 89
Conexión de señales / slots sobrecargados 90
Conexión de ranura de señal de ventana múltiple 91
Capítulo 21: Sistema de Recursos Qt 94
Introducción 94
Examples 94
Referencias de archivos dentro del código 94
Capítulo 22: Sobre el uso de diseños, la crianza de widgets 95
Introducción 95
Observaciones 95
Examples 95

Disposición horizontal básica 95

Diseño Vertical Básico 96

Combinando diseños 97

Ejemplo de diseño de cuadrícula 98

Capítulo 23: SQL en Qt 101

Examples 101

Conexión básica y consulta. 101

Parámetros de consulta Qt SQL 101

Conexión de base de datos de MS SQL Server utilizando QODBC 102

Capítulo 24: Usar hojas de estilo con eficacia 105

Examples 105

Configuración de la hoja de estilo de un widget UI 105

Creditos 106

Acerca de

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.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

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

Capítulo 1: Empezando con Qt

Observaciones

Como se indica en la documentación oficial , Qt es un marco de desarrollo de aplicaciones multiplataforma para computadoras de escritorio, integradas y móviles. Las plataformas compatibles incluyen Linux, OS X, Windows, VxWorks, QNX, Android, iOS, BlackBerry, Sailfish OS y otros.

Esta sección proporciona una descripción general de qué es Qt y por qué un desarrollador puede querer usarlo.

También debe mencionar cualquier tema importante dentro de Qt, y vincular a los temas relacionados. Dado que la documentación para qt es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

Versiones

Versión	Fecha de lanzamiento
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

Instalación y configuración en Windows y Linux

Descargar Qt para Linux Open Source Version

Vaya a https://www.qt.io/download-open-source/ y haga clic en Descargar ahora, asegúrese de

descargar el instalador Qt para Linux.

Recommended

We detected your operating system as: Linux
Recommended download: Qt Online Installer for Linux

Before you begin your download, please make sure you:

› learn about the obligations of the LGPL.

› read the FAQ about developing with the LGPL.

Download Now

Qt online installer is a small executable which downloads content over internet based on your selections. It provides all Qt 5.x binary & source packages and latest Qt Creator.

For more information visit our Developers page.
Not the download package you need? View All Downloads

Se descargará un archivo con el nombre qt-unified-linux-x-online.run, luego agregue el permiso exec

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

Recuerde cambiar 'x' para la versión real del instalador. Luego ejecuta el instalador

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

Descargar Qt para Windows Open Source Version

Vaya a https://www.qt.io/download-open-source/ . La siguiente captura de pantalla muestra la página de descarga en Windows:

Your download

We detected your operating system as: Windows
Recommended download: Qt Online Installer for Windows

Before you begin your download, please make sure you:

- learn about the obligations of the LGPL.
- read the FAQ about developing with the LGPL.

Download Now

Qt online installer is a small executable which downloads content over inte
based on your selections. It provides all Qt 5.x binary & source packages ar
latest Qt Creator.

For more information visit our Developers page.
Not the download package you need? View All Downloads

Lo que debe hacer ahora depende del IDE que vaya a utilizar. Si va a utilizar Qt Creator, que se incluye en el programa de instalación, simplemente haga clic en Descargar ahora y ejecute el archivo ejecutable.

Si va a usar Qt en Visual Studio, normalmente el botón Descargar ahora también debería funcionar. Asegúrese de que el archivo descargado se llame qt-opensource-windows-x86-msvc2015_64-xxxexe o qt-opensource-windows-x86-msvc2015_32-xxxexe (donde xxx es la versión de Qt, por ejemplo 5.7.0). Si ese no es el caso, haga clic en Ver todas las descargas y seleccione una de las primeras cuatro opciones en Windows Host.

Si va a utilizar Qt en Code :: Blocks, haga clic en Ver todas las descargas y seleccione Qt xxx para Windows de 32 bits (MinGW xxx, 1.2 GB) en Windows Host.

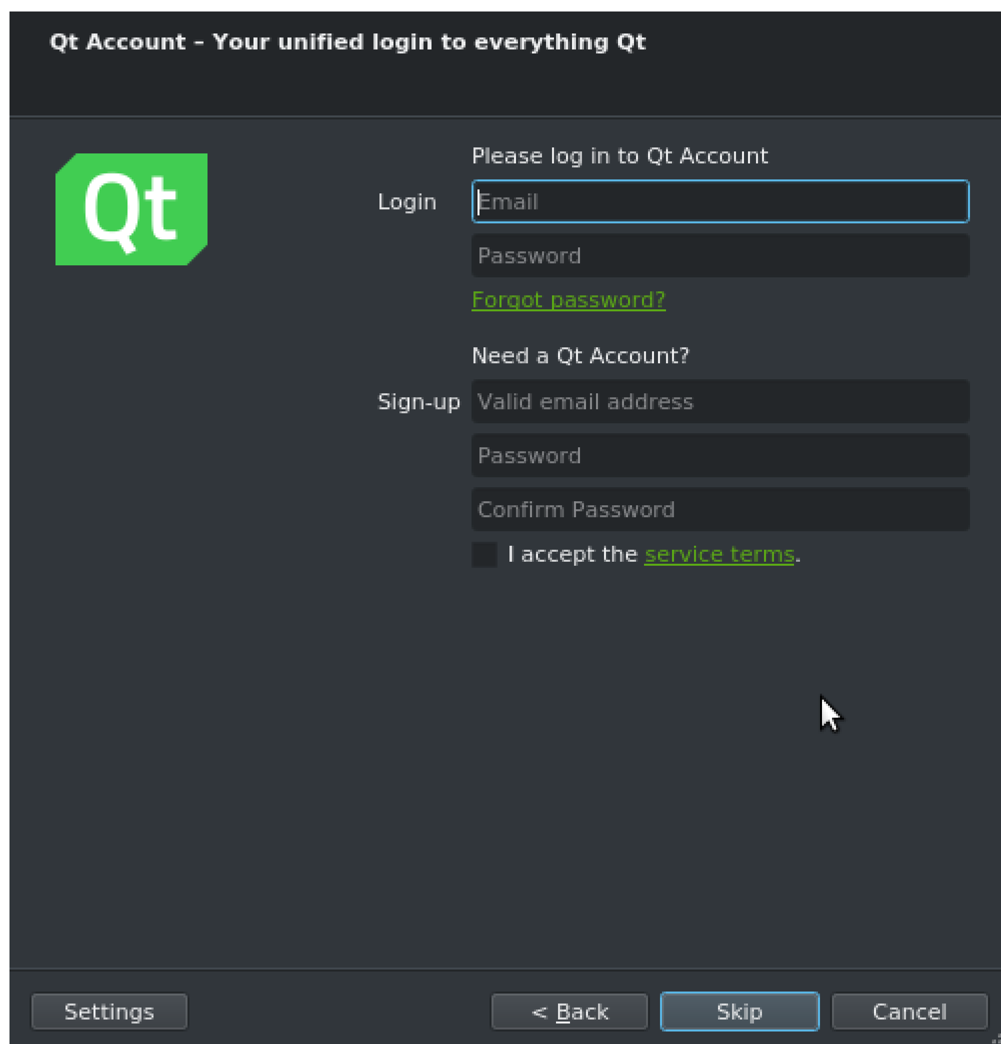
Una vez que haya descargado el archivo de instalación apropiado, ejecute el archivo ejecutable y siga las instrucciones a continuación. Tenga en cuenta que necesita ser administrador para

instalar Qt. Si no eres administrador, puedes encontrar varias soluciones alternativas aquí .

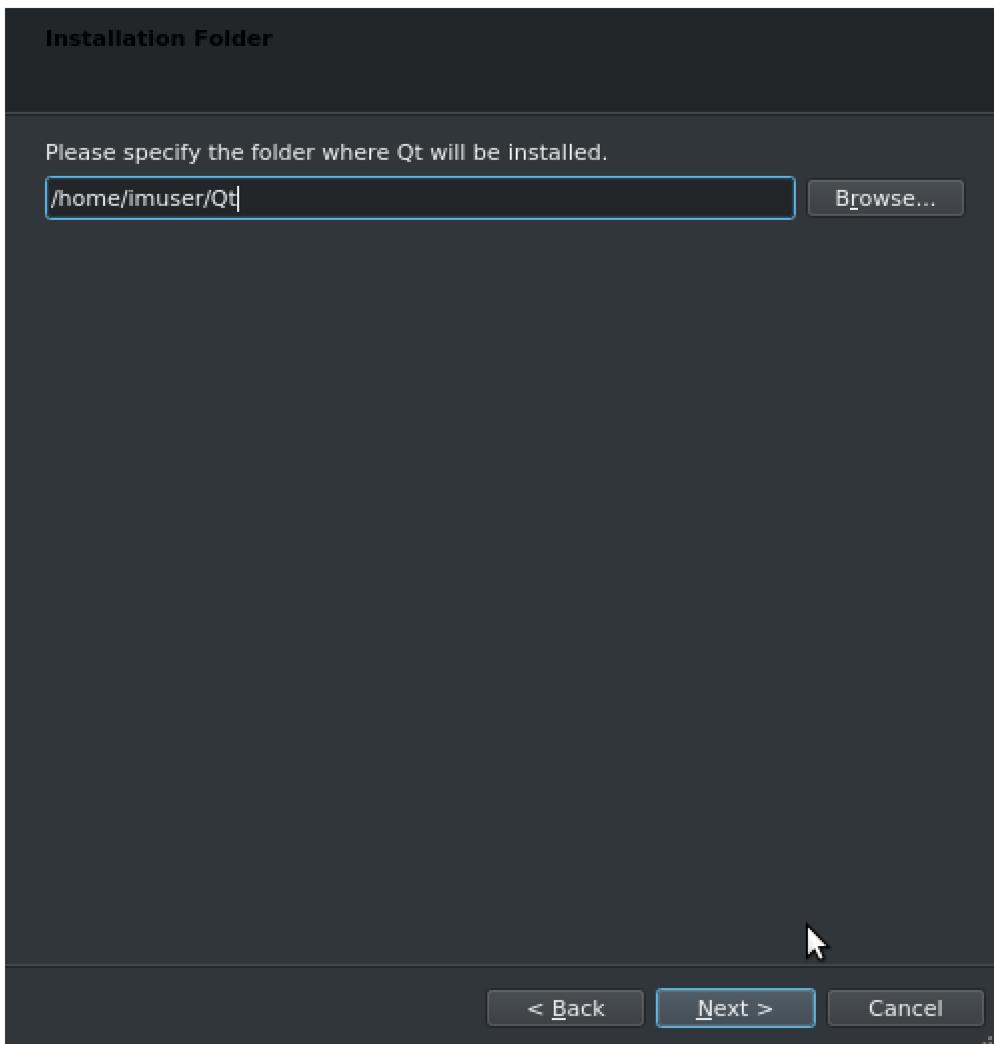
Instalar Qt en cualquier sistema operativo.

Una vez que haya descargado Qt y haya abierto el programa de instalación, el procedimiento de instalación es el mismo para todos los sistemas operativos, aunque las capturas de pantalla pueden parecer un poco diferentes. Las capturas de pantalla proporcionadas aquí son de Linux.

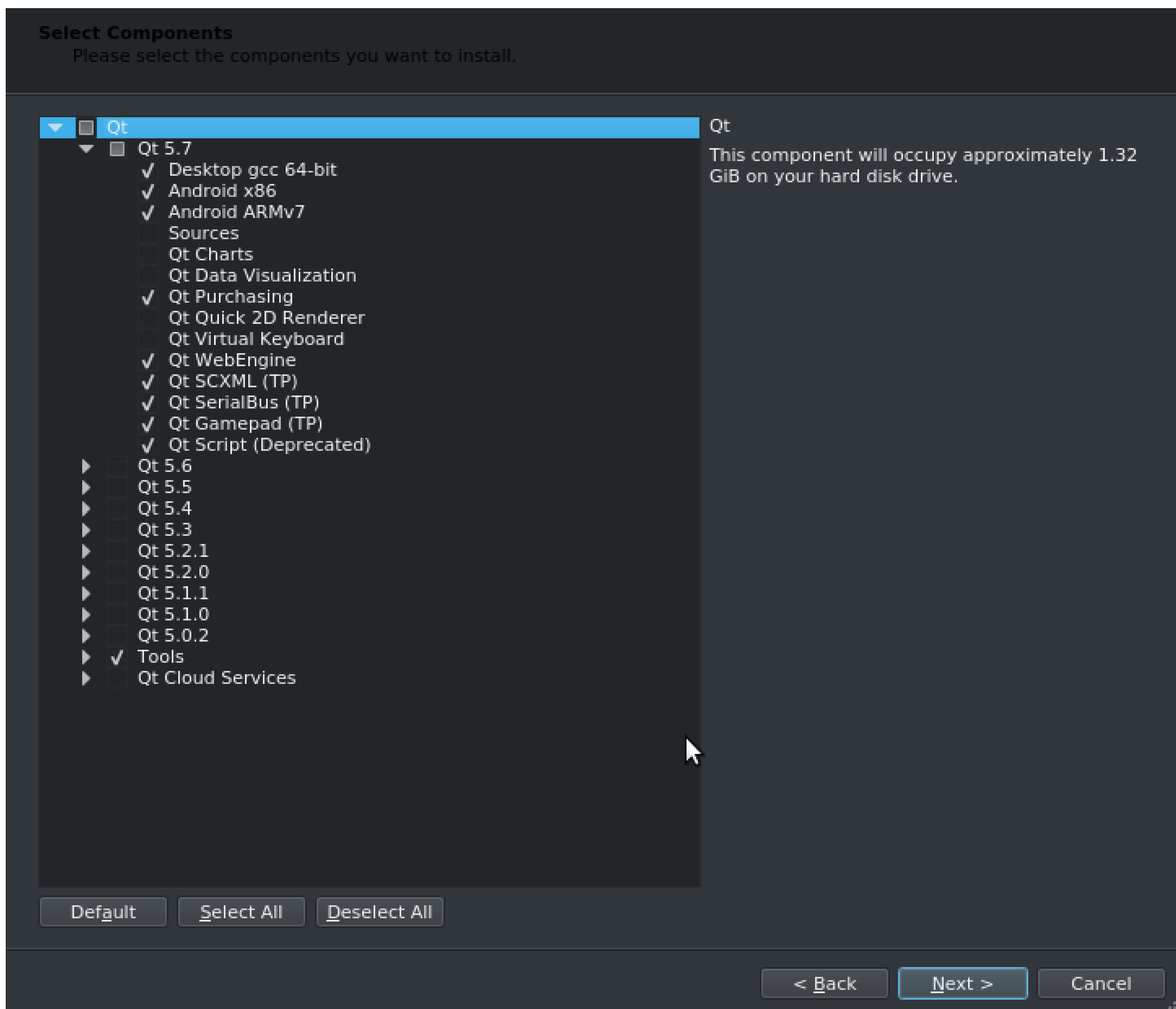
Inicie sesión con una cuenta Qt existente o cree una nueva:



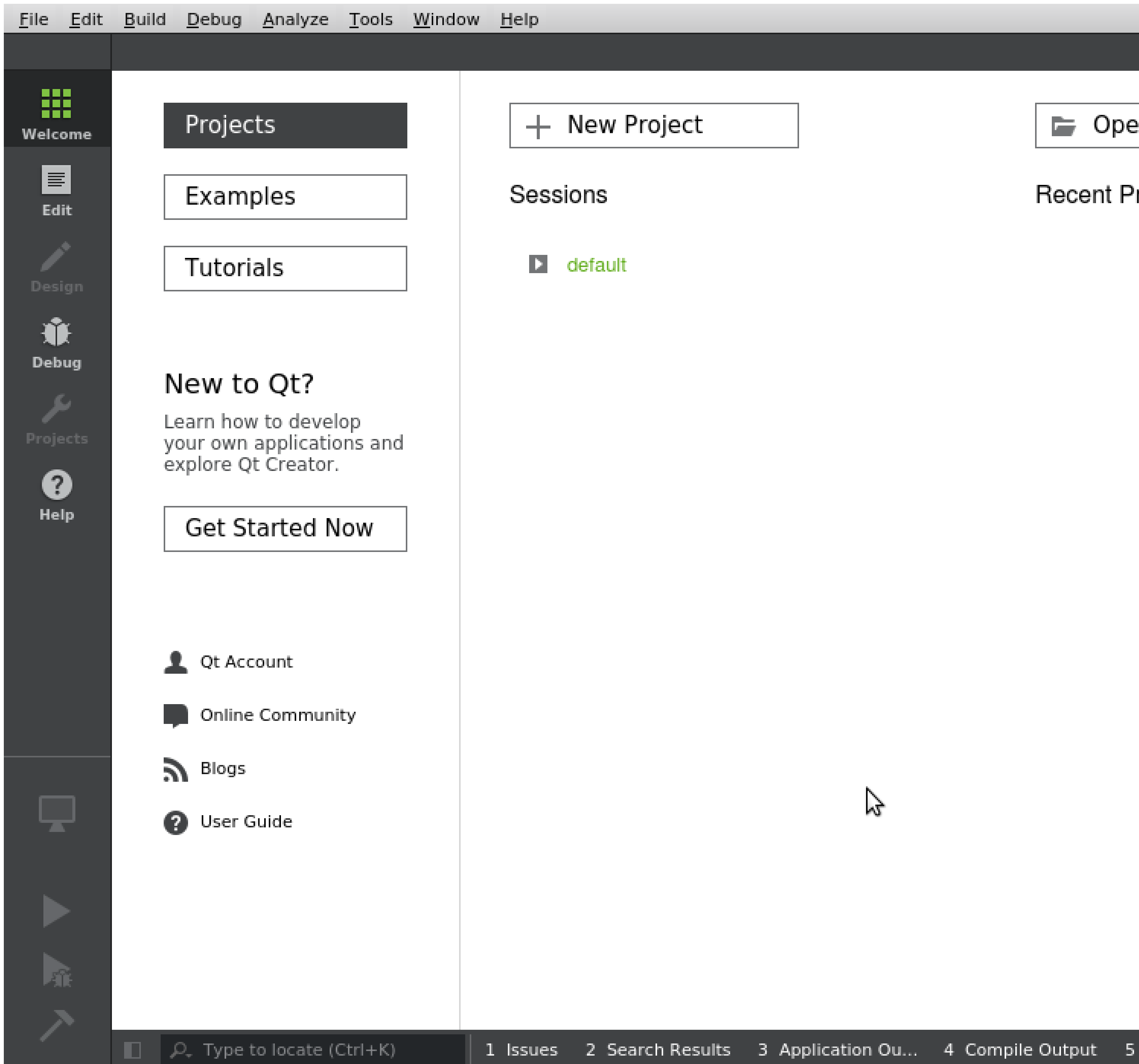
Seleccione una ruta para instalar las bibliotecas y herramientas Qt



Seleccione la versión de la biblioteca y las características que desea



Después de descargar y terminar la instalación, vaya al directorio de instalación de Qt e inicie Qt Creator o ejecútelo directamente desde la línea de comandos.



Hola Mundo

En este ejemplo, simplemente creamos y mostramos un botón en un marco de ventana en el escritorio. El pulsador tendrá la etiqueta `Hello world!`

Esto representa el programa Qt más simple posible.

En primer lugar necesitamos un archivo de proyecto:

helloworld.pro

```
QT       += core gui

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

```
TARGET = helloworld
TEMPLATE = app

SOURCES += main.cpp
```

- QT se utiliza para indicar qué bibliotecas (módulos Qt) se están utilizando en este proyecto. Ya que nuestra primera aplicación es una pequeña GUI, necesitaremos QtCore y QtGui. Como Qt5 separa los QtWidgets de QtGui, necesitamos agregar `greaterThan` línea mayor que la compilación con Qt5.
- TARGET es el nombre de la aplicación o la biblioteca.
- PLANTILLA describe el tipo de construir. Puede ser una aplicación (aplicación), una biblioteca (lib) o simplemente subdirectorios (subdirectorios).
- FUENTES es una lista de archivos de código fuente que se utilizarán al crear el proyecto.

También necesitamos el main.cpp que contiene una aplicación 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. Este objeto administra los recursos de toda la aplicación y es necesario para ejecutar cualquier programa Qt que tenga una GUI. Necesita argv y args porque Qt acepta algunos argumentos de línea de comando. Cuando se llama `a.exec()` se inicia el bucle de eventos Qt.
- Objeto QPushButton. El pulsador con la etiqueta `Hello world!` . La siguiente línea, `button.show()` , muestra el botón pulsador en la pantalla en su propio marco de ventana.

Finalmente, para ejecutar la aplicación, abra un símbolo del sistema e ingrese el directorio en el que tiene el archivo .cpp del programa. Escriba los siguientes comandos de shell para construir el programa.

```
qmake -project
qmake
make
```

Aplicación básica con QtCreator y QtDesigner.

QtCreator es, en este momento, la mejor herramienta para crear una aplicación Qt. En este

ejemplo, veremos cómo crear una aplicación Qt simple que administre un botón y escriba texto.

Para crear una nueva aplicación, haga clic en Archivo-> Nuevo archivo o proyecto:

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

https://riptutorial.com/es/home

que inicializan la interfaz de usuario.

Luego podemos crear MainWindow :: whenButtonIsClicked () en nuestra clase .cpp que podría cambiar el texto de la etiqueta de esa manera:

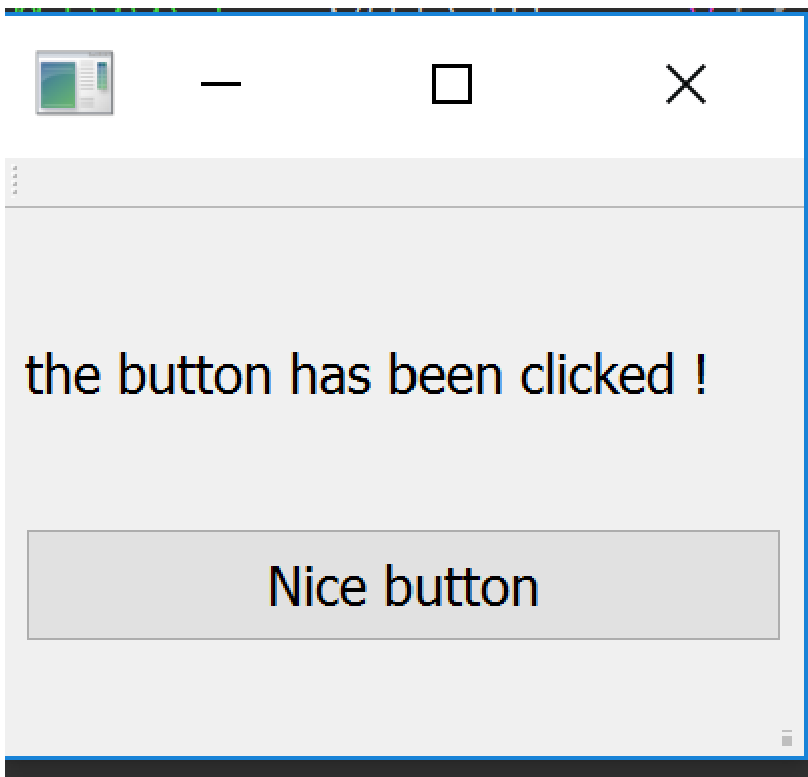
```
void MainWindow::whenButtonIsClicked()
{
    ui->label->setText("the button has been clicked !");
}
```

Y en nuestra mainwindow.h, necesitamos agregar:

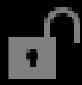
```
public slots:
    void whenButtonIsClicked();
```

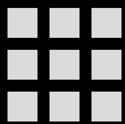
Las ranuras públicas significan que este método se puede llamar cuando se recibe una señal. Conecte el enlace de la señal cuando hacemos clic en el botón y un método para llamar.

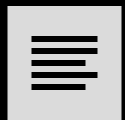
Así que ahora, si ejecutamos nuestra aplicación y hacemos clic en el botón, obtenemos:

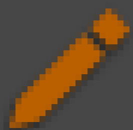


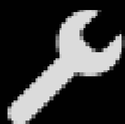
Lo que significa que nuestra conexión está funcionando. Pero con Qt Designer tenemos una forma aún más sencilla de hacerlo. Si desea hacer lo contrario, elimine la conexión para desconectar el botón (porque lo conectaremos de forma diferente), vuelva a mainwindow.ui y haga clic con el botón derecho en el botón. Haga clic en Ir a la ranura ..., seleccione clic () y presione Aceptar.

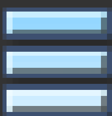


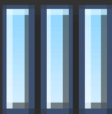


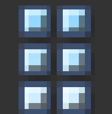


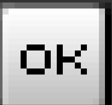


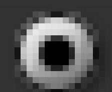












https://riptutorial.com/es/home

que es una clase increíble que puede convertir muchas cosas en muchas otras cosas. Así que a la izquierda agrega un int que aumenta cuando pulsamos el botón.

Así que el .h:

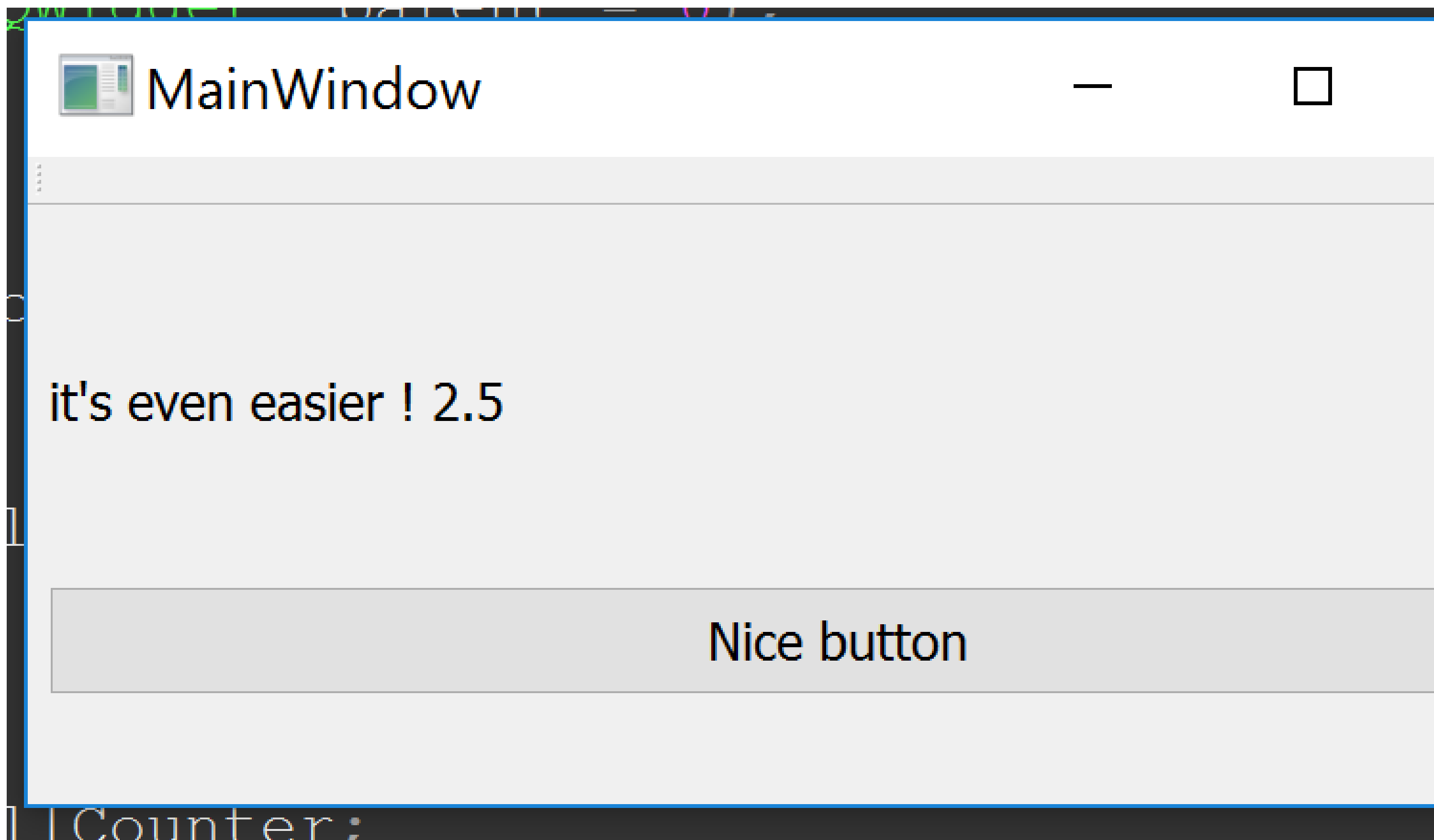
```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#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
```

El .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());
}
```

Y ahora, podemos guardar y correr de nuevo. Cada vez que hace clic en el botón, muestra "¡es aún más fácil!" Con el valor de _smallCounter. Entonces deberías tener algo como:



Este tutorial está hecho. Si desea obtener más información sobre Qt, veamos otros ejemplos y documentación de Qt en la documentación de StackOverflow o en la documentación de Qt.

Lea Empezando con Qt en línea: https://riptutorial.com/es/qt/topic/902/empezando-con-qt

Capítulo 2: Clases de Contenedores Qt

Observaciones

Qt proporciona sus propias clases de contenedor de plantillas. Todos ellos están compartidos implícitamente. Proporcionan dos tipos de iteradores (estilo Java y estilo STL).

Los contenedores secuenciales de Qt incluyen: QVector, QList, QLinkedList, QStack, QQueue.

Los contenedores asociativos Qt incluyen: QMap, QMultiMap, QHash, QMultiHash, QSet.

Examples

Uso de QStack

`QStack<T>` es una plantilla de clase Qt que proporciona pila. Su análogo en STL es `std::stack` . Es el último en entrar, primero en salir de la estructura (LIFO).

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

Saldrá: Tercero, Segundo, Primero.

`QStack` hereda de `QVector` por lo que su implementación es bastante diferente de STL. En STL `std::stack` se implementa como un envoltorio para escribir como argumento de plantilla (deque por defecto). Las operaciones principales son las mismas para `QStack` y para `std::stack` .

Uso de QVector

`QVector<T>` proporciona una clase de plantilla de matriz dinámica. Proporciona un mejor rendimiento en la mayoría de los casos que `QList<T>` por lo que debe ser la primera opción.

Se puede inicializar de varias maneras:

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

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

Lo último implica lista de inicialización.

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

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

Puedes obtener i-th elemento de vector de esta manera:

`v[i]` o `at[i]`

Asegúrese de que `i` sea una posición válida, incluso `at(i)` no se comprueba, esta es una diferencia de `std::vector` .

Uso de QLinkedList

En Qt, debe usar QLinkedList en caso de que necesite implementar una lista enlazada .

Es rápido de agregar, anteponer, insertar elementos en `QLinkedList` - O (1), pero la búsqueda de índices es más lenta que en `QList` o `QVector` - O (n). Esto es normal teniendo en cuenta que hay que recorrer los nodos para encontrar algo en la lista vinculada.

Tabla compexity algorítmico completo se puede encontrar aquí .

Solo para insertar algunos elementos en `QLinkedList` puede usar el operador `<<()` :

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

Para insertar elementos en medio de `QLinkedList` o modificar todos o algunos de sus elementos, puede usar iteradores de estilo Java o estilo STL. Aquí hay un ejemplo simple de cómo multiplicamos todos los elementos de `QLinkedList` por 2:

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

QList

La clase `QList` es una clase de plantilla que proporciona listas. Almacena elementos en una lista que proporciona un acceso rápido basado en índices e inserciones y eliminaciones basadas en índices.

Para insertar elementos en la lista, puede usar el `operator<<()` , `insert()` , `append()` o `prepend()` . Por ejemplo:

`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");
```

Para acceder al elemento en una posición de índice particular, puede usar el `operator[]()` o `at()` . `at()` puede ser más rápido que el `operator[]()` , nunca causa una copia profunda del contenedor y debería funcionar en tiempo constante. Ninguno de los dos hace la comprobación de argumentos. Ejemplos:

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

O

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

Para eliminar elementos, existen funciones como `removeAt()` , `takeAt()` , `takeFirst()` , `takeLast()` , `removeFirst()` , `removeLast()` o `removeOne()` . Ejemplos:

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");
```

Para encontrar todas las apariciones de un valor particular en una lista, puede usar `indexOf()` o

`lastIndexOf()` . Ejemplo:

`indexOf()`

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

Lea Clases de Contenedores Qt en línea: https://riptutorial.com/es/qt/topic/6303/clases-de-contenedores-qt

Capítulo 3: CMakeLists.txt para su proyecto Qt

Examples

CMakeLists.txt para Qt 5

Un archivo de proyecto CMake mínimo que utiliza Qt5 puede ser:

```
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
)
```

Se llama a `cmake_minimum_required` para establecer la versión mínima requerida para CMake. La versión mínima requerida para que este ejemplo funcione es `2.8.11` : las versiones anteriores de CMake necesitan un código adicional para que un objetivo use Qt.

Se llama a `find_package` para buscar una instalación de Qt5 con una versión dada (5.7.0 en el ejemplo) y componentes deseados: módulo principal en el ejemplo. Para obtener una lista de los módulos disponibles, consulte la documentación de Qt . Qt5 está marcado como `REQUIRED` en este proyecto. La ruta a la instalación puede indicarse estableciendo la variable `Qt5_DIR` .

`AUTOMOC` es un valor booleano que especifica si CMake manejará el preprocesador Qt `moc` automáticamente, es decir, sin tener que usar la macro `QT5_WRAP_CPP()` .

Otras variables "similares a AUTOMOC" son:

- `AUTOUIC` : un valor booleano que especifica si CMake manejará el generador de código Qt `uic` automáticamente, es decir, sin tener que usar la macro `QT5_WRAP_UI()` .
- `AUTORCC` : un valor booleano que especifica si CMake manejará el generador de código Qt `rcc` automáticamente, es decir, sin tener que usar la macro `QT5_ADD_RESOURCES()` .

Se llama a `add_executable` para crear un destino ejecutable a partir de los archivos de origen dados. El objetivo se vincula a los módulos de Qt enumerados con el comando

`target_link_libraries` . Desde CMake 2.8.11, `target_link_libraries` con los destinos importados de Qt manejan los parámetros del enlazador, así como también incluyen los directorios y las opciones del compilador.

Lea CMakeLists.txt para su proyecto Qt en línea:
https://riptutorial.com/es/qt/topic/1991/cmakelists-txt-para-su-proyecto-qt

Capítulo 4: Comunicación entre QML y C ++.

Introducción

Podemos usar QML para construir aplicaciones híbridas, ya que es mucho más fácil que C ++. Así que deberíamos saber cómo se comunican entre sí.

Examples

Llama a C ++ en QML

Registrar clases de C ++ en QML

En el lado de C ++, imagine que tenemos una clase llamada `QmlCppBridge` , que implementa un método llamado `printHello()` .

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

Queremos usarlo en el lado QML. Debemos registrar la clase llamando a `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");
```

En QML, usa el siguiente código para llamarlo:

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

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

Uso de `QQmlContext` para inyectar clases o variables de C ++ a QML

Todavía usamos la clase C ++ en el ejemplo anterior:

```
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);
```

En el lado QML:

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

> **Nota:** Este ejemplo se basa en Qt 5.7. No estoy seguro si se ajusta a versiones anteriores de Qt.

Llame a QML en C ++

Para llamar a las clases QML en C ++, debe establecer la propiedad objectName.

En tu Qml:

```
import QtQuick.Controls 2.0

Button {
    objectName: "buttonTest"
}
```

Luego, en su C ++, puede obtener el objeto con `QObject.FindChild<QObject*>(QString)`

Como eso:

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

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

Ahora tienes tu objeto QML en tu C ++. Pero eso podría parecer inútil ya que realmente no podemos obtener los componentes del objeto.

Sin embargo, podemos usarlo para enviar **señales** entre el QML y el C ++. Para hacer eso, necesita agregar una señal en su archivo QML como esa: `signal buttonClicked(string str)` . Una vez que creas esto, necesitas emitir la señal. Por ejemplo:

```
import QtQuick 2.0
import QtQuick.Controls 2.1

    Button {
        id: buttonTest
        objectName: "buttonTest"

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

```
    }
```

Aquí tenemos nuestro botón qml. Cuando hacemos clic en él, va al método **onClicked** (un método base para los botones que se llama cuando se presiona el botón). Luego usamos la identificación del botón y el nombre de la señal para emitir la señal.

Y en nuestro cpp, necesitamos conectar la señal con una ranura. como eso:

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();
}
```

Como puede ver, obtenemos nuestro botón qml con `findChild` como antes y conectamos la señal a un administrador de botones que es una clase creada y que se parece a eso. 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

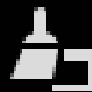
```
#include "ButtonManager.h"
```

```
#include <QDebug>

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

}

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

Entonces, cuando se reciba la señal, llamará al método `onButtonClicked` que escribirá `"button: clicked !"`

salida:

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/es/qt/topic/8735/comunicacion-entre-qml-y-c-plusplus-

Capítulo 5: Construye QtWebEngine desde la fuente

Introducción

A veces necesitamos compilar QtWebEngine desde la fuente por alguna razón, como para el soporte de mp3.

Examples

Construir en Windows

Requerimientos

- Windows 10, **configure la configuración regional de su sistema en inglés** , de lo contrario puede haber errores
- Visual Studio 2013 o 2015
- Código fuente de QtWebEngine 5.7 (se puede descargar desde aquí)
- Qt 5.7 instale la versión, instálela y agregue la carpeta `qmake.exe` a la ruta del sistema
- Python 2, agregue la carpeta `python.exe` a la ruta del sistema
- Git, agrega la carpeta `git.exe` a la ruta del sistema
- gperf, agregue la carpeta `gperf.exe` a la ruta del sistema
- flex-bison, agregue la carpeta `win_bison.exe` a la ruta del sistema y cambie el nombre a `bison.exe`

  Nota: no probé las versiones de Visual Studio, todas las versiones de Qt ... Tomemos un ejemplo aquí, otras versiones deberían ser casi iguales.

Pasos para construir

1. Descomprimir el código fuente a una carpeta, llamémoslo `ROOT`
2. Abra la `Developer Command Prompt for VS2013` y vaya a la carpeta `ROOT`
3. Ejecute `qmake WEBENGINE_CONFIG+=use_proprietary_codecs qtwebengine.pro` . Añadimos esta bandera para habilitar el soporte de mp3.
4. Corre `nmake`

   Nota: MtWebEngine no admite Mp3 de forma predeterminada, debido a un problema de licencia. Asegúrese de obtener una licencia para el códec que agregó.

Lea Construye QtWebEngine desde la fuente en línea: https://riptutorial.com/es/qt/topic/8718/construye-qtwebengine-desde-la-fuente

Capítulo 6: Despliegue de aplicaciones Qt

Examples

Despliegue en windows

Qt proporciona una herramienta de implementación para Windows: `windeployqt` . La herramienta inspecciona un ejecutable de la aplicación Qt por sus dependencias a los módulos Qt y crea un directorio de implementación con los archivos Qt necesarios para ejecutar el ejecutable inspeccionado. Un posible guión puede parecer:

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

Se llama al comando `set` para agregar el directorio `bin` de Qt a la `PATH` entorno `PATH` . `windeployqt` se llama entonces:

- La ruta al directorio de implementación recibe un argumento opcional dado con el parámetro `--dir` (la ruta predeterminada donde se llama `windeployqt` es la `windeployqt`).
- La ruta al ejecutable que se va a inspeccionar se proporciona como último argumento.

El directorio de implementación se puede agrupar con el ejecutable.

NOTA:

Si está utilizando Qt5.7.0 precompilado con vs2013 en Windows *(no estoy seguro de que todas las versiones tengan este problema)* , existe la posibilidad de que necesite copiar manualmente `<QTDIR>\5.7\msvc2015\qml` a su directorio bin de tu programa De lo contrario, el programa se cerrará automáticamente después del inicio.

Véase también la documentación de Qt .

Integración con CMake

Es posible ejecutar `windeployqt` y `macdeployqt` desde CMake, pero primero se debe encontrar la ruta a los ejecutables:

```
# 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}")
```

Para que `windeployqt` encuentre las bibliotecas Qt en su ubicación instalada, la carpeta debe agregarse a `%PATH%` . Para hacer esto para un objetivo llamado `myapp` después de ser construido:

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

Para ejecutar `macdeployqt` en un paquete, se haría de esta manera:

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

Implementación en Mac

Qt ofrece una herramienta de implementación para Mac: la herramienta de implementación de Mac.

La herramienta de implementación de Mac se puede encontrar en `QTDIR/bin/macdeployqt` . Está diseñado para automatizar el proceso de creación de un paquete de aplicaciones desplegable que contiene las bibliotecas Qt como marcos privados.

La herramienta de implementación de mac también implementa los complementos de Qt, de acuerdo con las siguientes reglas (a menos que **se use la opción -no-complementos**):

- El complemento de la plataforma siempre está desplegado.
- Las versiones de depuración de los complementos no están implementadas.
- Los complementos de diseño no están desplegados.
- Los complementos de formato de imagen siempre se implementan.
- El complemento de soporte de impresión siempre se implementa.
- Los complementos del controlador SQL se implementan si la aplicación utiliza el módulo Qt SQL.
- Los complementos de script se implementan si la aplicación utiliza el módulo Qt Script.
- El complemento del icono de SVG se implementa si la aplicación utiliza el módulo Qt SVG.
- El complemento de accesibilidad siempre está desplegado.

Para incluir una biblioteca de terceros en el paquete de la aplicación, copie la biblioteca en el paquete manualmente, después de crear el paquete.

Para usar la herramienta `macdeployqt` puede abrir el terminal y escribir:

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

El archivo de la aplicación ahora contendrá todas las bibliotecas Qt utilizadas como marcos privados.

`macdeployqt` también soporta las siguientes opciones

Opción	Descripción
-verbose = <0-3>	0 = sin salida, 1 = error / advertencia (predeterminado), 2 = normal, 3 = depuración
-no-plugins	Omitir despliegue de plugin
-dmg	Crear una imagen de disco .dmg
-no-tira	No corras 'strip' en los binarios
-use-debug-libs	Implementar con versiones de debug de marcos y complementos (implica -no-strip)
-ejecutable =	Deje que el ejecutable dado también use los frameworks desplegados.
-qmldir =	Implemente las importaciones utilizadas por los archivos .qml en la ruta dada

Las informaciones detalladas se pueden encontrar en la documentación de Qt.

Despliegue en linux

Hay una herramienta de implementación para Linux en GitHub . Aunque no es perfecto, está enlazado desde la wiki de Qt. Se basa conceptualmente en la herramienta de implementación de Qt Mac y funciona de manera similar al proporcionar una AppImage .

Dado que un archivo de escritorio se debe proporcionar con una AppImage, `linuxdeployqt` puede usar eso para determinar los parámetros de la compilación.

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

Donde el archivo de escritorio especifica el ejecutable que se ejecutará (con `EXEC=`), el nombre de la aplicación y un icono.

Lea Despliegue de aplicaciones Qt en línea: https://riptutorial.com/es/qt/topic/5857/despliegue-de-aplicaciones-qt

Capítulo 7: Encabezado en QListView

Introducción

El widget QListView es parte de los mecanismos de programación Modelo / Vista de Qt. Básicamente, permite mostrar elementos almacenados en un modelo bajo la forma de una lista. En este tema no profundizaremos en los mecanismos Modelo / Vista de Qt, sino que nos centraremos en el aspecto gráfico de un widget de Vista: el QListView, y especialmente cómo agregar un encabezado sobre este objeto a través del uso de QPaintEvent objeto.

Examples

Declaración personalizada QListView

```
/*!
 * \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. */
};
```

Implementación del QListView personalizado.

```
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();
}
```

Caso de uso: declaración de MainWindow

```
class MainMenuListView;

class MainWindow : public QMainWindow
{
    Q_OBJECT

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

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

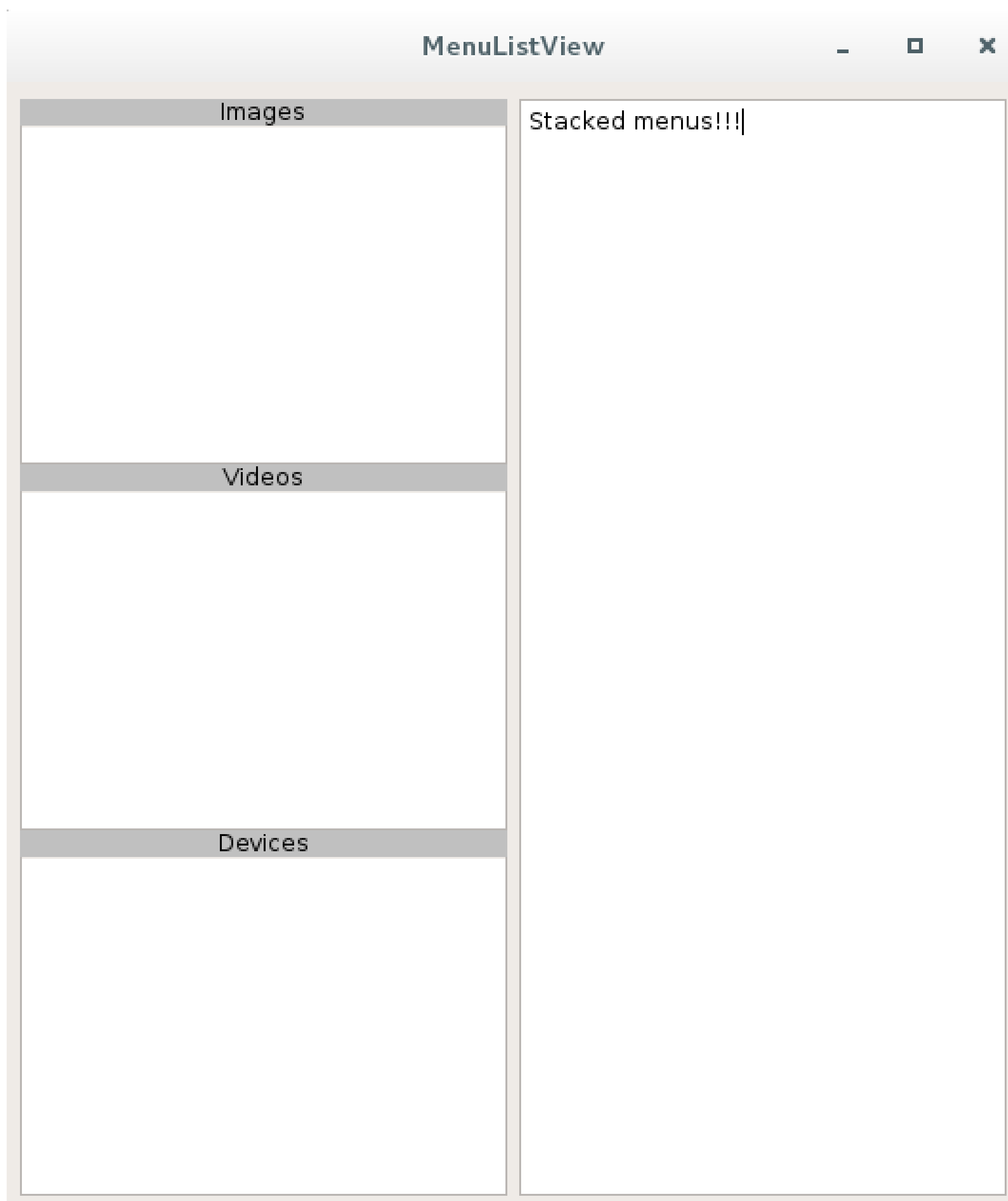
Caso de uso: Implementación

```
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() {}
```

Caso de uso: Salida de muestra

Aquí hay una salida de muestra:



Como puede ver arriba, puede ser útil para crear menús apilados. Tenga en cuenta que esta muestra es trivial. Los dos widgets tienen las mismas restricciones de tamaño.

Lea Encabezado en QListView en línea: https://riptutorial.com/es/qt/topic/9382/encabezado-en-qlistview

Capítulo 8: Errores comunes

Examples

Uso de Qt: DirectConnection cuando el objeto receptor no recibe señal

Algunas veces ves que se emite una señal en el subproceso del remitente pero la ranura conectada no llama (en otras palabras, no recibe señal), has preguntado al respecto y finalmente obtuviste el tipo de conexión Qt :: DirectConnection que lo solucionaría. Así que el problema encontrado y todo está bien.

Pero, en general, es una mala idea usar Qt: DirectConnection hasta que realmente sepa qué es esto y no haya otra manera. Expliquémoslo más. Cada subproceso creado por Qt (incluido el subproceso principal y los nuevos subprocesos creados por QThread) tiene un bucle de eventos, el bucle de eventos es responsable de recibir señales y llamar a ranuras apropiadas en su hilo. En general, la ejecución de una operación de bloqueo dentro de una ranura es una mala práctica, ya que bloquea el bucle de eventos de esos hilos para que no se llame a ninguna otra ranura.

Si bloquea un bucle de eventos (al realizar una operación que requiere mucho tiempo o bloqueo), no recibirá eventos en ese hilo hasta que se desbloquee el bucle de eventos. Si la operación de bloqueo bloquea el ciclo de eventos para siempre (por ejemplo, ocupado mientras está ocupado), las ranuras nunca podrían llamarse.

En esta situación, puede establecer el tipo de conexión en conectarse a Qt :: DirectConnection, ahora se llamarán las ranuras, incluso se bloquea el bucle de eventos. Entonces, ¿cómo esto podría hacer que todo se rompiera? En Qt :: DirectConnection, las ranuras se llamarán en subprocesos emiter, y no en subprocesos de receptor, ya que pueden interrumpir las sincronizaciones de datos y encontrar otros problemas. Nunca use Qt :: DirectConnection a menos que sepa qué está haciendo. Si su problema se resolverá mediante el uso de Qt :: DirectConnection, debe tener cuidado y observar su código y descubrir por qué se bloquea el bucle de eventos. No es una buena idea bloquear el bucle de eventos y no se recomienda en Qt.

Aquí hay un pequeño ejemplo que muestra el problema, ya que se puede ver que el bloqueo no bloqueado se llamaría incluso el bucle de evento bloqueado de bloqueo de ranura con while (1) que indica una codificación incorrecta

```
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();
}
```

Lea Errores comunes en línea: https://riptutorial.com/es/qt/topic/8238/errores-comunes

Capítulo 9: Intercambio implícito

Observaciones

Los iteradores de estilo STL en Qt Container pueden tener algún efecto secundario negativo debido a la participación implícita. Se recomienda evitar la copia de un contenedor Qt mientras tenga iteradores activos en ellos.

```
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

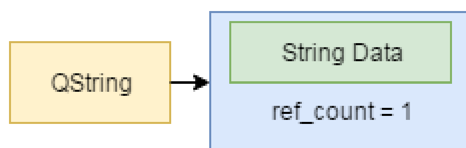
Concepto basico

Varios Objetos y Contenedores de Qt utilizan un concepto de **intercambio implícito de** calles, al que también se puede hacer referencia como **copia sobre escritura** .

El uso compartido implícito significa que las clases que usan este concepto comparten los mismos datos en la inicialización.

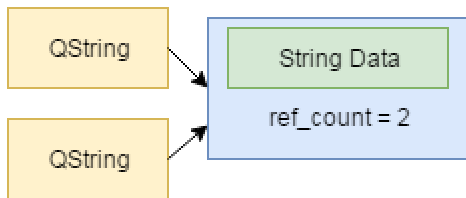
Una de estas clases para usar el concepto es QString.

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



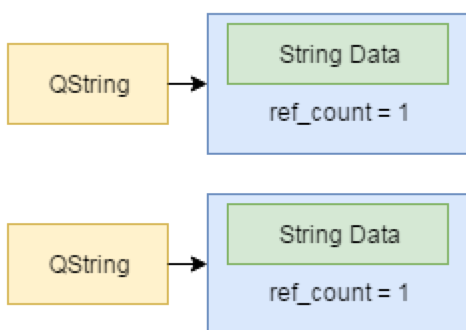
Este es un modelo simplificado de una QString. Internamente tiene un bloque de memoria, con los datos reales de la cadena y un contador de referencia.

```
QString s2 = s1;
```



Si ahora copiamos esta `QString` ambos objetos apuntarán internamente al mismo contenido, evitando así operaciones de copia innecesarias. Tenga en cuenta cómo el recuento de referencia también se elevó. Entonces, en caso de que la primera cadena se elimine, los datos compartidos todavía saben que otra `QString` hace referencia a `QString` .

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



Ahora, cuando la `QString` se modifica realmente, el objeto se "separa" del bloque de memoria, copiando su contenido y modificando el contenido.

Lea Intercambio implícito en línea: https://riptutorial.com/es/qt/topic/6801/intercambio-implicito

Capítulo 10: Modelo / Vista

Examples

Una tabla simple de solo lectura para ver datos de un modelo

Este es un ejemplo simple para mostrar datos de solo lectura que son de naturaleza tabular utilizando el Modelo / Marco de Vista de Qt. Específicamente, se usan los `Qt Objects` QAbstractTableModel (subclasificados en este ejemplo) y QTableView .

Las implementaciones de los métodos rowCount () , columnCount () , data () y headerData () son necesarias para proporcionar al objeto `QTableView` un medio para obtener información sobre los datos contenidos en el objeto `QAbstractTableModel` .

El método `populateData()` se agregó a este ejemplo para proporcionar un medio para rellenar el objeto `QAbstractTableModel` con datos de alguna fuente arbitraria.

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
```

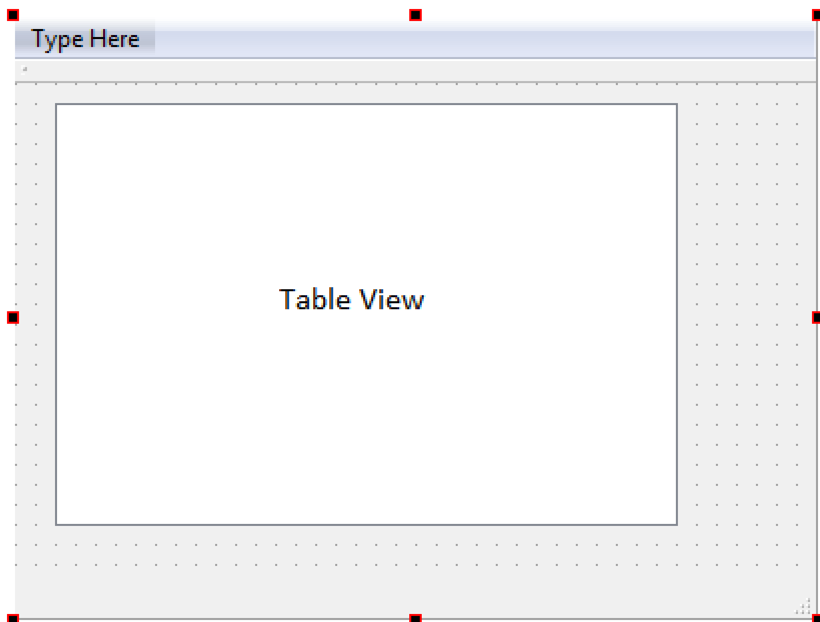
```
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;

};

#endif // MAINWINDOW_H
```

mainwindow.cpp

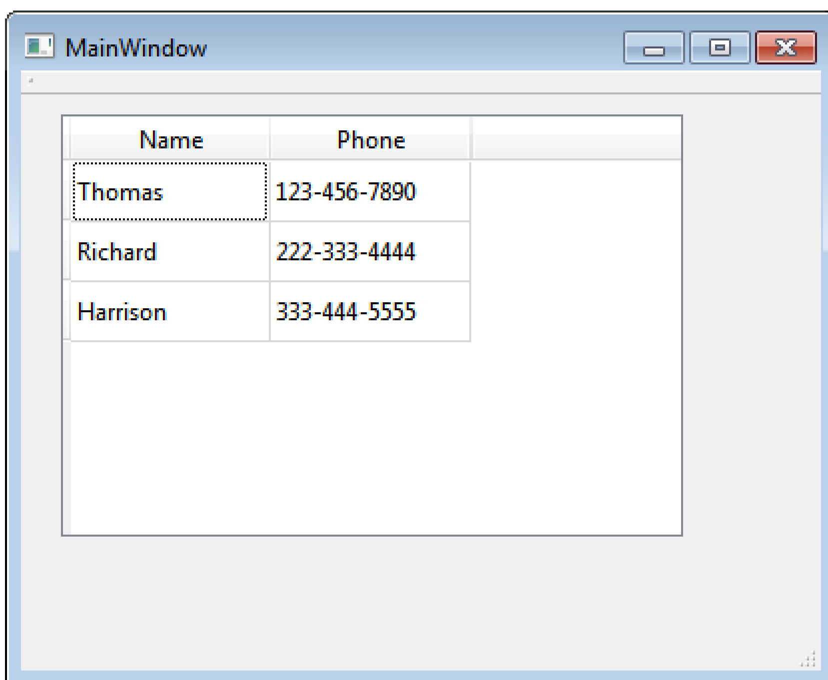
```
#include "mainwindow.h"
#include "ui_mainwindow.h"

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();
}
```

Usando `Qt Creator/Design` , coloque un objeto de `Table View` , denominado **tableView** en este ejemplo, en la **ventana principal** :



El programa resultante se muestra como:



Un modelo de árbol simple.

QModelIndex no sabe realmente sobre sus índices padre / hijo, solo contiene una **fila** , una **columna** y un **puntero** , y es responsabilidad del modelo utilizar estos datos para proporcionar información sobre las relaciones de un índice. Por lo tanto, el modelo necesita realizar muchas conversiones desde el `void*` almacenado dentro del `QModelIndex` a un tipo de datos interno y viceversa.

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();
}
```

Lea Modelo / Vista en línea: https://riptutorial.com/es/qt/topic/3938/modelo---vista

Capítulo 11: Multimedia

Observaciones

Qt Multimedia es un módulo que proporciona manejo de multimedia (audio, video) y también de cámara y radio.

Sin embargo, los archivos compatibles de QMediaPlayer dependen de la plataforma. De hecho, en Windows, QMediaPlayer usa DirectShow, en Linux, usa GStreamer. Entonces, dependiendo de la plataforma, algunos archivos pueden funcionar en Linux pero no en Windows o lo contrario.

Examples

Reproducción de video en Qt 5

Vamos a crear un reproductor de video muy simple usando el módulo QtMultimedia de Qt 5.

En el archivo .pro de su aplicación necesitará las siguientes líneas:

```
QT += multimedia multimediawidgets
```

Tenga en cuenta que los `multimediawidgets` son necesarios para el uso de `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();
```

Eso es todo: después de iniciar la aplicación (si los códecs necesarios están instalados en el sistema), se iniciará la reproducción del archivo de video.

De la misma manera puede reproducir videos desde URL en Internet, no solo archivos locales.

Reproducción de audio en Qt5

Como se trata de un audio, no necesitamos un QVideoWidget. Así que podemos hacer:

```
_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();
```

en el .h:

```
QMediaPlayer *_player;
```

Esto abrirá un diálogo donde puede elegir su música y la reproducirá.

Lea Multimedia en línea: https://riptutorial.com/es/qt/topic/7675/multimedia

Capítulo 12: QDialogs

Observaciones

La clase QDialog es la clase **base** de ventanas de diálogo. Una ventana de diálogo es una ventana de nivel superior utilizada principalmente para tareas a corto plazo y comunicaciones breves con el usuario. QDialogs puede ser **modal** o **modeless** .

Tenga en cuenta que QDialog (y cualquier otro widget que tenga el tipo Qt :: Dialog) utiliza el widget principal de forma ligeramente diferente a otras clases en Qt. Un diálogo **siempre** es **un widget de nivel superior** , pero si **tiene un padre, su ubicación predeterminada se centra en la parte superior del widget de nivel superior del padre** (si no es el nivel superior en sí). También compartirá la entrada de la barra de tareas de los padres.

Un diálogo **modal** es un diálogo que bloquea la entrada a otras ventanas visibles en la misma aplicación. Los diálogos que se utilizan para solicitar un nombre de archivo al usuario o que se usan para configurar las preferencias de la aplicación suelen ser modales. Los diálogos pueden ser **modal de aplicación** (el predeterminado) o **modal de ventana** .

La forma más común de mostrar un diálogo modal es llamar a su función exec (). Cuando el usuario cierra el cuadro de diálogo, exec () proporcionará un valor de retorno útil.

Un diálogo sin **modelo** es un diálogo que funciona independientemente de otras ventanas en la misma aplicación. Los cuadros de diálogo sin modo se muestran mediante show (), que devuelve el control a la persona que llama inmediatamente.

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>
```

Lea QDialogs en línea: https://riptutorial.com/es/qt/topic/7819/qdialogs

Capítulo 13: Qgraphics

Examples

Panorámica, zoom y rotación con QGraphicsView

`QGraphics` se puede usar para organizar escenas complicadas de objetos visuales en un marco que facilita su manejo.

Hay tres tipos principales de objetos utilizados en este marco QGraphicsView , QGraphicsScene y QGraphicsItems . QGraphicsItems son los elementos visuales básicos que existen en la escena.

Hay muchos tipos que están preconstruidos y se pueden usar, como elipses , líneas , rutas , pixmaps , polígonos , rectángulos y texto .

También puede hacer sus propios artículos heredando `QGraphicsItem` . Luego, estos elementos se colocan en un `QGraphicsScene` que es básicamente el mundo que planea ver. Los elementos pueden moverse dentro de la escena, lo cual es como hacer que se muevan en el mundo que estás mirando. El posicionamiento y la orientación de los elementos se manejan mediante matrices de transformación denominadas QTransforms . Qt tiene buenas funciones integradas, por lo que generalmente no necesita trabajar con `QTransforms` directamente, en lugar de eso, llama a funciones como rotar o escalar, que crean las transformaciones adecuadas para usted. Luego, la escena se ve según la perspectiva definida en `QGraphicsView` (de nuevo con `QTransforms`), que es la pieza que pondría en un widget en su interfaz de usuario.

En el siguiente ejemplo, hay una escena muy simple con un solo elemento (un mapa de píxeles), que se coloca en una escena y se muestra en una vista. Al activar el indicador `DragMode` , la escena se puede desplazar con el mouse y con las funciones de escala y rotación se puede ampliar y reducir con el desplazamiento del mouse y rotar con las teclas de flecha.

Si desea ejecutar este ejemplo, cree una instancia de Vista que se mostrará y cree un archivo de recursos con el prefijo / imágenes que contengan una imagen 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);
  }
};
```

Lea Qgraphics en línea: https://riptutorial.com/es/qt/topic/7539/qgraphics

Capítulo 14: qmake

Examples

Perfil por defecto.

qmake es una herramienta de automatización de compilación, que se envía con el marco *Qt* . Hace un trabajo similar a herramientas como *CMake* o *GNU Autotools* , pero está diseñado para ser utilizado específicamente con *Qt* . Como tal, está bien integrado con el ecosistema *Qt* , especialmente con *Qt Creator* IDE.

Si inicia *Qt Creator* y selecciona `File -> New File or Project -> Application -> Qt Widgets` , *Qt Creator* generará un esquema de proyecto para usted junto con un archivo "pro". *Qmake* procesa el archivo "pro" para generar archivos, que a su vez son procesados por sistemas de compilación subyacentes (por ejemplo, *GNU Make* o *nmake*).

Si nombró su proyecto "myapp", entonces aparecerá el archivo "myapp.pro". Así es como se ve ese archivo predeterminado, con comentarios, que describen cada variable *qmake* , agregada.

```
# 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
```

Preservar la estructura del directorio de origen en una opción de compilación (no documentada "object_parallel_to_source").

Si desea organizar su proyecto manteniendo los archivos de origen en diferentes subdirectorios, debe saber que durante una compilación *qmake* no conservará esta estructura de directorios y mantendrá todos los archivos ".o" en un solo directorio de compilación. Esto puede ser un problema si tuvo nombres de archivos en conflicto en diferentes directorios como los siguientes.

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

Ahora *qmake* decidirá crear dos archivos "file1.o" en un directorio de compilación, haciendo que uno de ellos sea sobrescrito por otro. El buld fallará. Para evitar esto, puede agregar `CONFIG += object_parallel_to_source` opción de configuración `CONFIG += object_parallel_to_source` a su archivo "pro". Esto le dirá a *qmake* que genere archivos de compilación que preserven la estructura del directorio de origen. De esta manera, su directorio de compilación reflejará la estructura del directorio de origen y los archivos de objetos se crearán en subdirectorios separados.

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

Ejemplo completo.

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

CONFIG += object_parallel_to_source

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

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

Tenga en cuenta que la opción `CONFIG` `object_parallel_to_source` **no** está **documentada oficialmente** .

Ejemplo simple (Linux)

Ventana.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.
```

Línea de comando

```
# 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.
```

Ejemplo de SUBDIRS

La capacidad SUBDIRS de qmake se puede utilizar para compilar un conjunto de bibliotecas, cada una de las cuales depende de otra. El siguiente ejemplo está ligeramente enrevesado para mostrar variaciones con la habilidad SUBDIRS.

Estructura de directorios

Algunos de los siguientes archivos se omitirán en aras de la brevedad. Se puede asumir que son el formato como ejemplos no subdirigidos.

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

proyecto.pro

Este es el archivo principal que habilita el ejemplo. Este es también el archivo al que se llamaría con qmake en la línea de comando (ver más abajo).

```
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
```

(*) Consulte la documentación de referencia para las otras opciones para un destino de subdirectorios.

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
```

logic / 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
```

construir.pro

```
TEMPLATE = app

SOURCES += main.cpp

LIBS += -Llogic -Lgui -llogic -lgui

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

Línea de comando

```
# 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.
```

Ejemplo de biblioteca

Un ejemplo simple para crear una biblioteca (en lugar de un ejecutable, que es el valor predeterminado). `TEMPLATE` variable `TEMPLATE` especifica el tipo de proyecto que está realizando. `lib` opción `lib` permite a makefile construir una biblioteca.

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
```

Cuando está construyendo una biblioteca, puede agregar opciones `dll` (predeterminado), `staticlib` o `plugin` a `CONFIG` .

Creando un archivo de proyecto a partir de código existente

Si tiene un directorio con archivos de origen existentes, puede usar `qmake` con la opción `-project`

para crear un archivo de proyecto.

Supongamos que la carpeta *MyProgram* contiene los siguientes archivos:

- main.cpp
- foo.h
- foo.cpp
- bar.h
- bar.cpp
- subdir / foobar.h
- subdir / foobar.cpp

Entonces llamando

```
qmake -project
```

Se crea un archivo *MyProgram.pro* con el siguiente contenido:

```
######################################################################
# 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
```

El código se puede construir como se describe en este ejemplo simple .

Lea qmake en línea: https://riptutorial.com/es/qt/topic/4438/qmake

Capítulo 15: QObject

Observaciones

`QObject` clase `QObject` es la clase base para todos los objetos Qt.

Examples

Ejemplo de QObject

`Q_OBJECT` macro `Q_OBJECT` aparece en la sección privada de una clase. `Q_OBJECT` requiere que la clase sea subclase de `QObject` . Esta macro es necesaria para que la clase declare sus señales / ranuras y use el sistema de metaobjetos Qt.

Si Meta Object Compiler (MOC) encuentra la clase con `Q_OBJECT` , la procesa y genera el archivo fuente de C ++ que contiene el código fuente del objeto meta.

Este es el ejemplo del encabezado de clase con `Q_OBJECT` y señal / ranuras:

```
#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)
```

Una funcionalidad que se agrega derivando de `QObject` y utilizando la macro `Q_OBJECT` es la capacidad de usar `qobject_cast` .

Ejemplo:

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

```
};

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

Para verificar si `obj` es un tipo `myObject` y para convertirlo en C ++, generalmente puede usar un `dynamic_cast` . Esto depende de tener habilitado RTTI durante la compilación.

La macro Q_OBJECT en las otras manos genera las verificaciones de conversión y el código que se puede usar en qobject_cast.

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

Esto no depende de RTTI. Y también le permite verter límites de bibliotecas dinámicas (a través de interfaces / complementos Qt).

QObject vida y propiedad

Los QObjects vienen con su propio concepto alternativo de vida en comparación con los punteros originales, únicos o compartidos de C ++.

QObjects tiene la posibilidad de construir un objecttree declarando las relaciones padre / hijo.

La forma más sencilla de declarar esta relación es pasar el objeto principal en el constructor. Como alternativa, puede establecer manualmente el padre de un `QObject` llamando a `setParent` . Esta es la única dirección para declarar esta relación. No puede agregar un niño a una clase de padres, sino solo al revés.

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

Cuando el `parent` ahora se elimina en la pila, el `child` también se eliminará.

Cuando eliminamos un objeto `QObject` , se "anulará el registro" del objeto principal;

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

Lo mismo se aplica a las variables de pila:

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

`child` se eliminará antes que el `parent` durante la retirada de la pila y se anulará el registro de su padre.

Nota: Se puede llamar manualmente `setParent` con un orden inverso de la declaración que **se** romperá la destrucción automática.

Lea QObject en línea: https://riptutorial.com/es/qt/topic/6304/qobject

Capítulo 16: Qt - Tratar con bases de datos

Observaciones

- Necesitará el complemento Qt SQL correspondiente al tipo dado a `QSqlDatabase::addDatabase`
- Si no tiene el complemento SQL requerido, Qt le advertirá que no puede encontrar el controlador solicitado
- Si no tiene el complemento SQL requerido, tendrá que compilarlo desde la fuente Qt

Examples

Usando una base de datos en Qt

En el archivo Project.pro agregamos:

```
CONFIG += sql
```

en MainWindow.h escribimos:

```
#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;
};
```

Ahora en 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 - Tratar con bases de datos Sqlite

En el archivo Project.pro agregamos: `CONFIG += sql`

en MainWindow.h escribimos:

```
#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;
};
```

Ahora en 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 - Tratar con bases de datos ODBC

En el archivo Project.pro agregamos: `CONFIG += sql`

en MainWindow.h escribimos:

```
#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;
};
```

Ahora en 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 - Tratar con las bases de datos Sqlite en memoria

En el archivo Project.pro agregamos: `CONFIG += sql`

en MainWindow.h escribimos:

```
#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;
};
```

Ahora en 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;
}
```

Eliminar la conexión de la base de datos correctamente

Si queremos eliminar alguna conexión de base de datos de la lista de conexiones de base de datos. necesitamos usar `QSqlDatabase::removeDatabase()` , sin embargo, es una función estática y la forma en que funciona es un poco cableada.

```
// 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 forma correcta en que el documento Qt nos sugiere es a continuación.

```
  {
      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
```

Lea Qt - Tratar con bases de datos en línea: https://riptutorial.com/es/qt/topic/1993/qt---tratar-con-

bases-de-datos

Capítulo 17: QTimer

Observaciones

QTimer también se puede usar para solicitar que una función se ejecute tan pronto como el bucle de eventos haya procesado todos los demás eventos pendientes. Para ello, utiliza un intervalo 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

Ejemplo simple

El siguiente ejemplo muestra cómo usar un `QTimer` para llamar a una ranura cada 1 segundo.

En el ejemplo, usamos una `QProgressBar` para actualizar su valor y comprobar que el temporizador funciona correctamente.

main.cpp

```
#include <QApplication>

#include "timer.h"

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

    Timer timer;
    timer.show();

    return app.exec();
}
```

temporizador.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
```

Temporizador Singleshot con función Lambda como ranura

Si se requiere un temporizador de disparo único, es bastante práctico tener la ranura como función lambda en el lugar donde se declara el temporizador:

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

Debido a este error (QTBUG-26406) , esto solo es posible desde Qt5.4.

En versiones anteriores de Qt5 se debe hacer con más código de placa de caldera:

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

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

Usando QTimer para ejecutar código en el hilo 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));
}
```

Esto es útil cuando necesita actualizar un elemento UI desde un hilo. Tenga en cuenta toda la vida de las referencias de devolución de llamada.

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

El mismo código podría adaptarse para ejecutar código en cualquier subproceso que ejecute el bucle de eventos Qt, implementando así un mecanismo de envío simple.

Uso básico

`QTimer` agrega la funcionalidad para tener una función / ranura específica llamada después de un cierto intervalo (repetidamente o solo una vez).

El `QTimer` tanto, permite que una aplicación GUI "verifique" las cosas regularmente o maneje los tiempos de espera **sin** tener que iniciar manualmente un subproceso adicional para esto y tenga cuidado con las condiciones de la carrera, ya que el temporizador se manejará en el bucle del evento principal.

Un temporizador se puede utilizar simplemente así:

```
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
```

El temporizador dispara la señal de `timeout` cuando termina el tiempo y esto se llamará en el bucle del evento principal.

QTimer :: singleShot uso simple

El **QTimer :: singleShot** se utiliza para llamar a una ranura / lambda de **forma asíncrona** después de n ms.

La sintaxis básica es:

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

con **myTime** el tiempo en ms, **myObject** el objeto que contiene el método y **myMethodInMyObject** la ranura para llamar

Así, por ejemplo, si desea tener un temporizador que escriba una línea de depuración "¡hola!" cada 5 segundos:

.cpp

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

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

.S.S

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

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

Lea QTimer en línea: https://riptutorial.com/es/qt/topic/4309/qtimer

Capítulo 18: Red qt

Introducción

Qt Network proporciona herramientas para usar fácilmente muchos protocolos de red en su aplicación.

Examples

Cliente TCP

Para crear una conexión **TCP** en Qt, usaremos QTcpSocket . Primero, necesitamos conectarnos con `connectToHost` .

Así, por ejemplo, para conectarse a un servidor local de TCP: `_socket.connectToHost(QHostAddress("127.0.0.1"), 4242);`

Luego, si necesitamos leer datos del servidor, necesitamos conectar la señal readyRead con una ranura. Como eso:

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

Y finalmente, podemos leer los datos así:

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

Para escribir datos, puede usar el método de `write(QByteArray)` :

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

Así que un cliente TCP básico puede verse así:

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: (vacío aquí)

```
<?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>
```

Servidor TCP

Crear un **servidor TCP** en Qt también es muy fácil, de hecho, la clase QTcpServer ya proporciona todo lo que necesitamos para hacer el servidor.

Primero, debemos escuchar cualquier ip, un puerto aleatorio y hacer algo cuando un cliente está conectado. como eso:

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

Luego, cuando haya una nueva conexión, podemos agregarla a la lista de clientes y prepararnos para leer / escribir en el socket. Como eso:

```
    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);
```

`stateChanged(QAbstractSocket::SocketState)` nos permite eliminar el socket de nuestra lista cuando el cliente está desconectado.

Así que aquí un servidor de chat básico:

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()));
    }
}
```

(use el mismo mainwindow.ui que el ejemplo anterior)

Lea Red qt en línea: https://riptutorial.com/es/qt/topic/9683/red-qt

Capítulo 19: Roscado y concurrencia

Observaciones

Algunas notas que ya se mencionan en los documentos oficiales aquí y aquí :

- Si un objeto tiene un padre, tiene que estar en el mismo hilo que el padre, es decir, no se puede mover a un hilo nuevo, ni se puede establecer un padre a un objeto si el padre y el objeto viven en hilos diferentes.
- Cuando un objeto se mueve a un nuevo hilo, todos sus hijos también se mueven al nuevo hilo
- Solo puedes *empujar* objetos a un nuevo hilo. No puede *arrastrarlos* a un nuevo hilo, es decir, solo puede llamar a `moveToThread` desde el hilo en el que el objeto vive actualmente.

Examples

Uso básico de QThread

`QThread` es un identificador de un hilo de plataforma. Le permite administrar el hilo monitoreando su vida útil y solicitando que finalice su trabajo.

En la mayoría de los casos no se recomienda heredar de la clase. El método de `run` predeterminado inicia un bucle de eventos que puede enviar eventos a objetos que viven en la clase. Las conexiones de ranura de señal entre hilos se implementan mediante el envío de un `QMetaCallEvent` al objeto de destino.

Una instancia de `QObject` se puede mover a un subproceso, donde procesará sus eventos, como eventos de temporizador o llamadas de ranura / método.

Para trabajar en un hilo, primero cree su propia clase de trabajador que se deriva de `QObject` . Luego muévelo al hilo. El objeto puede ejecutar su propio código automáticamente, por ejemplo, utilizando `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 su trabajador debería ser efímero y solo existir mientras se realiza su trabajo, es mejor enviar un functor o un método seguro para subprocesos para su ejecución en el grupo de subprocesos a través de `QtConcurrent::run` .

QtConcurrent Run

Si considera que la gestión de QThreads y primitivas de bajo nivel como mutex o semáforos es demasiado compleja, lo que está buscando es el espacio de nombres concurrente de Qt. Incluye clases que permiten una mayor gestión de hilos de alto nivel.

Echemos un vistazo a la carrera concurrente. `QtConcurrent::run()` permite ejecutar la función en un nuevo hilo. ¿Cuándo te gustaría usarlo? Cuando tienes alguna operación larga y no quieres crear un hilo manualmente.

Ahora el código:

```
#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();
}
```

Así que las cosas son simples: cuando necesitamos ejecutar otra función en otro subproceso, simplemente llame a `QtConcurrent::run` , pass function y sus parámetros, ¡y eso es todo!

`QFuture` presenta el resultado de nuestro cálculo asíncrono. En el caso de `QtConcurrent::run` no

podemos cancelar la ejecución de la función.

Invocando ranuras de otros hilos

Cuando se usa un bucle de eventos Qt para realizar operaciones y un usuario que no es de Qt-saavy necesita interactuar con ese bucle de eventos, escribir la ranura para manejar invocaciones regulares de otro hilo puede simplificar las cosas para otros usuarios.

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
```

Lea Roscado y concurrencia en línea: https://riptutorial.com/es/qt/topic/5022/roscado-y-concurrencia

Capítulo 20: Señales y Slots

Introducción

Las señales y las ranuras se utilizan para la comunicación entre objetos. El mecanismo de señales y ranuras es una característica central de Qt. En la programación de la GUI, cuando cambiamos un widget, a menudo queremos que otro widget sea notificado. De manera más general, queremos que los objetos de cualquier tipo puedan comunicarse entre sí. Las señales son emitidas por los objetos cuando cambian su estado de una manera que puede ser interesante para otros objetos. Las ranuras pueden usarse para recibir señales, pero también son funciones miembro normales.

Observaciones

La documentación oficial sobre este tema se puede encontrar aquí .

Examples

Un pequeño ejemplo

Las señales y las ranuras se utilizan para la comunicación entre objetos. El mecanismo de señales y ranuras es una característica central de Qt y probablemente la parte que más se diferencia de las características proporcionadas por otros marcos.

El ejemplo mínimo requiere una clase con una señal, una ranura y una conexión:

contador.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
```

El `main` establece un nuevo valor. Podemos comprobar cómo se llama la ranura, imprimiendo el valor.

```
#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();
}
```

Finalmente, nuestro archivo de proyecto:

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

La nueva sintaxis de conexión Qt5.

La sintaxis de `connect` convencional que usa las macros `SIGNAL` y `SLOT` funciona completamente en tiempo de ejecución, que tiene dos inconvenientes: tiene una sobrecarga de tiempo de ejecución (que también produce una sobrecarga de tamaño binario), y no hay verificación de corrección en tiempo de compilación. La nueva sintaxis aborda ambos problemas. Antes de verificar la sintaxis en un ejemplo, deberíamos saber qué sucede en particular.

Digamos que estamos construyendo una casa y queremos conectar los cables. Esto es exactamente lo que hace la función de conexión. Las señales y las ranuras son las que necesitan esta conexión. El punto es que si realiza una conexión, debe tener cuidado con las conexiones superpuestas. Cada vez que conectas una señal a una ranura, estás tratando de decirle al compilador que cada vez que se emitió la señal, simplemente invoca la función de ranura. Esto es lo que sucede exactamente.

Aquí hay un ejemplo 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();
}
```

Sugerencia: la sintaxis antigua (macros de `SIGNAL` / `SLOT`) requiere que el metacompilador Qt (MOC) se ejecute para cualquier clase que tenga ranuras o señales. Desde el punto de vista de la

codificación, eso significa que dichas clases deben tener la macro `Q_OBJECT` (lo que indica la necesidad de ejecutar MOC en esta clase).

La nueva sintaxis, por otro lado, aún requiere MOC para que las señales funcionen, pero **no** para las ranuras. Si una clase solo tiene slots y ninguna señal, no necesita tener la macro `Q_OBJECT` y, por lo tanto, no puede invocar el MOC, lo que no solo reduce el tamaño binario final sino que también reduce el tiempo de compilación (no hay una llamada MOC ni una llamada de compilador posterior para el `*_moc.cpp` archivo `*_moc.cpp`).

Conexión de señales / slots sobrecargados

Aunque es mejor en muchos aspectos, la nueva sintaxis de conexión en Qt5 tiene una gran debilidad: la conexión de señales y ranuras sobrecargadas. Para permitir que el compilador resuelva las sobrecargas, necesitamos usar `static_cast` s para los punteros de función miembro, o (comenzando en Qt 5.7) `qOverload` y amigos:

```
#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
}
```

Conexión de ranura de señal de ventana múltiple

Un ejemplo simple de ventanas múltiples que usa señales y ranuras.

Hay una clase de MainWindow que controla la vista de la ventana principal. Una segunda ventana controlada por clase de sitio web.

Las dos clases están conectadas para que cuando haga clic en un botón en la ventana del sitio web ocurra algo en la ventana principal (se cambia una etiqueta de texto).

Hice un ejemplo simple que también está en 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();
}
```

Composición del proyecto:

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

HEADERS  += mainwindow.h \
    website.h

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

Considere la posibilidad de componer la UIS:

- Ventana principal: una etiqueta llamada "texto" y un botón llamado "openButton"
- Ventana del sitio web: un botón llamado "changeButton"

Por lo tanto, los puntos clave son las conexiones entre señales y ranuras y la gestión de los punteros o referencias de Windows.

Lea Señales y Slots en línea: https://riptutorial.com/es/qt/topic/2136/senales-y-slots

Capítulo 21: Sistema de Recursos Qt

Introducción

El sistema de recursos Qt es una forma de incrustar archivos dentro de su proyecto. Cada archivo de recursos puede tener uno o más *prefijos* y cada *prefijo* puede tener archivos en él.

Cada archivo en los recursos es un enlace a un archivo en el sistema de archivos. Cuando se crea el ejecutable, los archivos se agrupan en el ejecutable, por lo que el archivo original no necesita ser distribuido con el binario.

Examples

Referencias de archivos dentro del código

Digamos que dentro de un archivo de recursos, tenías un archivo llamado /icons/ok.png

La url completa de este archivo dentro del código es `qrc:/icons/ok.png` . En la mayoría de los casos, esto se puede `:/icons/ok.png` a `:/icons/ok.png`

Por ejemplo, si desea crear un QIcon y configurarlo como el icono de un botón de ese archivo, puede usar

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

Lea Sistema de Recursos Qt en línea: https://riptutorial.com/es/qt/topic/8776/sistema-de-recursos-qt

Capítulo 22: Sobre el uso de diseños, la crianza de widgets

Introducción

Los diseños son necesarios en todas las aplicaciones Qt. Manejan el objeto, su posición, su tamaño, cómo se redimensionan.

Observaciones

De la documentación de diseño de Qt :

> Cuando utiliza un diseño, no necesita pasar un padre al construir los widgets hijos. El diseño reparará automáticamente los widgets (usando QWidget :: setParent ()) de modo que sean hijos del widget en el que está instalado el diseño.

Así que hazlo:

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

o hacer

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

es exactamente lo mismo.

Examples

Disposición horizontal básica

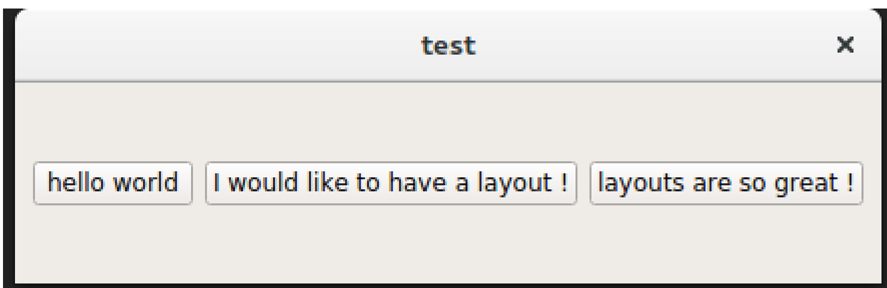
La disposición horizontal configura el objeto dentro de él horizontalmente.

codigo basico

```
#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();
}
```

esto dará como resultado:



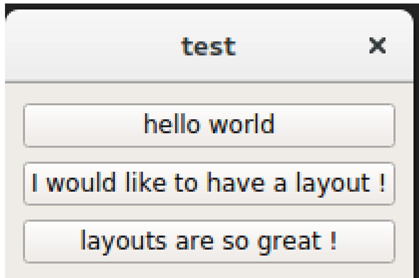
Diseño Vertical Básico

La disposición vertical configura el objeto dentro de él verticalmente.

```
#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();
}
```

salida:



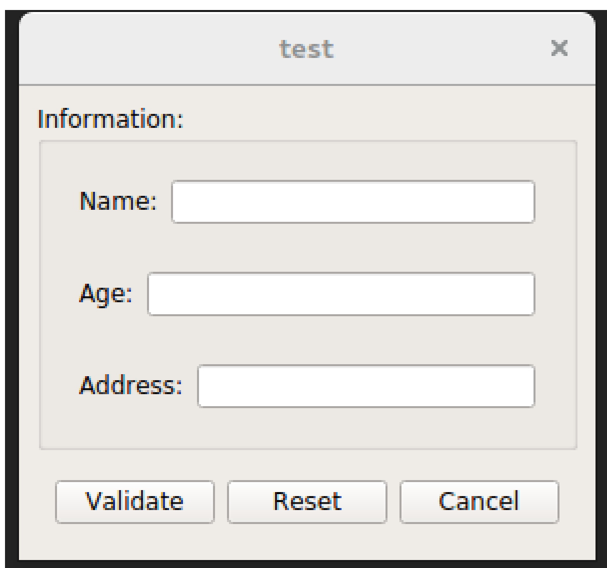
Combinando diseños

Puede combinar un diseño múltiple gracias a otros QWidgets en su diseño principal para realizar efectos más específicos, como un campo de información: por ejemplo:

```
#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();
}
```

saldrá:



Ejemplo de diseño de cuadrícula

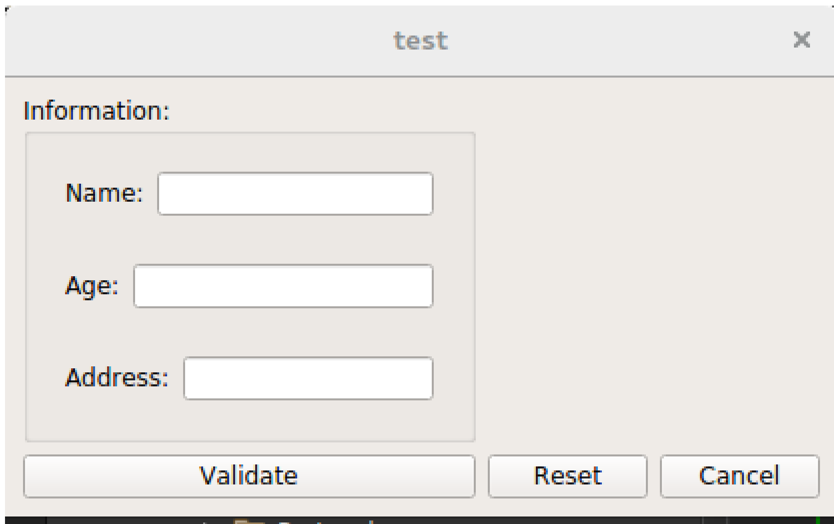
El diseño de cuadrícula es un diseño potente con el que puede hacer un diseño horizontal y vertical una vez.

ejemplo:

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

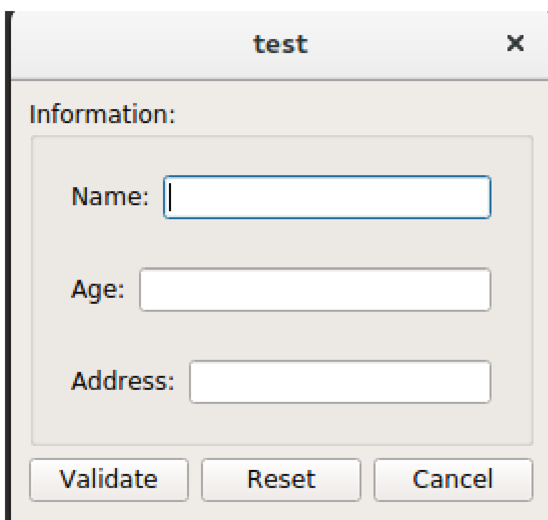
```
#include <QApplication>

#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();
}
```

dar



para que pueda ver que el cuadro de grupo está solo en la primera columna y en la primera fila ya que addWidget era `layout->addWidget(box, 0, 0);`

Sin embargo, si lo cambia a `layout->addWidget(box, 0, 0, 1, 3);` , el nuevo 0 y el 3 representan la cantidad de líneas y columnas que deseas para tu widget, por lo que ofrece:



exactamente lo mismo que creó un diseño horizontal y luego un diseño vertical en un subwidget.

Lea Sobre el uso de diseños, la crianza de widgets en línea: https://riptutorial.com/es/qt/topic/9380/sobre-el-uso-de-disenos--la-crianza-de-widgets

Capítulo 23: SQL en Qt

Examples

Conexión básica y consulta.

La clase QSqlDatabase proporciona una interfaz para acceder a una base de datos a través de una conexión. Una instancia de QSqlDatabase representa la conexión. La conexión proporciona acceso a la base de datos a través de uno de los controladores de base de datos compatibles. Asegúrese de añadir

```
QT += SQL
```

en el archivo .pro. Supongamos una base de datos SQL llamada TestDB con una tabla de país que contine la siguiente columna:

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

Para consultar y obtener datos SQL de 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();
}
```

Parámetros de consulta Qt SQL

A menudo es conveniente separar la consulta SQL de los valores reales. Esto se puede hacer utilizando marcadores de posición. Qt admite dos sintaxis de marcador de posición: enlace con nombre y enlace posicional.

encuadernación nombrada

```
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();
```

unión posicional

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

Tenga en cuenta que antes de llamar a `bindValue()` o `addBindValue()` debe llamar a QSqlQuery :: prepare () una vez.

Conexión de base de datos de MS SQL Server utilizando QODBC

Cuando intente abrir una conexión de base de datos con QODBC, asegúrese de

- Tienes el controlador QODBC disponible
- Su servidor tiene una interfaz ODBC y está habilitado para (esto depende de sus instalaciones de controlador ODBC)
- use el acceso a la memoria compartida, las conexiones TCP / IP o la conexión de tubería con nombre.

Todas las conexiones solo requieren que el nombre de la base de datos se establezca llamando a QSqlDatabase :: setDatabaseName.

Conexión abierta mediante acceso a memoria compartida

Para que esta opción funcione, deberá tener acceso a la memoria de la máquina y debe tener permisos para acceder a la memoria compartida. Para utilizar una conexión de memoria compartida, es necesario configurar lpc: delante de la cadena del servidor. La conexión mediante el SQL Server Native Client 11 se realiza mediante estos pasos:

```
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");
}
```

Conexión abierta usando una tubería con nombre

Esta opción requiere que su conexión ODBC tenga un DSN completo. La cadena del servidor se configura mediante el nombre de computadora de Windows y el nombre de instancia del servidor SQL. La conexión de ejemplo se abrirá utilizando 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");
}
```

Conexión abierta usando TCP / IP

Para abrir una conexión TCP / IP, el servidor debe configurarse para permitir conexiones en un puerto fijo, de lo contrario, primero deberá consultar el puerto activo actual. En este ejemplo, tenemos un puerto fijo en 5171. Puede encontrar un ejemplo para configurar el servidor para permitir conexiones en un puerto fijo en 1 . Para abrir una conexión usando TCP / IP, use una tupla de los servidores IP y Puerto:

```
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");
}
```

Lea SQL en Qt en línea: https://riptutorial.com/es/qt/topic/10628/sql-en-qt

Capítulo 24: Usar hojas de estilo con eficacia

Examples

Configuración de la hoja de estilo de un widget UI

Puede configurar la hoja de estilo del widget de la interfaz de usuario deseada utilizando cualquier CSS válido. El siguiente ejemplo establecerá un color de texto de QLabel en un borde a su alrededor.

```
#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;
}
```

Lea Usar hojas de estilo con eficacia en línea: https://riptutorial.com/es/qt/topic/5931/usar-hojas-de-estilo-con-eficacia

Creditos

S. No	Capítulos	Contributors
1	Empezando con Qt	agilob, Christopher Aldama, Community, demonplus, devbean, Dmitriy, Donald Duck, fat, Gabriel de Grimouard, Kamalpreet Grewal, Maxito, Tarod, thiagofalcao
2	Clases de Contenedores Qt	demonplus, Tarod
3	CMakeLists.txt para su proyecto Qt	Athena, demonplus, Robert, Velkan, wasthishelpful
4	Comunicación entre QML y C ++.	Gabriel de Grimouard, Martin Zhai
5	Construye QtWebEngine desde la fuente	Martin Zhai
6	Despliegue de aplicaciones Qt	Luca Angioloni, Martin Zhai, Nathan Osman, TriskalJM, wasthishelpful
7	Encabezado en QListView	Papipone
8	Errores comunes	e.jahandar
9	Intercambio implícito	Hayt
10	Modelo / Vista	Jan, KernelPanic, Tim D
11	Multimedia	demonplus, Gabriel de Grimouard
12	QDialogs	Wilmort
13	Qgraphics	Chris, demonplus
14	qmake	Caleb Huitt - cjhuitt, demonplus, doc, Gregor, Jon Harper
15	QObject	demonplus, Hayt
16	Qt - Tratar con bases de datos	Jan, Rinat, Shihe Zhang, Zylva
17	QTimer	avb, Caleb Huitt - cjhuitt, Eugene, Gabriel de Grimouard, Hayt,

		Rinat, Tarod, thuga, tpr, Victor Tran
18	Red qt	Gabriel de Grimouard
19	Roscado y concurrencia	demonplus, gmabey, Kuba Ober, Nathan Osman, RamenChef, thuga
20	Señales y Slots	Athena, devbean, fat, immerhart, Jan, Luca Angioloni, Robert, Tarod, Violet Giraffe
21	Sistema de Recursos Qt	Victor Tran
22	Sobre el uso de diseños, la crianza de widgets	Gabriel de Grimouard
23	SQL en Qt	Noam M
24	Usar hojas de estilo con eficacia	Nicholas Johnson