EBook Gratuito

APPRENDIMENTO

Qt

Free unaffiliated eBook created from
Stack Overflow contributors.

#qt

Sommario

Di

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

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

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

Capitolo 1: Iniziare con Qt

Osservazioni

Come indicato nella documentazione ufficiale , Qt è un framework di sviluppo di applicazioni multipiattaforma per desktop, embedded e mobile. Le piattaforme supportate includono Linux, OS X, Windows, VxWorks, QNX, Android, iOS, BlackBerry, Sailfish OS e altri.

Questa sezione fornisce una panoramica di cosa sia Qt e perché uno sviluppatore potrebbe volerlo utilizzare.

Dovrebbe anche menzionare qualsiasi argomento di grandi dimensioni all'interno di Qt e collegarsi agli argomenti correlati. Poiché la documentazione per qt è nuova, potrebbe essere necessario creare versioni iniziali di tali argomenti correlati.

Versioni

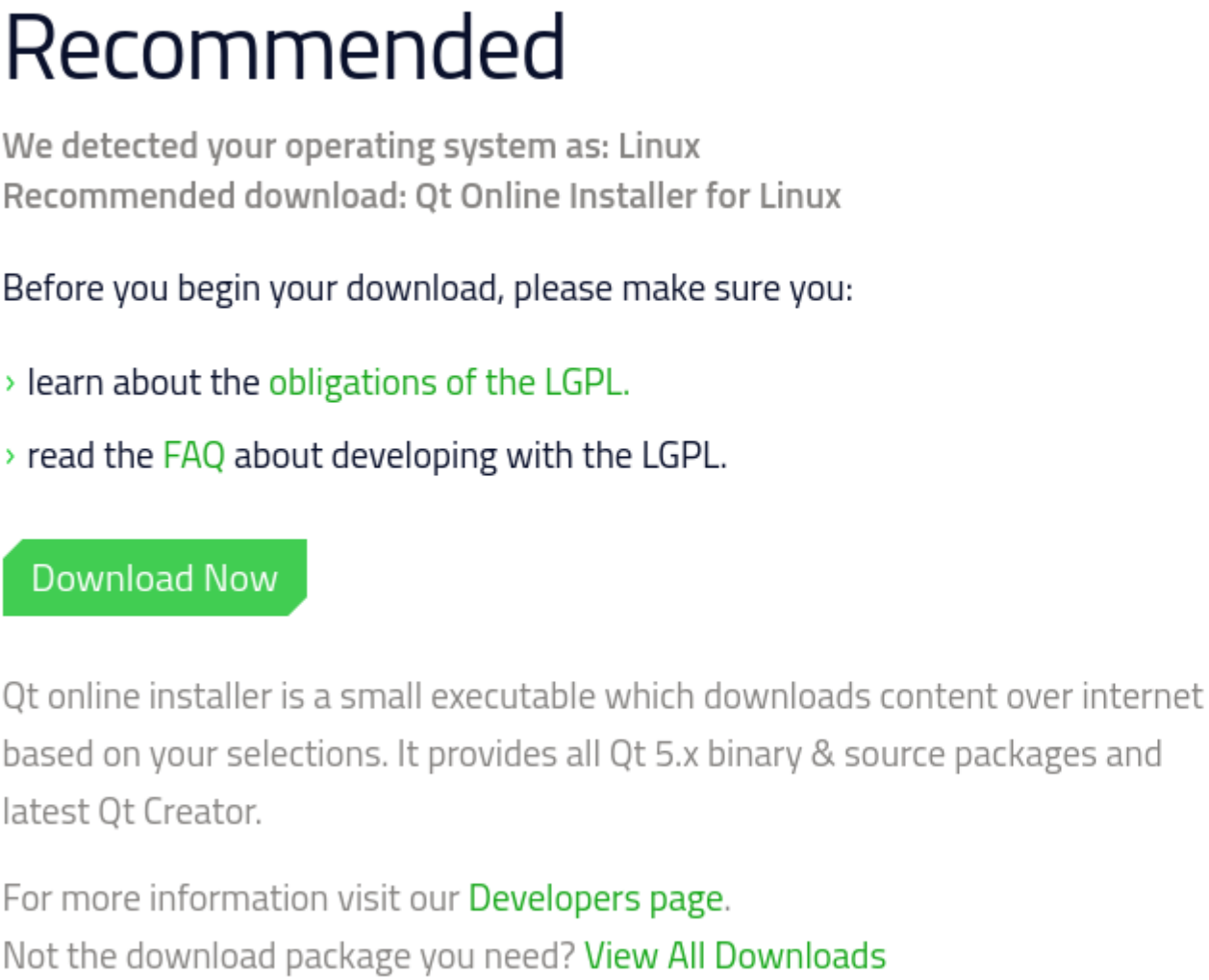
Versione	Data di rilascio
Qt 3.0	2001/10/16
Qt 3.3	2004-02-05
Qt 4,1	2005-12-20
Qt 4,8	2011-12-15
Qt 5.0	2012/12/19
Qt 5.6	2016/03/16
Qt 5.7	2016/06/16
Qt 5.8	2017/01/23
Qt 5.9	2017/05/31

Examples

Installazione e installazione su Windows e Linux

Scarica Qt per Linux Open Source Version

Vai a https://www.qt.io/download-open-source/ e fai clic su Scarica ora, assicurati di scaricare il programma di installazione Qt per Linux.



Un file con il nome qt-unified-linux-x-online.run verrà scaricato, quindi aggiungere il permesso exec

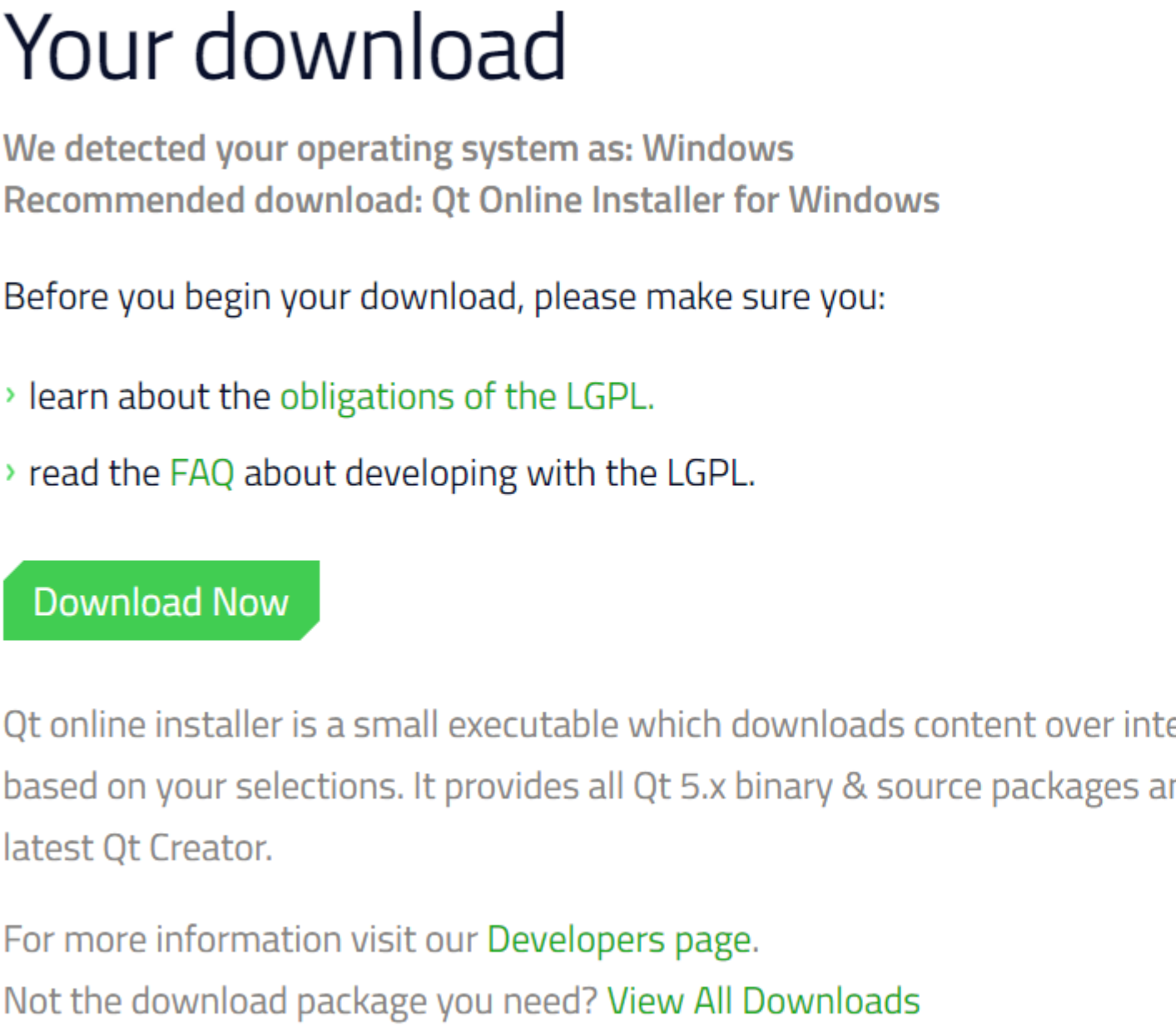
```
chmod +x qt-unified-linux-x-online.run
```

Ricordarsi di cambiare "x" per la versione attuale del programma di installazione. Quindi esegui il programma di installazione

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

Scarica Qt per Windows Open Source Version

Vai a https://www.qt.io/download-open-source/ . La seguente schermata mostra la pagina di download su Windows:



Quello che dovresti fare ora dipende da quale IDE intendi utilizzare. Se si intende utilizzare Qt Creator, incluso nel programma di installazione, fare clic su Scarica ora ed eseguire l'eseguibile.

Se si utilizza Qt in Visual Studio, normalmente anche il pulsante Scarica ora dovrebbe funzionare. Assicurati che il file scaricato si chiami qt-opensource-windows-x86-msvc2015_64-xxxexe o qt-opensource-windows-x86-msvc2015_32-xxxexe (dove xxx è la versione di Qt, ad esempio 5.7.0). In caso contrario, fai clic su Visualizza tutti i download e seleziona una delle prime quattro opzioni in Windows Host.

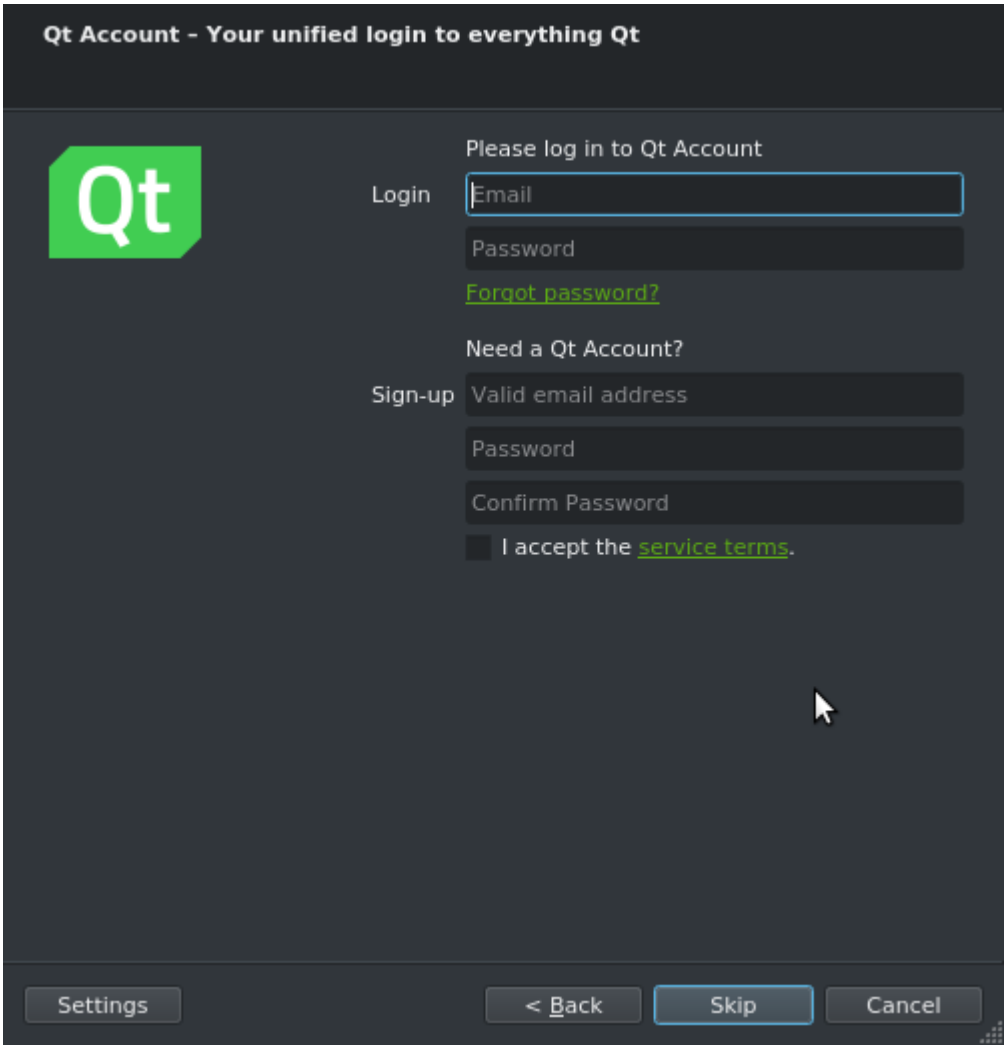
Se si utilizza Qt in Code :: Blocks, fare clic su Visualizza tutti i download e selezionare Qt xxx per Windows a 32 bit (MinGW xxx, 1,2 GB) in Host Windows.

Una volta scaricato il file di installazione appropriato, esegui il file eseguibile e segui le istruzioni di seguito. Nota che devi essere un amministratore per installare Qt. Se non sei un amministratore, puoi trovare diverse soluzioni alternative qui .

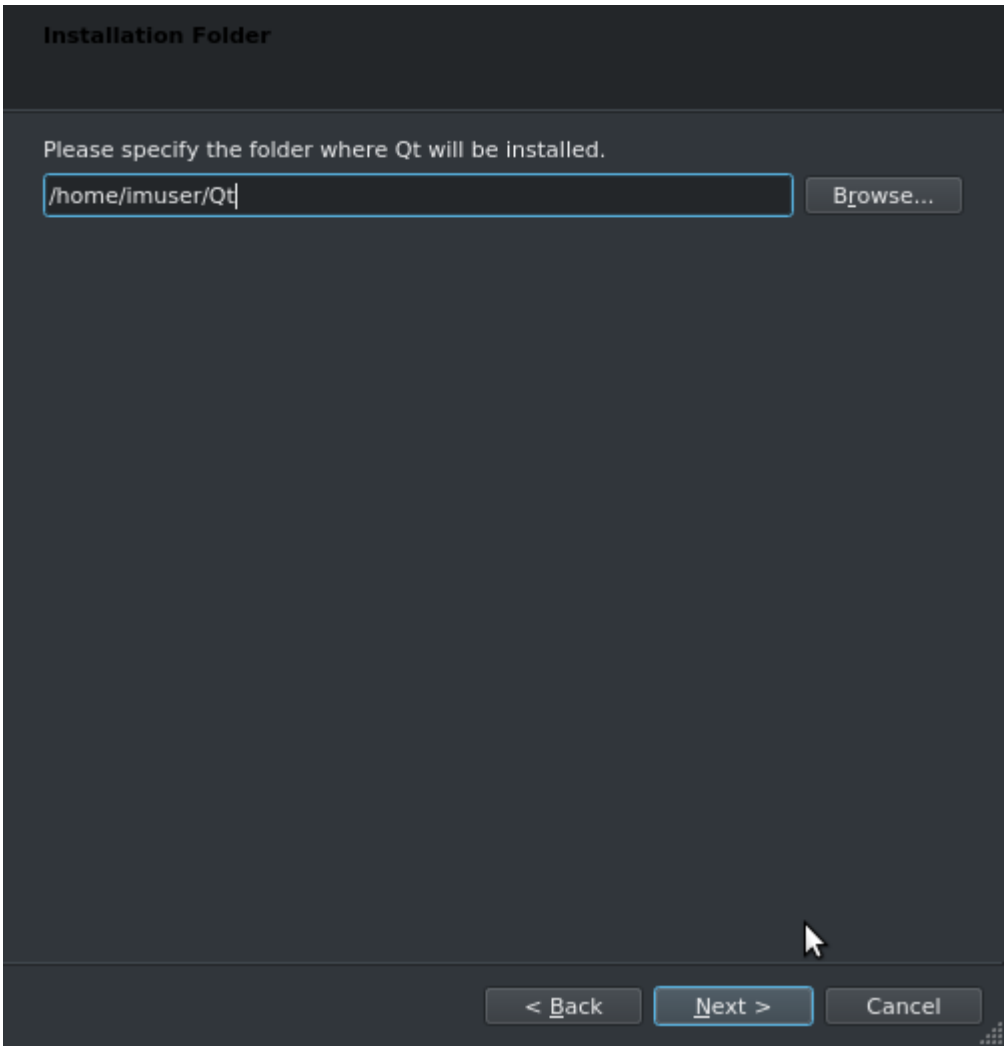
Installa Qt in qualsiasi sistema operativo

Una volta scaricato Qt e aperto il programma di installazione, la procedura di installazione è la stessa per tutti i sistemi operativi, anche se gli screenshot potrebbero sembrare un po 'diversi. Gli screenshot qui forniti sono di Linux.

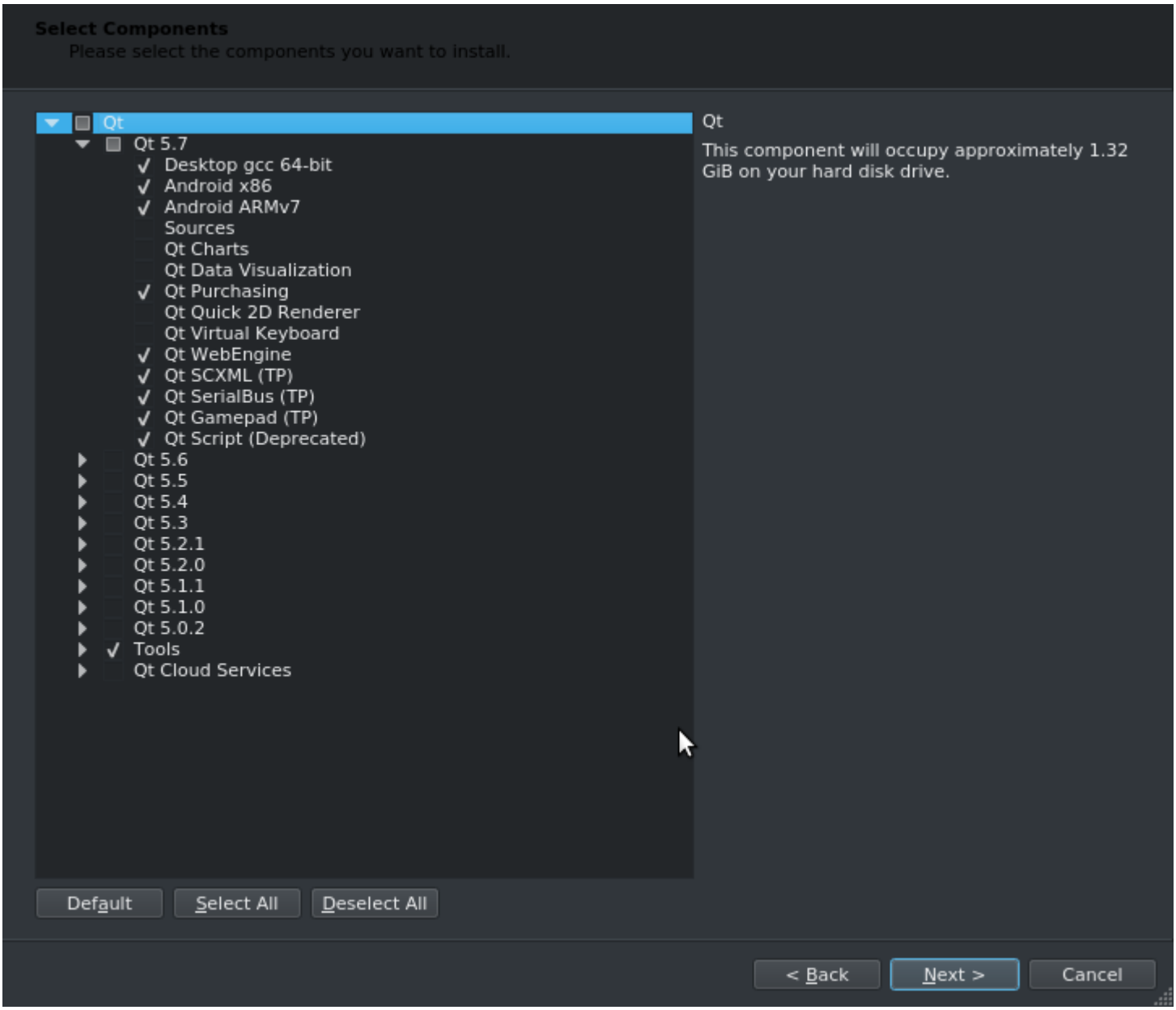
Accedi con un account Qt esistente o creane uno nuovo:



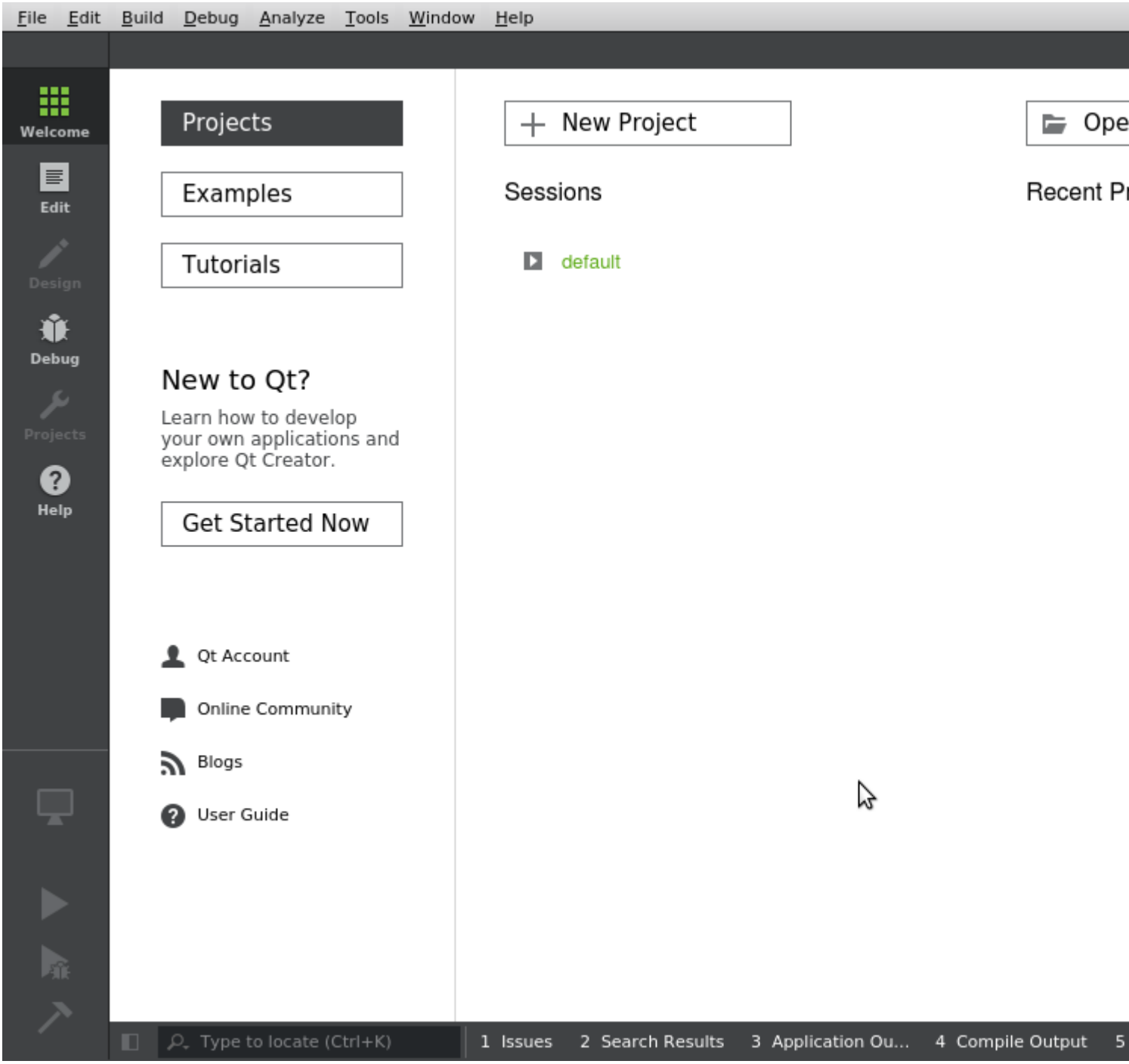
Selezionare un percorso per installare le librerie e gli strumenti Qt



Seleziona la versione della libreria e le funzionalità che desideri



Dopo aver scaricato e completata l'installazione, accedere alla directory di installazione Qt e avviare Qt Creator o eseguirlo direttamente dalla riga di comando.



Ciao mondo

In questo esempio, semplicemente creiamo e mostriamo un pulsante in una cornice della finestra sul desktop. Il pulsante avrà l'etichetta `Hello world!`

Questo rappresenta il programma Qt più semplice possibile.

Prima di tutto abbiamo bisogno di un file di progetto:

helloworld.pro

```
QT       += core gui

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

```
TARGET = helloworld
TEMPLATE = app

SOURCES += main.cpp
```

- QT viene utilizzato per indicare quali librerie (moduli Qt) vengono utilizzate in questo progetto. Poiché la nostra prima app è una piccola GUI, avremo bisogno di QtCore e QtGui. Essendo Qt5 separato da QtWidgets da QtGui, abbiamo bisogno di aggiungere `greaterThan` riga più grande per compilarlo con Qt5.
- TARGET è il nome dell'app o della biblioteca.
- TEMPLATE descrive il tipo da costruire. Può essere un'applicazione (app), una libreria (lib) o semplicemente sottodirectory (sottodirectory).
- SOURCES è un elenco di file di codice sorgente da utilizzare durante la creazione del progetto.

Abbiamo anche bisogno del main.cpp che contiene un'applicazione 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'
}
```

- QApplicazione oggetto. Questo oggetto gestisce le risorse a livello di applicazione ed è necessario per eseguire qualsiasi programma Qt con una GUI. Ha bisogno di argv e arg perché Qt accetta alcuni argomenti della riga di comando. Quando si chiama `a.exec()` viene avviato il ciclo di eventi Qt.
- Oggetto QPushButton. Il pulsante con l'etichetta `Hello world!` . La riga successiva, `button.show()` , mostra il pulsante sullo schermo nella sua cornice della finestra.

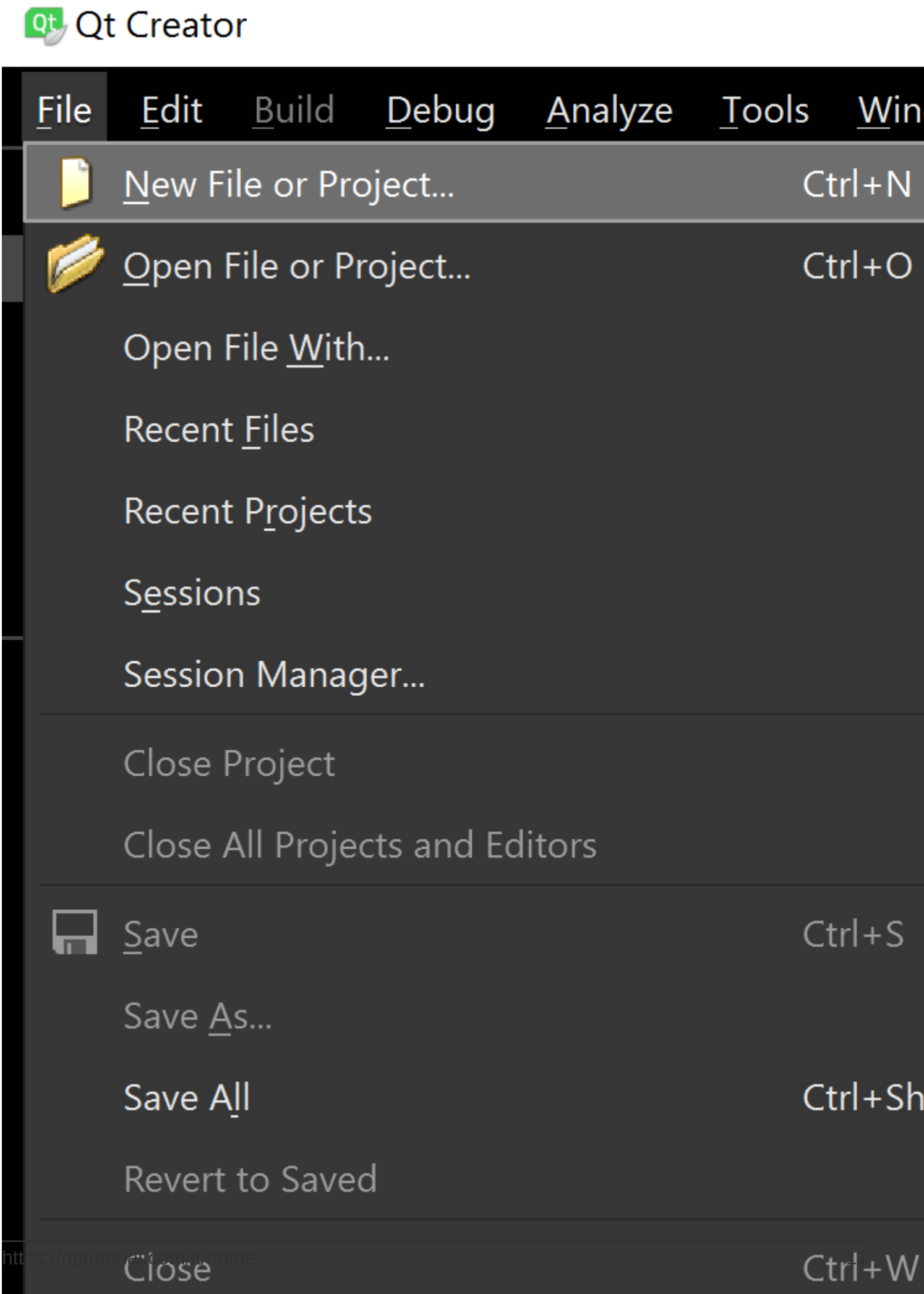
Infine, per eseguire l'applicazione, apri un prompt dei comandi e inserisci la directory in cui è presente il file .cpp del programma. Digitare i seguenti comandi della shell per creare il programma.

```
qmake -project
qmake
make
```

Applicazione di base con QtCreator e QtDesigner

QtCreator è, al momento, lo strumento migliore per creare un'applicazione Qt. In questo esempio, vedremo come creare una semplice applicazione Qt che gestisca un pulsante e scriva del testo.

Per creare una nuova applicazione, fare clic su File-> Nuovo file o progetto:


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

che è una classe fantastica che può convertire molte cose in molte altre cose. Così a sinistra aggiungi un int che aumenta quando premiamo il pulsante.

Quindi il .h:

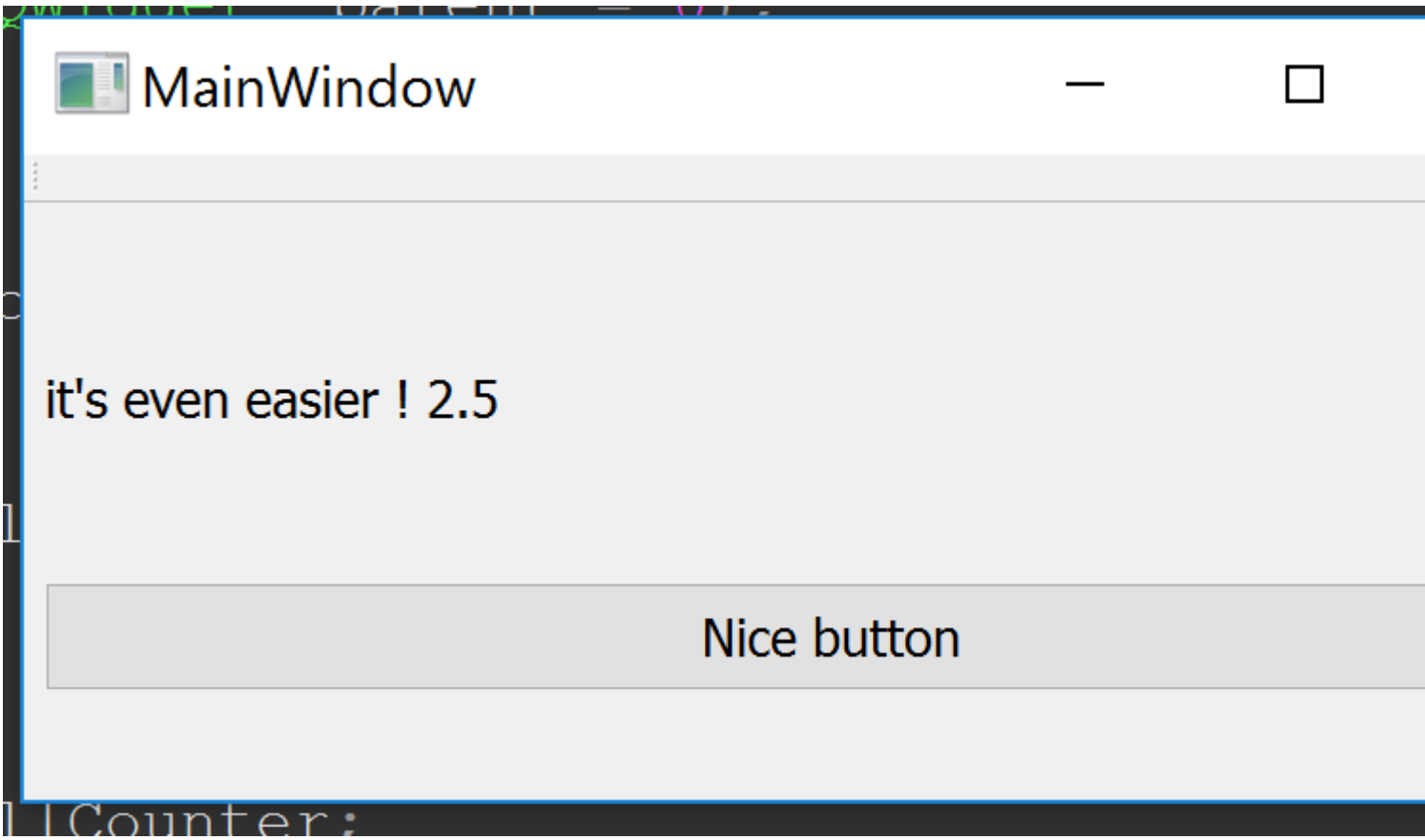
```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#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
```

Il .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());
}
```

E ora, possiamo salvare e correre di nuovo. Ogni volta che fai clic sul pulsante, mostra "è ancora più semplice!" Con il valore di _smallCounter. Quindi dovresti avere qualcosa del tipo:



Questo tutorial è fatto. Se vuoi saperne di più su Qt, vediamo gli altri esempi e la documentazione di Qt sulla Documentazione StackOverflow o sulla documentazione Qt

Leggi Iniziare con Qt online: https://riptutorial.com/it/qt/topic/902/iniziare-con-qt

Capitolo 2: Classi di contenitori Qt

Osservazioni

Qt fornisce le proprie classi di contenitori template. Sono tutti implicitamente condivisi. Forniscono due tipi di iteratori (stile Java e stile STL).

I contenitori sequenziali Qt includono: QVector, QList, QLinkedList, QStack, QQueue.

I contenitori associativi Qt includono: QMap, QMultiMap, QHash, QMultiHash, QSet.

Examples

QStack di utilizzo

`QStack<T>` è una classe Qt di template che fornisce stack. Il suo analogo in STL è `std::stack` . È l'ultimo, la prima struttura (LIFO).

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

Produrrà: Terzo, Secondo, Primo.

`QStack` eredita da `QVector` quindi la sua implementazione è abbastanza diversa da STL. In STL `std::stack` è implementato come wrapper da digitare passato come argomento template (deque di default). Le operazioni principali ancora sono le stesse per `QStack` e per `std::stack` .

Uso di QVector

`QVector<T>` fornisce una classe template array dinamica. Fornisce prestazioni migliori nella maggior parte dei casi rispetto a `QList<T>` quindi dovrebbe essere la prima scelta.

Può essere inizializzato in vari modi:

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

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

L'ultimo include l'elenco di inizializzazione.

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

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

Puoi ottenere l'elemento i-esimo del vettore in questo modo:

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

Assicurati che `i` sia una posizione valida, anche `at(i)` non effettua un controllo, questa è una differenza rispetto a `std::vector` .

Uso di QLinkedList

In Qt dovresti usare QLinkedList nel caso sia necessario implementare l' elenco collegato .

È veloce aggiungere, anteporre, inserire elementi in `QLinkedList` - O (1), ma la ricerca dell'indice è più lenta che in `QList` o `QVector` - O (n). È normale prendere in considerazione che devi scorrere i nodi per trovare qualcosa nella lista collegata.

La tabella completa della compexity algoritmica può essere trovata qui .

Solo per inserire alcuni elementi in `QLinkedList` è possibile utilizzare l'operatore `<<()` :

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

Per inserire elementi nel mezzo di `QLinkedList` o modificare tutti o alcuni dei suoi elementi è possibile utilizzare gli stili iteratori stile Java o STL. Ecco un semplice esempio di come moltiplichiamo tutti gli elementi di `QLinkedList` per 2:

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

QList

La classe `QList` è una classe template che fornisce elenchi. Memorizza gli articoli in un elenco che fornisce accesso rapido basato su indice e inserzioni e rimozioni basate su indici.

Per inserire elementi nell'elenco, puoi utilizzare l' `operator<<()` , `insert()` , `append()` o `prepend()` . Per esempio:

`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");
```

Per accedere all'elemento in una particolare posizione di indice, è possibile utilizzare l' `operator[]()` o `at()` . `at()` può essere più veloce `operator[]()` , non causa mai una copia profonda del contenitore e dovrebbe funzionare in tempo costante. Nessuno dei due controlla gli argomenti. Esempi:

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

O

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

Per rimuovere elementi, esistono funzioni come `removeAt()` , `takeAt()` , `takeFirst()` , `takeLast()` , `removeFirst()` , `removeLast()` o `removeOne()` . Esempi:

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");
```

Per trovare tutte le occorrenze di un determinato valore in un elenco, è possibile utilizzare `indexOf()` o `lastIndexOf()` . Esempio:

indexOf()

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

Leggi Classi di contenitori Qt online: https://riptutorial.com/it/qt/topic/6303/classi-di-contenitori-qt

Capitolo 3: CMakeLists.txt per il tuo progetto Qt

Examples

CMakeLists.txt per Qt 5

Un file di progetto CMake minimo che utilizza Qt5 può essere:

```
cmake_minimum_required(VERSION 2.8.11)

project(myproject)

find_package(Qt5 5.7.0 REQUIRED COMPONENTS
    Core
)

set(CMAKE_AUTOMOC ON)

add_executable(${PROJECT_NAME}
    main.cpp
)

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

`cmake_minimum_required` è chiamato per impostare la versione minima richiesta per CMake. La versione minima richiesta per questo esempio per funzionare è `2.8.11` - le versioni precedenti di CMake necessitano di codice aggiuntivo per una destinazione da utilizzare Qt.

`find_package` viene chiamato per cercare un'installazione di Qt5 con una determinata versione - 5.7.0 nell'esempio - e componenti ricercati - modulo Core nell'esempio. Per un elenco dei moduli disponibili, vedere Documentazione Qt . Qt5 è contrassegnato come `REQUIRED` in questo progetto. Il percorso per l'installazione può essere suggerito impostando la variabile `Qt5_DIR` .

`AUTOMOC` è un valore booleano che specifica se CMake gestirà automaticamente il preprocessore di Qt `moc` , cioè senza dover utilizzare la macro `QT5_WRAP_CPP()` .

Altre variabili "AUTOMOC-like" sono:

- `AUTOUIC` : un booleano che specifica se CMake gestirà automaticamente il generatore di codici `uic` Qt, cioè senza dover utilizzare la macro `QT5_WRAP_UI()` .
- `AUTORCC` : un booleano che specifica se CMake gestirà automaticamente il generatore di codici Qt `rcc` , cioè senza dover utilizzare la macro `QT5_ADD_RESOURCES()` .

`add_executable` viene chiamato per creare un target eseguibile dai file di origine dati. Il target viene quindi collegato ai moduli di Qt elencati con il comando `target_link_libraries` . Da CMake

`target_link_libraries` , `target_link_libraries` con i target importati da Qt gestisce i parametri del linker, oltre a includere le directory e le opzioni del compilatore.

Leggi CMakeLists.txt per il tuo progetto Qt online: https://riptutorial.com/it/qt/topic/1991/cmakelists-txt-per-il-tuo-progetto-qt

Capitolo 4: Comunicazione tra QML e C ++

introduzione

Possiamo usare QML per costruire applicazioni ibride, dal momento che è molto più facile di C ++. Quindi dovremmo sapere come comunicano tra loro.

Examples

Chiama C ++ in QML

Registra le classi C ++ in QML

In C ++, immaginiamo di avere una classe chiamata `QmlCppBridge` , che implementa un metodo chiamato `printHello()` .

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

Vogliamo usarlo in QML. Dovremmo registrare la classe chiamando `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");
```

In QML, usa il seguente codice per chiamarlo:

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

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

Usando `QQmlContext` per iniettare classi o variabili C ++ in QML

Usiamo ancora la classe C ++ nell'esempio precedente:

```
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);
```

Sul lato QML:

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

> **Nota:** questo esempio si basa su Qt 5.7. Non sono sicuro se si adatta alle versioni precedenti di Qt.

Chiama QML in C ++

Per chiamare le classi QML in C ++, è necessario impostare la proprietà objectName.

Nel tuo Qml:

```
import QtQuick.Controls 2.0

Button {
    objectName: "buttonTest"
}
```

Quindi, nel tuo C ++, puoi ottenere l'oggetto con `QObject.FindChild<QObject*>(QString)`

Come quello:

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

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

Ora hai il tuo oggetto QML nel tuo C ++. Ma ciò potrebbe sembrare inutile dal momento che non possiamo davvero ottenere i componenti dell'oggetto.

Tuttavia, possiamo usarlo per inviare **segnali** tra QML e C ++. Per fare ciò, è necessario aggiungere un segnale nel file QML in questo modo: `signal buttonClicked(string str)` . Una volta creato questo, è necessario emettere il segnale. Per esempio:

```
import QtQuick 2.0
import QtQuick.Controls 2.1

    Button {
        id: buttonTest
        objectName: "buttonTest"

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

```
    }
```

Qui abbiamo il nostro pulsante qml. Quando facciamo clic su di esso, si passa al metodo **onClicked** (un metodo di base per i pulsanti che viene chiamato quando si preme il pulsante). Quindi usiamo l'id del pulsante e il nome del segnale per emettere il segnale.

E nel nostro cpp, dobbiamo connettere il segnale con uno slot. come quello:

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();
}
```

Come puoi vedere, otteniamo il nostro pulsante qml con `findChild` come prima e colleghiamo il segnale a un gestore Button che è una classe creata e che assomiglia a quella. 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
```

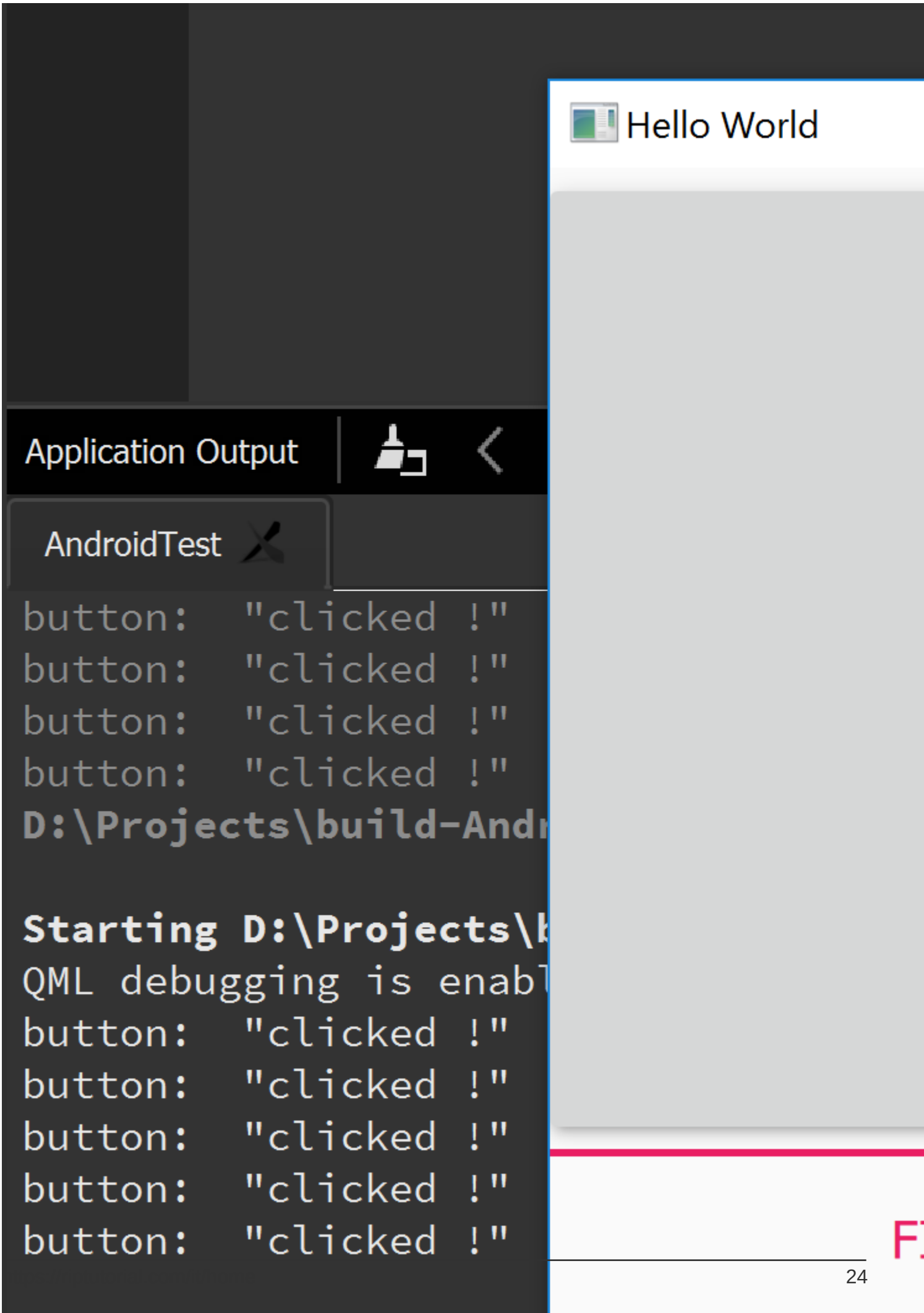
ButtonManager.cpp

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

```
#include <QDebug>

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

}

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

Quindi, quando il segnale sarà ricevuto, chiamerà il metodo `onButtonClicked` che scriverà `"button: clicked !"` `onButtonClicked` `"button: clicked !"`

produzione:


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/it/qt/topic/8735/comunicazione-tra-qml-e-c-plusplus

Capitolo 5: Condivisione implicita

Osservazioni

Gli iteratori di stile STL su Qt Container possono avere un effetto collaterale negativo a causa della condivisione implicita. Si consiglia di evitare di copiare un contenitore Qt mentre sono attivi su iteratori.

```
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

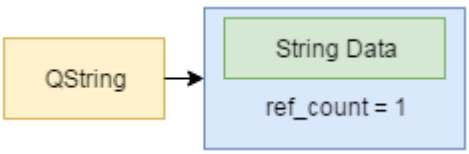
Concetto di base

Diversi oggetti e contenitori Qt usano un concetto che chiama la **condivisione implicita** , che può anche essere chiamata **copia-su-scrittura** .

Condivisione implicita significa che le classi che usano questo concetto condividono gli stessi dati sull'inizializzazione.

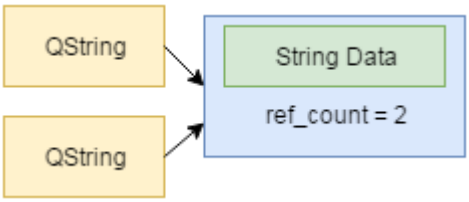
Una di queste classi per usare il concetto è QString.

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



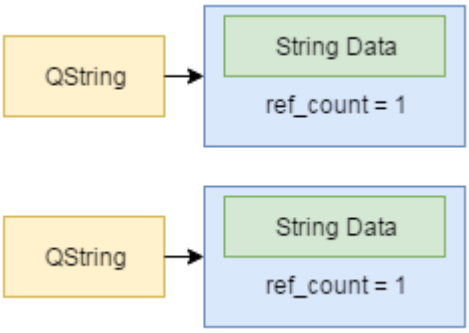
Questo è un modello semplificato di QString. Internamente ha un blocco di memoria, con i dati di stringa effettivi e un contatore di riferimento.

```
QString s2 = s1;
```



Se ora copiamo questo `QString` entrambi gli oggetti puntano internamente allo stesso contenuto, evitando così operazioni di copia non necessarie. Nota come anche il conteggio dei riferimenti è aumentato. Quindi, nel caso in cui la prima stringa venga cancellata, i dati condivisi sanno ancora di essere referenziati da un altro `QString` .

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



Ora, quando `QString` viene effettivamente modificato, l'oggetto "si stacca" dal blocco di memoria, copiandolo e modificando il contenuto.

Leggi Condivisione implicita online: https://riptutorial.com/it/qt/topic/6801/condivisione-implicita

Capitolo 6: Costruisci QtWebEngine dal sorgente

introduzione

A volte abbiamo bisogno di costruire QtWebEngine dal sorgente per qualche motivo, come per il supporto mp3.

Examples

Costruisci su Windows

Requisiti

- Windows 10, si prega di **impostare le impostazioni internazionali del sistema in inglese** , altrimenti potrebbero esserci degli errori
- Visual Studio 2013 o 2015
- QtWebEngine 5.7 codice sorgente (può essere scaricato da qui)
- Qt 5.7 installa la versione, installala e aggiungi la cartella `qmake.exe` al percorso di sistema
- Python 2, aggiungi la cartella `python.exe` al percorso di sistema
- Git, aggiungi la cartella `git.exe` al percorso di sistema
- gperf, aggiungi la cartella `gperf.exe` al percorso di sistema
- flex-bison, aggiungere la cartella `win_bison.exe` al percorso di sistema e rinominarla in `bison.exe`

  Nota: non ho testato le versioni di Visual Studio, tutte le versioni di Qt. Prendiamo un esempio qui, le altre versioni dovrebbero essere più o meno le stesse.

Passi per costruire

1. Decomprimi il codice sorgente in una cartella, chiamiamolo `ROOT`
2. Aprire il `Developer Command Prompt for VS2013` e `Developer Command Prompt for VS2013` alla cartella `ROOT`
3. Esegui `qmake WEBENGINE_CONFIG+=use_proprietary_codecs qtwebengine.pro` . Aggiungiamo questo flag per abilitare il supporto mp3.
4. Esegui `nmake`

   Nota: l' Mp3 non è supportato da QtWebEngine per impostazione predefinita, a causa del problema di licenza. Assicurati di ottenere una licenza per il codec che hai aggiunto.

Leggi Costruisci QtWebEngine dal sorgente online:
https://riptutorial.com/it/qt/topic/8718/costruisci-qtwebengine-dal-sorgente

Capitolo 7: Distribuzione di applicazioni Qt

Examples

Distribuzione su Windows

Qt fornisce uno strumento di distribuzione per Windows: `windeployqt` . Lo strumento controlla un eseguibile dell'applicazione Qt per le sue dipendenze nei moduli Qt e crea una directory di distribuzione con i file Qt necessari per eseguire l'eseguibile ispezionato. Un possibile script potrebbe essere simile a:

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

Il comando `set` viene chiamato per aggiungere la directory `bin` di Qt alla variabile di ambiente `PATH` . `windeployqt` viene quindi chiamato:

- Il percorso della directory di implementazione viene fornito un argomento facoltativo fornito con il parametro `--dir` (il percorso predefinito in cui viene chiamato `windeployqt` è il valore predefinito).
- Il percorso dell'eseguibile da ispezionare viene indicato come ultimo argomento.

La directory di implementazione può quindi essere associata all'eseguibile.

NOTA:

Se stai utilizzando Qt5.7.0 precompilato con vs2013 su Windows *(non sono sicuro che tutte le versioni abbiano questo problema)* , esiste la possibilità di copiare manualmente `<QTDIR>\5.7\msvc2015\qml` nella directory bin di il tuo programma In caso contrario, il programma si chiuderà automaticamente dopo l'avvio.

Vedi anche la documentazione Qt .

Integrazione con CMake

È possibile eseguire `windeployqt` e `macdeployqt` da CMake, ma prima è necessario trovare il percorso degli eseguibili:

```
# 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}")
```

Affinché `windeployqt` trovi le librerie Qt nella loro posizione installata, la cartella deve essere aggiunta a `%PATH%` . Per fare questo per un obiettivo chiamato `myapp` dopo essere stato costruito:

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

Per eseguire `macdeployqt` su un bundle, sarebbe fatto in questo modo:

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

Distribuzione su Mac

Qt offre uno strumento di distribuzione per Mac: lo strumento di distribuzione Mac.

Lo strumento di distribuzione Mac è disponibile in `QTDIR/bin/macdeployqt` . È progettato per automatizzare il processo di creazione di un bundle applicativo distribuibile che contiene le librerie Qt come framework privati.

Lo strumento di distribuzione mac distribuisce anche i plugin Qt, in base alle seguenti regole (a meno che **non venga utilizzata l'opzione -no-plugins**):

- Il plug-in della piattaforma viene sempre distribuito.
- Le versioni di debug dei plug-in non vengono distribuite.
- I plug-in di progettazione non vengono distribuiti.
- I plug-in di formato immagine vengono sempre distribuiti.
- Il plug-in di supporto per la stampa viene sempre distribuito.
- I plugin del driver SQL vengono distribuiti se l'applicazione utilizza il modulo Qt SQL.
- I plug-in di script vengono distribuiti se l'applicazione utilizza il modulo Qt Script.
- Il plug-in per icone SVG viene distribuito se l'applicazione utilizza il modulo QG SVG.
- Il plug-in per l'accessibilità è sempre implementato.

Per includere una libreria di terze parti nel pacchetto di applicazioni, copiare manualmente la raccolta nel pacchetto, dopo aver creato il pacchetto.

Per utilizzare `macdeployqt` strumento `macdeployqt` è possibile aprire il terminale e digitare:

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

Il file dell'app ora conterrà tutte le librerie Qt utilizzate come framework privati.

`macdeployqt` supporta anche le seguenti opzioni

Opzione	Descrizione
verbose = <0-3>	0 = nessun output, 1 = errore / avviso (predefinito), 2 = normale, 3 = debug
-no-plugins	Ignora la distribuzione del plug-in
-dmg	Creare un'immagine del disco .dmg
-no-strip	Non eseguire 'strip' sui binari
-use-debug-libs	Distribuire con versioni di debug di framework e plug-in (implica -no-strip)
-executable =	Lascia che il dato eseguibile usi anche i framework distribuiti
-qmldir =	Distribuire le importazioni utilizzate dai file .qml nel percorso specificato

Informazioni dettagliate possono essere fonte di informazioni sulla documentazione Qt

Distribuzione su Linux

C'è uno strumento di distribuzione per linux su GitHub . Anche se non è perfetto, è collegato al wiki Qt. Si basa concettualmente sul Qt Mac Deployment Tool e funziona in modo simile fornendo un'Immagine .

Dato che un file desktop deve essere fornito con un'Immagine, `linuxdeployqt` può usarlo per determinare i parametri della build.

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

Dove il file desktop specifica l'eseguibile da eseguire (con `EXEC=`), il nome dell'applicazione e un'icona.

Leggi Distribuzione di applicazioni Qt online: https://riptutorial.com/it/qt/topic/5857/distribuzione-di-applicazioni-qt

Capitolo 8: Informazioni sull'uso di layout, genitorialità di widget

introduzione

I layout sono necessari in ogni applicazione Qt. Gestiscono l'oggetto, la loro posizione, la loro dimensione, il modo in cui vengono ridimensionati.

Osservazioni

Dalla documentazione del layout Qt :

> Quando si utilizza un layout, non è necessario passare un genitore quando si costruiscono i widget secondari. Il layout riparerà automaticamente i widget (usando QWidget :: setParent ()) in modo che siano figli del widget su cui è installato il layout.

Quindi:

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

o fare:

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

è esattamente lo stesso

Examples

Layout orizzontale di base

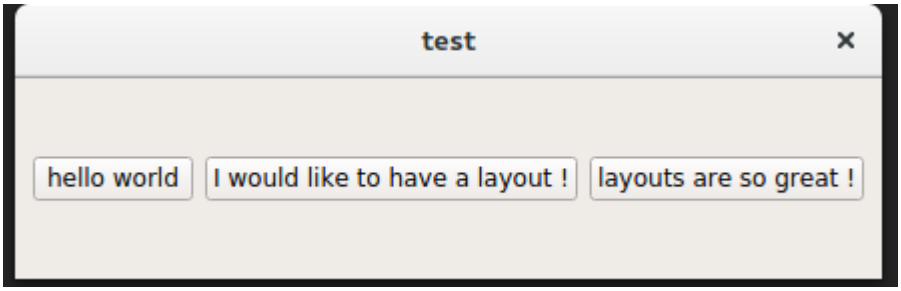
Il layout orizzontale imposta l'oggetto al suo interno in senso orizzontale.

codice di base:

```
#include <QApplication>

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

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

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

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



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

    window.show();

    return a.exec();
}
```

questo produrrà:



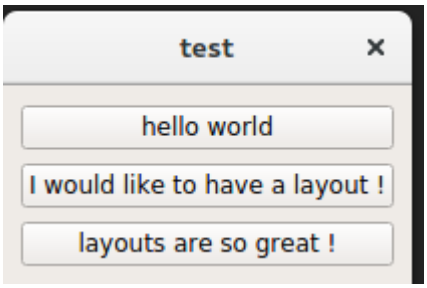
Layout verticale di base

Il layout verticale imposta l'oggetto al suo interno in verticale.

```
#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();
}
```

produzione:



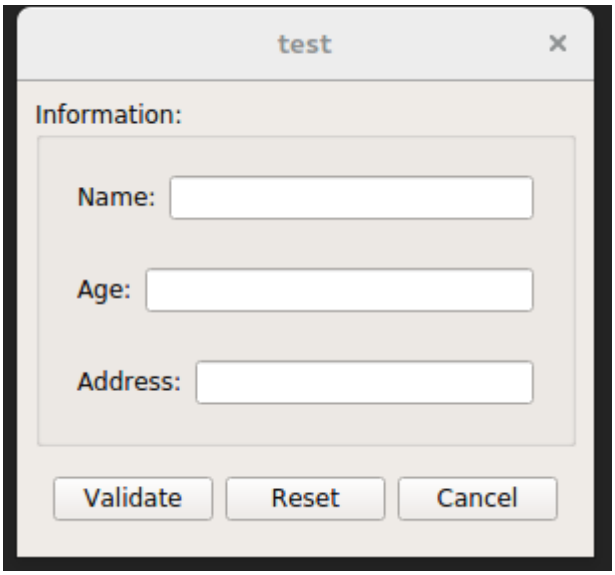
Combinazione di layout

Puoi combinare il layout multiplo grazie ad altri QWidgets nel tuo layout principale per fare effetti più specifici come un campo di informazione: ad esempio:

```
#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();
}
```

produrrà:



Esempio di layout della griglia

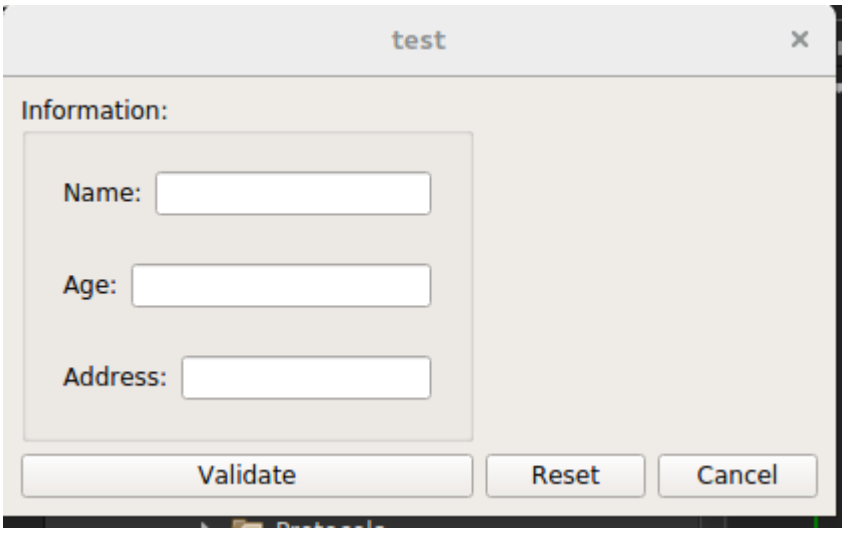
Il layout della griglia è un layout potente con cui è possibile eseguire una volta una disposizione orizzontale e verticale.

esempio:

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

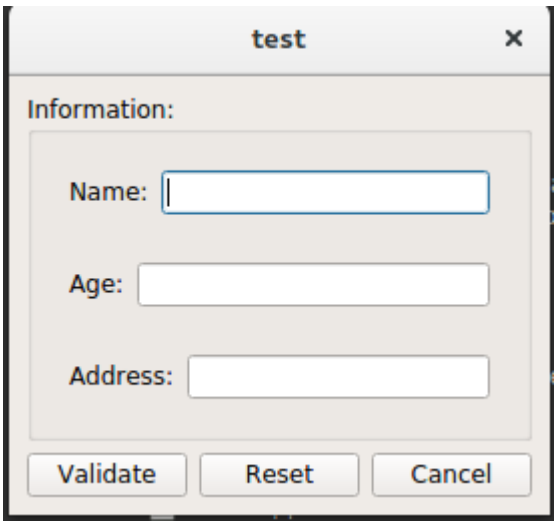
```
#include <QApplication>

#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();
}
```

dare:



quindi puoi vedere che la casella di gruppo è solo nella prima colonna e prima riga come addWidget era `layout->addWidget(box, 0, 0);`

Tuttavia, se lo si modifica in `layout->addWidget(box, 0, 0, 1, 3);` , i nuovi 0 e 3 rappresentano il numero di linee e colonne che vuoi per il tuo widget, così da:



esattamente come hai creato un layout orizzontale e quindi verticale in una subwidget.

Leggi Informazioni sull'uso di layout, genitorialità di widget online:
https://riptutorial.com/it/qt/topic/9380/informazioni-sull-uso-di-layout--genitorialita-di-widget

Capitolo 9: Insidie comuni

Examples

Usando Qt: DirectConnection quando l'oggetto ricevente non riceve il segnale

Alcune volte si vede un segnale emesso nella sequenza del mittente, ma lo slot collegato non viene chiamato (in altre parole non riceve il segnale), lo si è chiesto e finalmente si è capito che il tipo di connessione Qt :: DirectConnection lo avrebbe risolto, quindi il problema è stato riscontrato e tutto è a posto.

Ma in generale questa è una pessima idea usare Qt: DirectConnection fino a quando non si sa veramente cos'è questo e non c'è altro modo. Lo spieghiamo di più, Ogni thread creato da Qt (incluso il thread principale e i nuovi thread creati da QThread) ha un loop degli eventi, il loop degli eventi è responsabile della ricezione dei segnali e chiama gli slot aproporiate nella sua discussione. Generalmente l'esecuzione di un'operazione di blocco all'interno di uno slot è una cattiva pratica, perché blocca il loop degli eventi di tali thread in modo che non vengano chiamati altri slot.

Se si blocca un ciclo di eventi (effettuando un'operazione molto lunga o bloccante) non si riceveranno eventi su quel thread fino a quando il ciclo degli eventi non verrà sbloccato. Se l'operazione di blocco blocca il ciclo di eventi per sempre (come occupato mentre), gli slot non potranno mai essere chiamati.

In questa situazione è possibile impostare il tipo di connessione in connessione a Qt :: DirectConnection, ora gli slot verranno chiamati anche se il loop eventi è bloccato. quindi come questo avrebbe potuto rompere tutto? In Qt :: DirectConnection gli slot verranno chiamati nei thread degli emitter e non nei thread del ricevente e possono rompere le sincronizzazioni dei dati e incorrere in altri problemi. Quindi non usare mai Qt :: DirectConnection se non sai cosa stai facendo. Se il tuo problema verrà risolto usando Qt :: DirectConnection, dovrai fare attenzione e guardare il tuo codice e scoprire perché il tuo ciclo degli eventi è bloccato. Non è una buona idea bloccare il ciclo degli eventi e non è raccomandato in Qt.

Ecco un piccolo esempio che mostra il problema, come si può vedere il nonBlockingSlot sarebbe chiamato anche il ciclo di eventi blockingSlot bloccato con while (1) che indica una codifica errata

```
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();
}
```

Leggi Insidie comuni online: https://riptutorial.com/it/qt/topic/8238/insidie---comuni

Capitolo 10: Intestazione su QListView

introduzione

Il widget QListView fa parte dei meccanismi di programmazione Model / View di Qt. Fondamentalmente, consente di visualizzare gli elementi memorizzati in un modello sotto forma di elenco. In questo argomento non approfondiremo i meccanismi Model / View di Qt, ma ci concentreremo piuttosto sull'aspetto grafico di un widget View: il QListView, e specialmente come aggiungere un'intestazione sopra questo oggetto attraverso l'uso di QPaintEvent oggetto.

Examples

Dichiarazione QListView personalizzata

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

Implementazione del QListView personalizzato

```
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 d'uso: dichiarazione MainWindow

```
class MainMenuListView;

class MainWindow : public QMainWindow
{
    Q_OBJECT

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

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

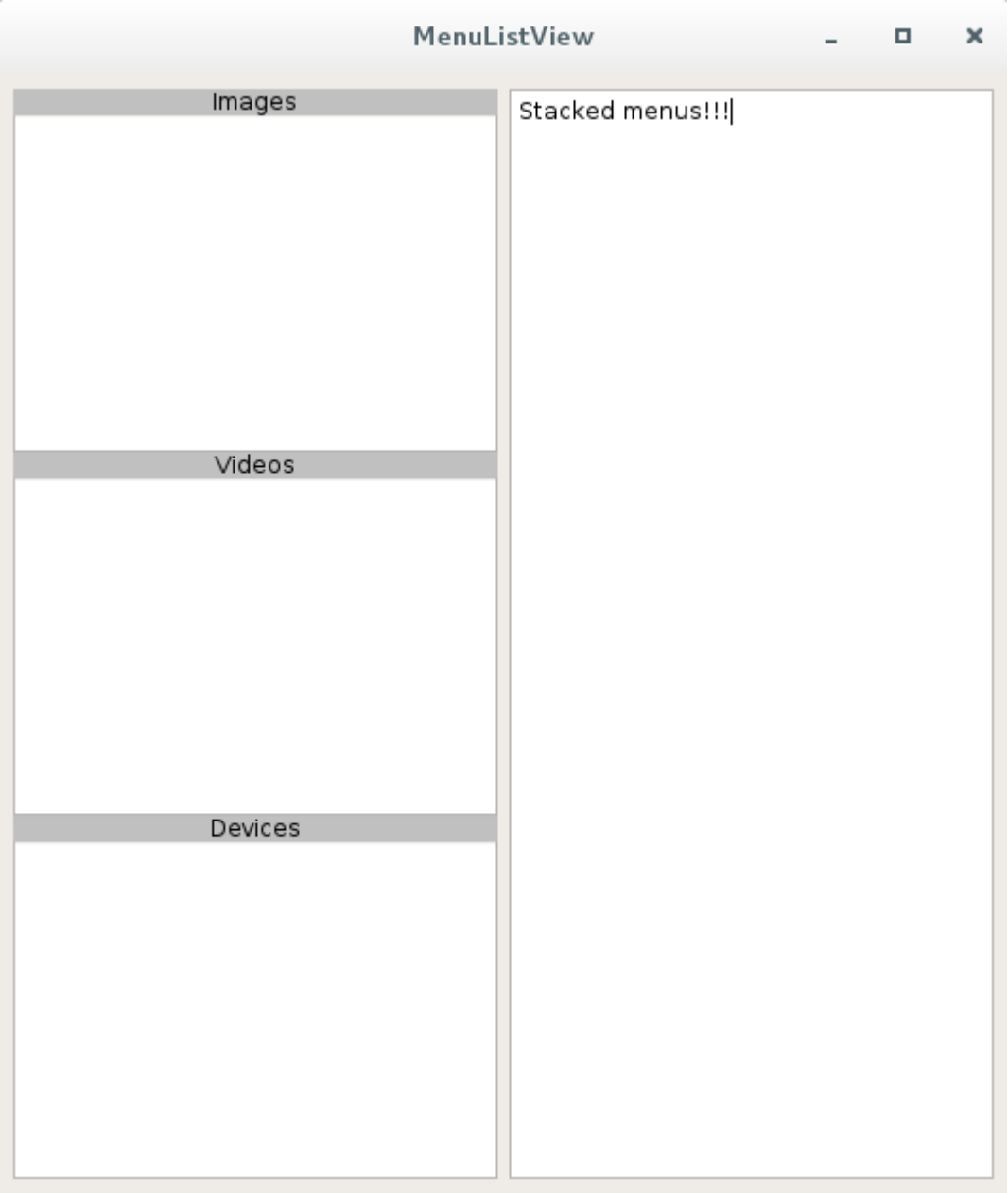
Caso d'uso: implementazione

```
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 d'uso: uscita di esempio

Ecco un esempio di output:



Come puoi vedere sopra, può essere utile per creare menu impilati. Si noti che questo esempio è banale. I due widget hanno gli stessi vincoli di dimensione.

Leggi Intestazione su QListView online: https://riptutorial.com/it/qt/topic/9382/intestazione-su-qlistview

Capitolo 11: Model / View

Examples

Una semplice tabella di sola lettura per visualizzare i dati da un modello

Questo è un semplice esempio per visualizzare dati di sola lettura di natura tabulare utilizzando Qt's Model / View Framework . Nello specifico, vengono utilizzati gli `Qt Objects` QAbstractTableModel (sottoclasse in questo esempio) e QTableView .

Le implementazioni dei metodi rowCount () , columnCount () , data () e headerData () sono necessarie per fornire all'oggetto `QTableView` un mezzo per ottenere informazioni sui dati contenuti nell'oggetto `QAbstractTableModel` .

Il metodo `populateData()` stato aggiunto a questo esempio per fornire un mezzo per popolare l'oggetto `QAbstractTableModel` con i dati provenienti da una fonte 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
```

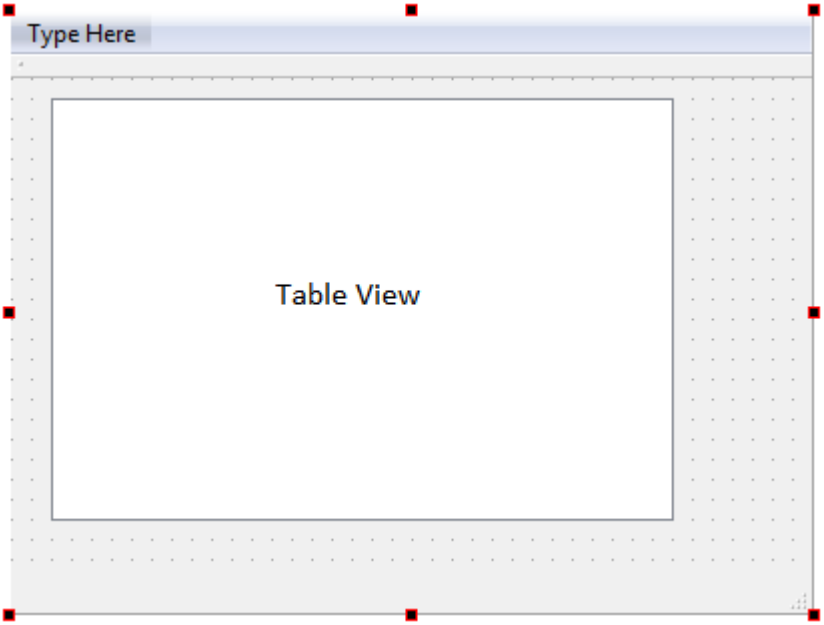
```
public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;

};

#endif // MAINWINDOW_H
```

mainwindow.cpp

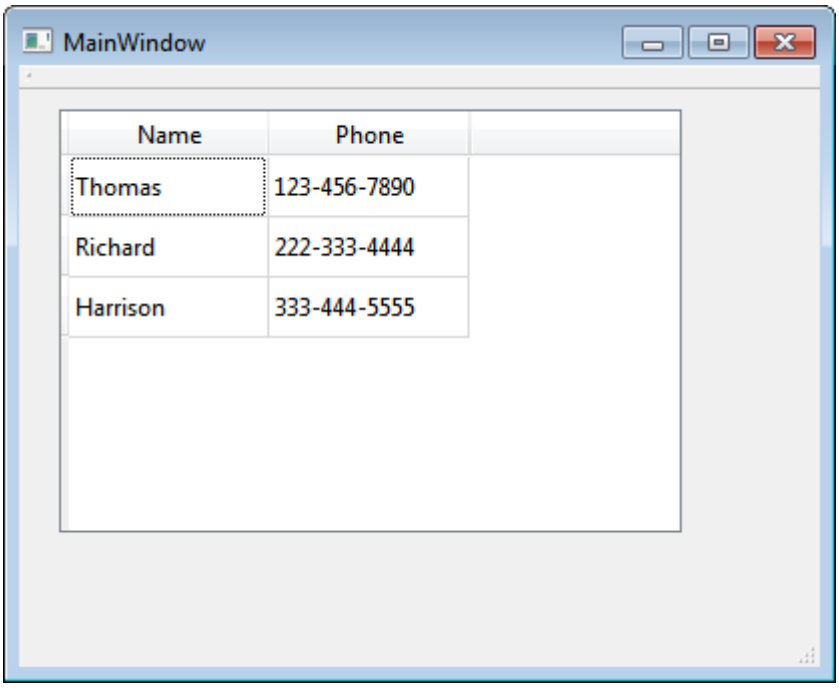
```
#include "mainwindow.h"
#include "ui_mainwindow.h"

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();
}
```

Utilizzando `Qt Creator/Design` , posizionare un oggetto `Table View` , denominato **tableView** in questo esempio, nella **finestra principale** :



Il programma risultante viene visualizzato come:



Un semplice modello ad albero

QModelIndex in realtà non conosce gli indici genitore / figlio, contiene solo una **riga** , una **colonna** e un **puntatore** ed è responsabilità dei modelli utilizzare tali dati per fornire informazioni sulle relazioni dell'indice. Pertanto, il modello deve eseguire molte conversioni dal `void*` memorizzato all'interno di `QModelIndex` a un tipo di dati interno e `QModelIndex` .

TreeModel.h:

```
#pragma once

#include <QAbstractItemModel>

class TreeModel : public QAbstractItemModel
```

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

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

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

private:
    struct Item
    {
        ~Item();

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

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

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

TreeModel.cpp:

```
#include "TreeModel.h"

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Leggi Model / View online: https://riptutorial.com/it/qt/topic/3938/model---view

Capitolo 12: Multimedia

Osservazioni

Qt Multimedia è un modulo che offre la gestione di funzionalità multimediali (audio, video) e anche di telecamere e radio.

Tuttavia, i file supportati di QMediaPlayer dipendono dalla piattaforma. Infatti, su Windows, QMediaPlayer usa DirectShow, su Linux, utilizza GStreamer. Quindi, a seconda della piattaforma, alcuni file potrebbero funzionare su Linux ma non su Windows o il contrario.

Examples

Riproduzione video in Qt 5

Creiamo video player molto semplice usando il modulo QtMultimedia di Qt 5.

Nel file .pro della tua applicazione avrai bisogno delle seguenti linee:

```
QT += multimedia multimediawidgets
```

Si noti che i dispositivi `multimediawidgets` sono necessari per l'utilizzo di `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();
```

Questo è tutto - dopo aver lanciato l'applicazione (se necessario i codec sono installati nel sistema), verrà avviata la riproduzione del file video.

Allo stesso modo in cui puoi riprodurre video da URL in Internet, non solo file locali.

Riproduzione audio in Qt5

Poiché si tratta di un audio, non è necessario un QVideoWidget. Quindi possiamo fare:

```
_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();
```

nella .h:

```
QMediaPlayer *_player;
```

questo aprirà una finestra di dialogo in cui puoi scegliere la tua musica e la riprodurrà.

Leggi Multimedia online: https://riptutorial.com/it/qt/topic/7675/multimedia

Capitolo 13: QDialogs

Osservazioni

La classe QDialog è la classe **base** delle finestre di dialogo. Una finestra di dialogo è una finestra di livello superiore utilizzata principalmente per attività a breve termine e comunicazioni brevi con l'utente. QDialogs può essere **modale** o non **modale** .

Si noti che QDialog (e qualsiasi altro widget con tipo Qt :: Dialog) utilizza il widget padre in modo leggermente diverso rispetto alle altre classi in Qt. Una finestra di dialogo è **sempre un widget di primo livello** , ma se **ha un genitore, la sua posizione predefinita è centrata in cima al widget di livello superiore del genitore** (se non è di per sé stesso di livello superiore). Condivide anche la voce della barra delle applicazioni del genitore.

Una finestra di dialogo **modale** è una finestra di dialogo che blocca l'input ad altre finestre visibili nella stessa applicazione. Le finestre di dialogo utilizzate per richiedere un nome file dall'utente o utilizzate per impostare le preferenze dell'applicazione sono in genere modali. Le finestre di dialogo possono essere **modali applicazione** (impostazione predefinita) o **modali finestra** .

Il modo più comune per visualizzare una finestra di dialogo modale è chiamare la sua funzione exec (). Quando l'utente chiude la finestra di dialogo, exec () fornirà un utile valore di ritorno.

Una finestra di dialogo non **modale** è una finestra di dialogo che opera indipendentemente dalle altre finestre nella stessa applicazione. Le finestre di dialogo non modali vengono visualizzate usando show (), che restituisce immediatamente il controllo al chiamante.

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>
```

Leggi QDialogs online: https://riptutorial.com/it/qt/topic/7819/qdialogs

Capitolo 14: QGraphics

Examples

Panoramica, zoom e rotazione con QGraphicsView

`QGraphics` può essere utilizzato per organizzare scene complicate di oggetti visivi in una struttura che li rende più facili da gestire.

Esistono tre tipi principali di oggetti utilizzati in questo framework QGraphicsView , QGraphicsScene e QGraphicsItems . QGraphicsItem sono gli elementi visivi di base che esistono nella scena.

Ci sono molti tipi che sono pre-costruiti e possono essere usati come Ellissi , Linee , Percorsi , Pixmap , Poligoni , Rettangoli e Testo .

Puoi anche creare i tuoi oggetti ereditando `QGraphicsItem` . Questi elementi vengono quindi inseriti in una `QGraphicsScene` che è fondamentalmente il mondo che stai pensando di guardare. Gli oggetti possono muoversi all'interno della scena che è come farli muovere nel mondo che stai guardando. Il posizionamento e l'orientamento degli oggetti è gestito da matrici di trasformazione denominate QTransforms . Qt ha delle belle funzioni incorporate, quindi di solito non è necessario lavorare con `QTransforms` direttamente, ma chiamate funzioni come rotate o scale che creano le trasformazioni appropriate per voi. La scena viene quindi visualizzata dalla prospettiva definita in `QGraphicsView` (sempre con `QTransforms`), che è il pezzo che inseriresti in un widget nell'interfaccia utente dell'utente.

Nell'esempio seguente c'è una scena molto semplice con un solo oggetto (una pixmap), che viene inserito in una scena e visualizzato in una vista. `DragMode` flag `DragMode` , la scena può essere ruotata attorno al mouse e usando la scala e ruotando le funzioni può essere ridimensionata dentro e fuori con lo scroll del mouse e ruotata con i tasti freccia.

Se si desidera eseguire questo esempio, creare un'istanza di Vista che verrà visualizzata e creare un file di risorse con il prefisso / le immagini che contengono un'immagine 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);
  }
};
```

Leggi QGraphics online: https://riptutorial.com/it/qt/topic/7539/qgraphics

Capitolo 15: qmake

Examples

Profilo base.

qmake è uno strumento di automazione build, fornito con framework *Qt* . Funziona in modo simile a strumenti come *CMake* o *GNU Autotools* , ma è progettato per essere utilizzato specificamente con *Qt* . Come tale è ben integrato con l'ecosistema *Qt* , in particolare *Qt Creator* IDE.

Se avvii *Qt Creator* e selezioni `File -> New File or Project -> Application -> Qt Widgets` , *Qt Creator* genererà per te un progetto scheletro insieme a un file "pro". Il file "pro" viene elaborato da *qmake* per generare file, che vengono a loro volta elaborati dai sistemi di compilazione sottostanti (ad esempio *GNU Make* o *nmake*).

Se hai chiamato il tuo progetto "myapp", apparirà il file "myapp.pro". Ecco come appare questo file predefinito, con commenti, che descrivono ogni variabile *qmake* , aggiunta.

```
# 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
```

Conservazione della struttura della directory di origine in una build (opzione "object_parallel_to_source" non documentata).

Se ti piace organizzare il tuo progetto mantenendo i file di origine in diverse sottodirectory, dovresti sapere che durante una generazione *qmake* non preserverà questa struttura di directory e manterrà tutti i file ".o" in una singola directory di compilazione. Questo può essere un problema se avessi nomi di file in conflitto in diverse directory come segue.

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

Ora *qmake* deciderà di creare due file "file1.o" in una directory di build, facendo in modo che uno di essi venga sovrascritto da un altro. Il buld fallirà. Per evitare ciò, puoi aggiungere l'opzione di configurazione `CONFIG += object_parallel_to_source` al tuo file "pro". Questo dirà a *qmake* di generare file di build che preservano la struttura della directory di origine. In questo modo la directory di costruzione rifletterà la struttura della directory di origine e i file oggetto verranno creati in sottodirectory separate.

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

Esempio 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
```

Nota che l'opzione `CONFIG object_parallel_to_source` **non** è **documentata ufficialmente** .

Semplice esempio (Linux)

Window.h

```
#include <QWidget>

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

main.cpp

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

int main()
{
    QApplication app;
    Window window;
    window.show();
    return app.exec();
}
```

example.pro

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

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

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

Riga di 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.
```

Esempio di SUBDIRS

L'abilità SUBDIRS di qmake può essere utilizzata per compilare un set di librerie, ognuna delle quali dipende da un'altra. L'esempio sotto è leggermente contorto per mostrare variazioni con l'abilità SUBDIRS.

Struttura della directory

Alcuni dei seguenti file saranno omessi nell'interesse della brevità. Si può presumere che siano il formato come esempi non-subdir.

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

project.pro

Questo è il file principale che abilita l'esempio. Questo è anche il file che verrebbe chiamato con qmake sulla riga di comando (vedi sotto).

```
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
```

(*) Vedere la documentazione di riferimento per le altre opzioni per un obiettivo di sottodirectory.

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
```

logica / logic.pro

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

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

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

gui / gui.pro

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

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

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

build.pro

```
TEMPLATE = app

SOURCES += main.cpp

LIBS += -Llogic -Lgui -llogic -lgui

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

Riga di 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.
```

Esempio di libreria

Un semplice esempio per creare una libreria (piuttosto che un eseguibile, che è l'impostazione predefinita). `TEMPLATE` variabile `TEMPLATE` specifica il tipo del progetto che stai creando. `lib` opzione `lib` consente a makefile di creare una libreria.

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
```

Quando si `staticlib` una libreria, è possibile aggiungere le opzioni `dll` (default), `staticlib` o `plugin` a `CONFIG` .

Creazione di un file di progetto dal codice esistente

Se hai una directory con i file sorgente esistenti, puoi usare `qmake` con `-project` -option per creare un file di progetto.

Supponiamo che la cartella *MyProgram* contenga i seguenti file:

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

Quindi chiamando

```
qmake -project
```

un file *MyProgram.pro* viene creato con il seguente contenuto:

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

Il codice può quindi essere costruito come descritto in questo semplice esempio .

Leggi qmake online: https://riptutorial.com/it/qt/topic/4438/qmake

Capitolo 16: QObject

Osservazioni

`QObject` classe `QObject` è la classe base per tutti gli oggetti Qt.

Examples

Esempio di QObject

`Q_OBJECT` macro `Q_OBJECT` viene visualizzata nella sezione privata di una classe. `Q_OBJECT` richiede che la classe diventi sottoclasse di `QObject` . Questa macro è necessaria affinché la classe dichiari i suoi segnali / slot e utilizzi il sistema meta-oggetto Qt.

Se Meta Object Compiler (MOC) trova la classe con `Q_OBJECT` , la elabora e genera il file sorgente C ++ contenente il codice sorgente del meta oggetto.

Ecco l'esempio dell'intestazione della classe con `Q_OBJECT` e signal / slots:

```
#include <QObject>

class MyClass : public QObject
{
    Q_OBJECT

public:

public slots:
    void setNumber(double number);

signals:
    void numberChanged(double number);

private:
}
```

qobject_cast

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

Una funzionalità che viene aggiunta derivando da `QObject` e utilizzando la macro `Q_OBJECT` è la possibilità di utilizzare `qobject_cast` .

Esempio:

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

```
};

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

Per verificare se `obj` è un tipo `myObject` e per lanciarlo in tale in C ++ puoi generalmente usare un `dynamic_cast` . Questo dipende dall'avere RTTI abilitato durante la compilazione.

La macro Q_OBJECT d'altra parte genera i controlli di conversione e il codice che possono essere utilizzati in qobject_cast.

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

Questo non dipende da RTTI. Inoltre, consente di eseguire il cast attraverso i confini delle librerie dinamiche (tramite interfacce / plug-in Qt).

QObject Lifetime and Ownership

QObjects viene fornito con il proprio concetto di durata alternativa rispetto ai puntatori grezzi, univoci o condivisi del C ++ nativo.

QObjects ha la possibilità di creare un oggetto object dichiarando le relazioni genitore / figlio.

Il modo più semplice per dichiarare questa relazione è passare l'oggetto genitore nel costruttore. Come alternativa puoi impostare manualmente il genitore di un `QObject` chiamando `setParent` . Questa è l'unica direzione per dichiarare questa relazione. Non è possibile aggiungere un figlio a una classe di genitori, ma solo il contrario.

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

Quando il `parent` ora viene eliminato nel `child` stack-unwind, verrà eliminato anche.

Quando cancelliamo un oggetto `QObject` esso "annullerà la registrazione" di per sé dall'oggetto padre;

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

Lo stesso vale per le variabili dello stack:

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

`child` verrà cancellato prima del `parent` durante lo sbobinamento e si annullerà la registrazione dal genitore.

Nota: è possibile chiamare manualmente `setParent` con un ordine inverso di dichiarazione che **romperà** la distruzione automatica.

Leggi QObject online: https://riptutorial.com/it/qt/topic/6304/qobject

Capitolo 17: Qt - Gestire i database

Osservazioni

- Avrai bisogno del plugin Qt SQL corrispondente al tipo dato a `QSqlDatabase::addDatabase`
- Se non si dispone del plug-in SQL richiesto, Qt ti avviserà che non è possibile trovare il driver richiesto
- Se non si dispone del plug-in SQL richiesto, sarà necessario compilare i file dal sorgente Qt

Examples

Utilizzando un database su Qt

Nel file Project.pro aggiungiamo:

```
CONFIG += sql
```

in MainWindow.h scriviamo:

```
#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;
};
```

Ora in 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 - Gestire i database Sqlite

Nel file Project.pro aggiungiamo: `CONFIG += sql`

in MainWindow.h scriviamo:

```
#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;
};
```

Ora in 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 - Gestire i database ODBC

Nel file Project.pro aggiungiamo: `CONFIG += sql`

in MainWindow.h scriviamo:

```
#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;
};
```

Ora in 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 - Gestire i database Sqlite in memoria

Nel file Project.pro aggiungiamo: `CONFIG += sql`

in MainWindow.h scriviamo:

```
#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;
};
```

Ora in 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;
}
```

Rimuovere correttamente la connessione al database

Se vogliamo rimuovere alcune connessioni al database dall'elenco delle connessioni del database. abbiamo bisogno di usare `QSqlDatabase::removeDatabase()` , tuttavia è una funzione statica e il modo in cui funziona è un po 'cablato.

```
// 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
```

Di seguito è riportato il modo corretto che il documento Qt ci suggerisce.

```
  {
      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
```

Leggi Qt - Gestire i database online: https://riptutorial.com/it/qt/topic/1993/qt---gestire-i-database

Capitolo 18: Qt Network

introduzione

Qt Network fornisce strumenti per utilizzare facilmente molti protocolli di rete nella tua applicazione.

Examples

Client TCP

Per creare una connessione **TCP** in Qt, useremo QTcpSocket . Per prima cosa, dobbiamo connetterci con `connectToHost` .

Ad esempio, per connettersi a un `_socket.connectToHost(QHostAddress("127.0.0.1"), 4242);` tcp locale: `_socket.connectToHost(QHostAddress("127.0.0.1"), 4242);`

Quindi, se abbiamo bisogno di leggere i dati dal server, abbiamo bisogno di collegare il segnale readyRead con uno slot. Come quello:

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

e infine, possiamo leggere i dati in questo modo:

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

Per scrivere i dati, è possibile utilizzare il metodo `write(QByteArray)` :

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

Quindi un client TCP di base può apparire così:

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: (vuoto qui)

```
<?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>
```

Server TCP

Creare un **server TCP** in Qt è anche molto semplice, infatti la classe QTcpServer fornisce già tutto ciò di cui abbiamo bisogno per fare il server.

Per prima cosa, dobbiamo ascoltare qualsiasi ip, una porta casuale e fare qualcosa quando un client è connesso. come quello:

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

Quindi, quando questa è una nuova connessione, possiamo aggiungerla all'elenco dei client e prepararci a leggere / scrivere sul socket. Come quello:

```
   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);
```

Lo `stateChanged(QAbstractSocket::SocketState)` ci consente di rimuovere il socket dalla nostra lista quando il client è disconnesso.

Quindi qui un server di chat di base:

main.cpp:

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

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

    return a.exec();
}
```

mainwindow.h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

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

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

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

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

```
};

#endif // MAINWINDOW_H
```

mainwindow.cpp:

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

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

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

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

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

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

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

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

```
}
```

(usa lo stesso mainwindow.ui dell'esempio precedente)

Leggi Qt Network online: https://riptutorial.com/it/qt/topic/9683/qt-network

Capitolo 19: Qt Resource System

introduzione

Il sistema Qt Resource è un modo per incorporare i file all'interno del tuo progetto. Ogni file di risorse può avere uno o più *prefissi* e ogni *prefisso* può contenere file.

Ogni file nelle risorse è un collegamento a un file sul file system. Quando viene creato l'eseguibile, i file vengono raggruppati nell'eseguibile, quindi non è necessario distribuire il file originale con il file binario.

Examples

Riferimento a file all'interno del codice

Diciamo che all'interno di un file di risorse, avevi un file chiamato /icons/ok.png

L'URL completo di questo file nel codice è `qrc:/icons/ok.png` . Nella maggior parte dei casi, questo può essere abbreviato in `:/icons/ok.png`

Ad esempio, se si desidera creare un QIcon e impostarlo come l'icona di un pulsante da quel file, è possibile utilizzarlo

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

Leggi Qt Resource System online: https://riptutorial.com/it/qt/topic/8776/qt-resource-system

Capitolo 20: QTimer

Osservazioni

QTimer può anche essere utilizzato per richiedere una funzione da eseguire non appena il ciclo degli eventi ha elaborato tutti gli altri eventi in sospeso. Per fare ciò, utilizzare un intervallo di 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

Semplice esempio

L'esempio seguente mostra come utilizzare un `QTimer` per chiamare uno slot ogni 1 secondo.

Nell'esempio, utilizziamo un `QProgressBar` per aggiornare il suo valore e controllare che il timer `QProgressBar` correttamente.

main.cpp

```
#include <QApplication>

#include "timer.h"

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

    Timer timer;
    timer.show();

    return app.exec();
}
```

timer.h

```
#ifndef TIMER_H
#define TIMER_H
```

```
#include <QWidget>

class QProgressBar;

class Timer : public QWidget
{
    Q_OBJECT

public:
    Timer(QWidget *parent = 0);

public slots:
    void updateProgress();

private:
    QProgressBar *progressBar;
};

#endif
```

timer.cpp

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

#include "timer.h"

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

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

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

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

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

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

timer.pro

```
QT += widgets

HEADERS = \
          timer.h
```

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

Singleshot Timer con funzione Lambda come slot

Se è richiesto un timer singolo, è silenzioso avere lo slot come funzione lambda nel punto in cui è stato dichiarato il timer:

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

A causa di questo bug (QTBUG-26406) , questo è il modo è possibile solo dal momento che Qt5.4.

Nelle versioni precedenti di Qt5 deve essere fatto con più codice della piastra della caldaia:

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

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

Usando QTimer per eseguire il codice sul thread principale

```
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));
}
```

Questo è utile quando devi aggiornare un elemento dell'interfaccia utente da un thread. Tieni a mente la durata di tutto ciò che fa riferimento alla richiamata.

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

Lo stesso codice potrebbe essere adattato per eseguire codice su qualsiasi thread che esegue il ciclo di eventi Qt, implementando così un semplice meccanismo di invio.

Uso di base

`QTimer` aggiunge la funzionalità per avere una funzione / slot specifica chiamata dopo un certo intervallo (ripetutamente o solo una volta).

`QTimer` consente quindi a un'applicazione GUI di "controllare" le cose regolarmente o gestire i timeout **senza** dover avviare manualmente un thread aggiuntivo per questo e fare attenzione alle condizioni di gara, perché il timer verrà gestito nel ciclo dell'evento principale.

Un timer può essere semplicemente usato in questo modo:

```
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
```

Il timer attiva il segnale di `timeout` quando il tempo è scaduto e questo verrà chiamato nel ciclo dell'evento principale.

QTimer :: singleShot semplice utilizzo

Il **QTimer :: singleShot** viene utilizzato per chiamare uno slot / lambda in **modo asincrono** dopo n ms.

La sintassi di base è:

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

con **myTime** il tempo in ms, **myObject** l'oggetto che contiene il metodo e **myMethodInMyObject** lo slot da chiamare

Quindi per esempio se vuoi avere un timer che scrive una riga di debug "ciao!" ogni 5 secondi:

cpp

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

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

.hh

```
class MyObject : public QObject {
    Q_OBJECT
    ...
```

```
    void startHelloWave();

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

Leggi QTimer online: https://riptutorial.com/it/qt/topic/4309/qtimer

Capitolo 21: Segnali e slot

introduzione

Segnali e slot sono utilizzati per la comunicazione tra oggetti. Il meccanismo di segnali e slot è una caratteristica centrale di Qt. Nella programmazione GUI, quando cambiamo un widget, spesso vogliamo che venga notificato un altro widget. Più in generale, vogliamo che oggetti di qualsiasi tipo siano in grado di comunicare tra loro. I segnali vengono emessi dagli oggetti quando cambiano il loro stato in un modo che potrebbe essere interessante per altri oggetti. Gli slot possono essere utilizzati per ricevere segnali, ma sono anche normali funzioni membro.

Osservazioni

La documentazione ufficiale su questo argomento può essere trovata qui .

Examples

Un piccolo esempio

Segnali e slot sono utilizzati per la comunicazione tra oggetti. Il meccanismo di segnali e slot è una caratteristica centrale di Qt e probabilmente la parte che differisce maggiormente dalle funzionalità fornite da altri framework.

L'esempio minimo richiede una classe con un segnale, uno slot e una connessione:

counter.h

```
#ifndef COUNTER_H
#define COUNTER_H

#include <QWidget>
#include <QDebug>

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

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

            /*
             * The most important line: connect the signal to the slot.
             */
```

```
            connect(this, &Counter::valueChanged, this, &Counter::printvalue);
    }

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

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

signals:
    void valueChanged(int newValue);

private:
    int m_value;

};

#endif
```

Il `main` imposta un nuovo valore. Possiamo controllare come viene chiamato lo slot, stampando il valore.

```
#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();
}
```

Infine, il nostro file di progetto:

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

La nuova sintassi per la connessione Qt5

La sintassi di `connect` convenzionale che utilizza le macro `SIGNAL` e `SLOT` funziona interamente in fase di runtime, che presenta due inconvenienti: presenta alcuni sovraccarichi di runtime (che si verificano anche in overhead di dimensioni binarie) e non esiste un controllo di correttezza in fase di compilazione. La nuova sintassi affronta entrambi i problemi. Prima di controllare la sintassi in un esempio, faremmo meglio a sapere cosa succede in particolare.

Diciamo che stiamo costruendo una casa e vogliamo collegare i cavi. Questo è esattamente ciò che fa la funzione di connessione. Segnali e slot sono quelli che necessitano di questa connessione. Il punto è che se si effettua una connessione, è necessario fare attenzione alle ulteriori connessioni di sovrapposizione. Ogni volta che si collega un segnale a uno slot, si sta tentando di dire al compilatore che ogni volta che viene emesso il segnale, basta richiamare la funzione dello slot. Questo è esattamente ciò che accade.

Ecco un esempio di **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();
}
```

Suggerimento: la vecchia sintassi (macro `SIGNAL` / `SLOT`) richiede che il metacompiler Qt (MOC) venga eseguito per qualsiasi classe che abbia uno slot o un segnale. Dal punto di vista della codifica ciò significa che tali classi devono avere la macro `Q_OBJECT` (che indica la necessità di eseguire MOC su questa classe).

La nuova sintassi, d'altra parte, richiede ancora MOC per far funzionare i segnali, ma **non** per gli slot. Se una classe ha solo slot e nessun segnale, non ha bisogno della macro `Q_OBJECT` e quindi potrebbe non invocare il MOC, che non solo riduce la dimensione binaria finale ma riduce anche il tempo di compilazione (nessuna chiamata MOC e nessuna successiva chiamata del compilatore per il generato `*_moc.cpp` file `*_moc.cpp`).

Collegamento di segnali / slot sovraccarichi

Pur essendo migliore sotto molti aspetti, la nuova sintassi della connessione in Qt5 presenta una grande debolezza: connessione di segnali e slot sovraccaricati. Per consentire al compilatore di risolvere i sovraccarichi, è necessario utilizzare `static_cast` s per i puntatori di funzioni dei membri o (a partire da Qt 5.7) `qOverload` e amici:

```
#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
}
```

Connessione slot segnale multi finestra

Un semplice esempio multiwindow che utilizza segnali e slot.

Esiste una classe MainWindow che controlla la vista della finestra principale. Una seconda finestra controllata dalla classe del sito web.

Le due classi sono collegate in modo che quando si fa clic su un pulsante nella finestra del sito Web, qualcosa accade nella finestra principale (un'etichetta di testo viene modificata).

Ho fatto un semplice esempio che è anche su 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();
}
```

Composizione del progetto:

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

HEADERS  += mainwindow.h \
    website.h

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

Considera l'Uis da comporre:

- Finestra principale: un'etichetta chiamata "testo" e un pulsante chiamato "openButton"
- Finestra del sito Web: un pulsante chiamato "changeButton"

Quindi i punti chiave sono le connessioni tra segnali e slot e la gestione dei puntatori o riferimenti di Windows.

Leggi Segnali e slot online: https://riptutorial.com/it/qt/topic/2136/segnali-e-slot

Capitolo 22: SQL su Qt

Examples

Connessione di base e query

La classe QSqlDatabase fornisce un'interfaccia per l'accesso a un database tramite una connessione. Un'istanza di QSqlDatabase rappresenta la connessione. La connessione fornisce l'accesso al database tramite uno dei driver di database supportati. Assicurati di aggiungere

```
QT += SQL
```

nel file .pro. Si supponga che un DB SQL denominato TestDB con un countryTable contine la colonna successiva:

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

Per interrogare e ottenere dati SQL da 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();
}
```

Parametri di query Qt SQL

Spesso è conveniente separare la query SQL dai valori effettivi. Questo può essere fatto usando i segnaposti. Qt supporta due sintassi segnaposto: binding con nome e binding posizionale.

named binding:

```
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();
```

vincolo posizionale:

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

Si noti che prima di chiamare `bindValue()` o `addBindValue()` è necessario chiamare QSqlQuery :: prepare () una volta.

Connessione al database MS SQL Server tramite QODBC

Quando si tenta di aprire una connessione al database con QODBC, assicurarsi

- È disponibile il driver QODBC
- Il tuo server ha un'interfaccia ODBC ed è abilitato (dipende dalle installazioni del tuo driver ODBC)
- utilizzare l'accesso alla memoria condivisa, connessioni TCP / IP o named pipe connection.

Tutte le connessioni richiedono solo che DatabaseName sia impostato chiamando QSqlDatabase :: setDatabaseName.

Apri connessione utilizzando l'accesso alla memoria condivisa

Affinché questa opzione funzioni, è necessario avere accesso alla memoria della macchina e disporre delle autorizzazioni per accedere alla memoria condivisa. Per utilizzare una connessione di memoria condivisa è necessario impostare lpc: davanti alla stringa del server. La connessione che utilizza SQL Server Native Client 11 viene effettuata utilizzando questi passaggi:

```
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");
}
```

Apri connessione usando Named Pipe

Questa opzione richiede che la tua connessione ODBC abbia un DSN completo. La stringa del server viene configurata utilizzando il nomecomputer di Windows e il nome istanza di SQL Server. La connessione di esempio verrà aperta utilizzando 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");
}
```

Apri connessione tramite TCP / IP

Per aprire una connessione TCP / IP il server deve essere configurato per consentire le connessioni su una porta fissa, altrimenti sarà necessario prima eseguire una query per la porta attualmente attiva. In questo esempio abbiamo una porta fissa su 5171. È possibile trovare un esempio per configurare il server per consentire le connessioni su una porta fissa su 1 . Per aprire una connessione utilizzando TCP / IP utilizzare una tupla dei server IP e Porta:

```
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");
}
```

Leggi SQL su Qt online: https://riptutorial.com/it/qt/topic/10628/sql-su-qt

Capitolo 23: Threading e concorrenza

Osservazioni

Alcune note che sono già menzionate nei documenti ufficiali qui e qui :

- Se un oggetto ha un genitore, deve essere nello stesso thread del genitore, cioè non può essere spostato in un nuovo thread, né è possibile impostare un genitore su un oggetto se il genitore e l'oggetto vivono in thread diversi
- Quando un oggetto viene spostato su un nuovo thread, anche tutti i relativi figli vengono spostati nel nuovo thread
- Puoi solo *spostare* oggetti su un nuovo thread. Non puoi *trascinarli* su un nuovo thread, ovvero puoi solo `moveToThread` dal thread in cui l'oggetto sta attualmente vivendo

Examples

Utilizzo di base di QThread

`QThread` è un handle per un thread di piattaforma. Ti consente di gestire il thread monitorandone la durata e richiedendo il completamento del lavoro.

Nella maggior parte dei casi, l'ereditarietà dalla classe non è raccomandata. Il metodo di `run` predefinito avvia un ciclo di eventi che può inviare eventi agli oggetti che vivono nella classe. Le connessioni dello slot del segnale cross-thread vengono implementate `QMetaCallEvent` un oggetto `QMetaCallEvent` all'oggetto target.

Un'istanza `QObject` può essere spostata su un thread, dove elaborerà i suoi eventi, come eventi timer o chiamate slot / metodo.

Per eseguire il lavoro su un thread, innanzitutto creare la propria classe worker che deriva da `QObject` . Quindi spostalo nella discussione. L'oggetto può eseguire automaticamente il proprio codice, ad esempio utilizzando `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);
    }
};
```

Se il tuo operatore deve essere effimero e esistere solo mentre il suo lavoro è in esecuzione, è meglio inviare un metodo o un metodo thread-safe per l'esecuzione nel pool di thread tramite `QtConcurrent::run` .

QtConcurrent Run

Se trovi la gestione di QThread e di primitive di basso livello come mutex o semafori troppo complessi, lo spazio dei nomi simultaneo di Qt è quello che stai cercando. Include classi che consentono una gestione dei thread di alto livello.

Diamo un'occhiata a Concurrent Run. `QtConcurrent::run()` consente di eseguire la funzione in un nuovo thread. Quando vorresti utilizzarlo? Quando hai qualche operazione lunga e non vuoi creare il thread manualmente.

Ora il codice:

```
#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();
}
```

Quindi le cose sono semplici: quando abbiamo bisogno di eseguire un'altra funzione in un altro thread, basta chiamare `QtConcurrent::run` , passare la funzione ei suoi parametri e il gioco è fatto!

`QFuture` presenta il risultato del nostro calcolo asincrono. In caso di `QtConcurrent::run` non

possiamo annullare l'esecuzione della funzione.

Richiamo di slot da altri thread

Quando un ciclo di eventi Qt viene utilizzato per eseguire operazioni e un utente non-Qt-saavy deve interagire con quel ciclo di eventi, scrivere lo slot per gestire le chiamate regolari da un altro thread può semplificare le cose per gli altri utenti.

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
```

Leggi Threading e concorrenza online: https://riptutorial.com/it/qt/topic/5022/threading-e-concorrenza

Capitolo 24: Utilizzo di fogli di stile in modo efficace

Examples

Impostazione del foglio di stile del widget dell'interfaccia utente

È possibile impostare il foglio di stile del widget dell'interfaccia utente desiderato utilizzando qualsiasi CSS valido. L'esempio seguente imposta un colore del testo di QLabel attorno ad esso.

```
#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;
}
```

Leggi Utilizzo di fogli di stile in modo efficace online: https://riptutorial.com/it/qt/topic/5931/utilizzo-di-fogli-di-stile-in-modo-efficace

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Qt	agilob, Christopher Aldama, Community, demonplus, devbean, Dmitriy, Donald Duck, fat, Gabriel de Grimouard, Kamalpreet Grewal, Maxito, Tarod, thiagofalcao
2	Classi di contenitori Qt	demonplus, Tarod
3	CMakeLists.txt per il tuo progetto Qt	Athena, demonplus, Robert, Velkan, wasthishelpful
4	Comunicazione tra QML e C ++	Gabriel de Grimouard, Martin Zhai
5	Condivisione implicita	Hayt
6	Costruisci QtWebEngine dal sorgente	Martin Zhai
7	Distribuzione di applicazioni Qt	Luca Angioloni, Martin Zhai, Nathan Osman, TriskalJM, wasthishelpful
8	Informazioni sull'uso di layout, genitorialità di widget	Gabriel de Grimouard
9	Insidie comuni	e.jahandar
10	Intestazione su QListView	Papipone
11	Model / View	Jan, KernelPanic, Tim D
12	Multimedia	demonplus, Gabriel de Grimouard
13	QDialogs	Wilmort
14	QGraphics	Chris, demonplus
15	qmake	Caleb Huitt - cjhuitt, demonplus, doc, Gregor, Jon Harper
16	QObject	demonplus, Hayt

17	Qt - Gestire i database	Jan, Rinat, Shihe Zhang, Zylva
18	Qt Network	Gabriel de Grimouard
19	Qt Resource System	Victor Tran
20	QTimer	avb, Caleb Huitt - cjhuitt, Eugene, Gabriel de Grimouard, Hayt, Rinat, Tarod, thuga, tpr, Victor Tran
21	Segnali e slot	Athena, devbean, fat, immerhart, Jan, Luca Angioloni, Robert, Tarod, Violet Giraffe
22	SQL su Qt	Noam M
23	Threading e concorrenza	demonplus, gmabey, Kuba Ober, Nathan Osman, RamenChef, thuga
24	Utilizzo di fogli di stile in modo efficace	Nicholas Johnson