Qt Programmierung – Teil 1 – Fenster erstellen – by NBBN (http://nbbn.wordpress.com) – CrashKurs-Artig.

Was brauche ich?

-Einige C++ Kenntnisse

Wie man in C++ mit Hilfe von Qt GUI-Anwendungen programmieren kann. Ich möchte hier irgendwie nicht wirklich auf die Hintergrundgeschichten(durchlesen wäre nicht schlecht) eingehen, dazu gibt es auf anderen Seiten genug. Fenster lassen sich auch wesentlich leichter mit dem QtDesigner erstellen, was aber meines erachtens nicht der beste Weg ist um ins Framework einzusteigen. Außerdem lässt sich mit RAD Tools sowieso nicht alles machen, was per Code geht. Für alle Beispiele empfehle ich auch die Dokumentation offen zu halten, um gegenfalls weitere Infos zu erhalten.

1) Wir erstellen mit der Klasse QWidget ein Fenster[1]. QWidget ist die Basisklasse aller untergeordneten Elemente, wie zum Beispiel Textfelder oder Buttons.

Zuerst einmal ist eine Headerdatei dran. Außerdem sollte man ein paar Basics in Klassen haben, sonst kommt man eventuell schnell zu Verständnisproblemen. Solche einfachen Beispielen werden (unter „Linux“) mit:

```
qmake -project
```

Die .pro Datei wird hiermit erstellt. Diese beinhaltet unter anderem, welche Header und Sourcedateien dann im Makefile auftauchen werden. Außerdem werden durch diese weitere Einstellungen für den Linker vorgenommen, was wir aber in diesem Tutorial nicht brauchen.

```
qmake
```

Dies generiert das Makefile.

```
make
```

anschließend wird der Kompiliervorgang gestartet.

kompiliert.

Unter MS Windows System läuft AFAIK genauso ab. Jedoch sollte man hier das Qt Command Prompt benutzen.

[1] Das hier ist nur ein Fenster, und kein „Hauptfenster“. Man kann es zwar dafür verwenden, aber es eben keins ;). Die Klasse für ein Hauptfenster ist QMainWindow

window.h:

```
#ifndef window_h
#define window_h
Verhindert ein mehrfaches „include“ in einem Programm, sonst würde es zu Problemen kommen.

#include <QApplication>
Dies brauchen wir nicht direkt für das Fenster, sondern für die Datei main.cpp, die wir danach erstellen werden. Könnte man aber auch dort machen, ist logischerweise in dieser Datei nicht
```

zwingend erforderlich.

```
#include <QWidget>
#include <QPushButton>
#include <QMessageBox>
```

Für Leute die nicht jedesmal sich um jedes Widget und andere Klassen kümmern wollen, reicht auch in den meißten Fällen, die was mit GUI zu tun haben:

```
#include <QtGui>
```

Was natürlich seine Nachteile mit sich bringt...

QWidget wird fürs Fenster hier benötigt. Außerdem nimmt diese Klasse Maus- und Tastatur Aktion etc. auf.

QPushButton für unseren schönen Knopf.

QMessageBox, na im Grunde ein Pendat wie in VB(S) MsgBox oder so ähnlich, aber ich glaube mit mehre Einfluss auf den Inhalt ;).

```
class window : public QWidget
{
```

Wir erstellen eine Klasse „window“, diese vererbt Qwidget.

```
Q_OBJECT
```

Dieses Macro wird unter anderem für das Signal-Slot Konzept verwendet.

```
public:
window();
```

Nun kommt alles hier rein, was „public“ sein soll. Jap, dass sind in etwa die benötigten Klassenkenntnisse.

```
private:
QPushButton *button; // Ein stinknormaler Button.
private slots:
void meldungsBox();
};
#endif
```

Hier unter private:

-Der Button wird deklariert.

-Private Slots. Der Slot Bereich wird offensichtlich mit private slots: begonnen.

Wer sich fragt, was Signal und Slots sind, klickt am besten darauf. Jetzt erstellt man die dazugehörige .cpp Datei.

window.cpp

```
#include "window.h"
```

Tja das ist wohl selbsterklärend, zumindest sollte dieser Teil an Grundkenntnissen schon dabei sein.

```
window::window()
```

Hier ist window(); aus der Headerdatei.

```
{
setWindowTitle("Der Titel"); // Setzt das Titel eines Fensters.
resize(500, 400); // Breite, Höhe des Fensters in diesem Fall.

button = new QPushButton(tr("Klick"), this);
```

Mit „this“ wurde das Elternfenster angegeben. Und das ist nunmal das hier, also **this**, würde aber auch mit einem Namen funktionieren. tr() ist kaum von Bedeutung, meistens wenn man ein Programm Übersetzen möchte(in ne andere Sprache), oder für Umlaute bzw Kodierung, da verwendet man allerdings trUtf8("...");. Der Button wird an dieser Stelle auf dem Fenster erscheinen.

```
connect(button, SIGNAL(clicked()), this, SLOT(meldungsBox()));
```

1:button ist hier der Auslöser vom Signal

2: SIGNAL(hier clicked, dass heißt, bei einem Klick mit der Maus wird der Slot aufgerufen.

3: Objekt welches den Slot beinhaltet, in unserem Fall also **this.**
4: meldungsBox() ist unser Slot.

```
}


void window::meldungsBox()
{
 QMessageBox msgBox;
 msgBox.setText("N Guten Tag");
 msgBox.exec();
}
```

Unser Slot. Dieser gibt eine „Meldungs-Box“(was für ein Wort :/) aus.

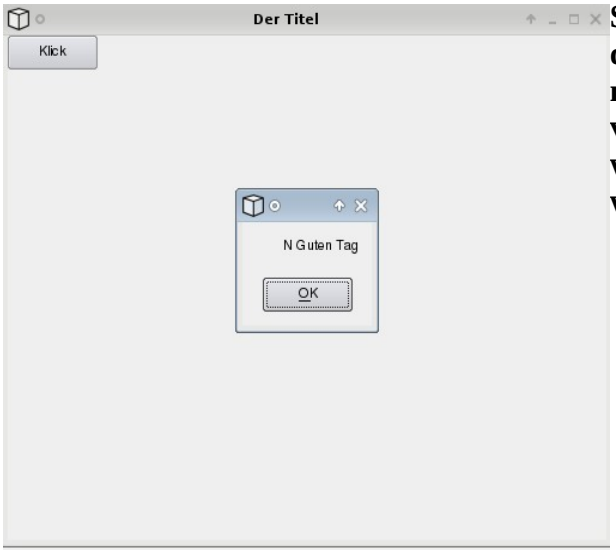
main.cpp

```
#include "window.h"


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

```
window *derpointer = new window;
derpointer->show();
return app.exec();
}
```

Diese Datei führt erst unseren Laden aus.



So in etwa dürfte das Ganze bisher aussehen. Da dies relativ „uncool“ ist, ist logischerweise etwas mehr zu tun. Zum Beispiel könnte man Layouts verwenden. Oder für den Anfang dies ausgeben, was ein User in ein Textfeld eingibt. Betrachten wir uns dazu einmal unteren Code.

window.h

```
#ifndef window_h
#define window_h

#include <QApplication>

#include <QtGui>
class window : public QWidget
{

Q_OBJECT

public:
window();
```

```
private:
QPushButton *button;
QLineEdit   *text;
QGridLayout *layout;
private slots:
void gibAus();
};
#endif
```

window.cpp

```
#include "window.h"
window::window()
{
setWindowTitle("Der Titel"); // Setzt das Titel eines Fensters.
resize(300, 200); // Breite, Höhe des Fensters in diesem Fall.


button = new QPushButton(tr("Klick"), this);
text   = new QLineEdit(this);
layout = new QGridLayout;

layout->addWidget(text, 0,0);
layout->addWidget(button, 0,1);

setLayout(layout);
connect(button, SIGNAL(clicked()), this, SLOT(gibAus()));
}

void window::gibAus()
{
QString inhalt = text->text();
QMessageBox msgBox;
 msgBox.setText(inhalt);
 msgBox.exec();
}
```

Erklärung:

In der HeaderDatei wurde:

```
QLineEdit   *text;
QGridLayout *layout;
```

hinzugefügt. QLineEdit ist logischerweise das Edit-Feld einer Zeile. Wer mehr willl (Multiline), verwendet QTextEdit, hat jedoch auch andere Slots. QGridLayout ist eine interessante Möglichkeit, Widgets einfach anzuordnen und auch ohne RAT-Tools. Dabei wird das ganze in einem entsprechendem Raster angeordnet. Leute die heutztage noch Schach spielen, könnten sich das wie A2, B3 vorstellen, nur halt mit Zahlen. Die Anordnung bleibt auch beim maximieren, verkleinern, vergrößern etc. eines Fensters gegeben.

```
text   = new QLineEdit(this);
```

Auch hier wird wieder einmal das Elternfenster gesetzt.

```
layout = new QGridLayout;
```

Hier nicht, der Grund steht weiter unten...

```
layout->addWidget(text, 0,0);
```

Entspricht A8.
```
layout->addWidget(button, 0,1);
```

Entspricht B8.

```
setLayout(layout);
```

Das ist der Grund. Dies ist ein Layout, wird also mit setLayout gesetzt. Es gibt noch andere, etwa QHBoxLayout oder QVBoxLayout (Horizontal, Vertikal).

Im Slot findet man:

```
QString inhalt = text->text();
QMessageBox msgBox;
 msgBox.setText(inhalt);
 msgBox.exec();
```

Für Erklärungen zu QString sollte man sich die Doku ans Herz nehmen. Mit dem QLineEdit Slot „text()“, welcher einen QString als Rückgabewert hat, wir der Inhalt des Widgets im QString „inhalt“ gespeichert. setText erwartet einen QString, „inhalt“ wird also an den Slot übergeben.

Soviel zum Einstieg in die GUI-Programmierung unter Qt. Das alles hier sind natürlich nur Spitzen von Eisbergen. Wer mehr zu den Klassen will, der schaut sich am besten die wirklich gute Dokumentation an.