

C++ GUI mit Qt

Link Klassen EN: <http://doc.qt.io/qt-5/classes.html>
Link Beispiel Liste EN: <http://doc.qt.io/qt-5/all-examples.html>
Link Beispiele & Tutorials EN: <http://doc.qt.io/qt-5/qtexamplesandtutorials.html>

Buch EN: C++ GUI Programming with Qt 4
EAN 9780132354165 / ISBN 978-0-13-235416-5



INHALTSVERZEICHNIS

1.	Qt einrichten	4
2.	Qt automatisch Kommentar-Header aktivieren.....	5
3.	Qt Beautifier aktivieren (Code Style)	6
4.	GitHub installieren.....	7
5.	GitHub Repository erstellen	8
6.	Linken mit Bibliotheken	9
7.	Begriffe und Grundlagen bei Qt.....	11
8.	Vererbungsbaum vom QT Klassen	11
9.	Übersicht der wichtigsten Widget's.....	12
10.	Erstellen neues Projekt	13
11.	Qmake Projekt Datei.....	15
12.	Widget sktrukturiert.....	16
13.	Widget objektorientiert	17
14.	Vorwärtsdeklaration (Forward declaration).....	18
15.	Signal und Slot.....	20
16.	Widget & Item-Widgets Methoden.....	27
17.	QString	28
18.	CheckBox	29
19.	RadioButton	30
20.	PushButton	31
21.	Slider	32
22.	ProgressBar.....	34
23.	SpinBox	36
24.	Label.....	37
25.	LineEdit.....	38
26.	TextEdit	40
27.	ComboBox.....	42
28.	ScrollArea	43
29.	TableWidget	45
30.	TableWidgetItem.....	47
31.	TabWidget	48
32.	Layout.....	50
32.1.	Kein Layout	50
32.2.	QHBoxLayout.....	51
32.3.	QVBoxLayout.....	52
32.4.	QHBoxLayout & QHBoxLayout kombiniert	53
32.5.	QFormLayout	54
32.6.	QGridLayout	55
32.7.	ContentMargin	56
32.8.	Spacing.....	56
32.9.	Layout Methoden von QBoxLayout	56
33.	Frame	57
34.	MesseageBox	58
35.	Validator	60
35.1.	LineEdit / QString.....	61
35.2.	LineEdit / Integer.....	62
35.3.	Validate Resultat selbst.....	63
36.	Mask.....	65
37.	Completer	66
38.	Icon.....	67
39.	Pixmap / Image (Bilder einfügen)	70
40.	MainWindow	74
41.	Tool-Bar und Tool-Button.....	76
42.	MouseEvent.....	78
43.	PaintEvent und Painter	80

44.	Zeichen mit MouseEvent	82
45.	Dialog	84
46.	ChildWidget (Vererbung)	86
47.	MySQL	87
48.	TcpServer	88
49.	TcpSocket.....	89
50.	Lesen Dateien.....	90
51.	Schreiben Dateien	91
52.	Mutex.....	92
53.	Thread	93
54.	KeyboardHandler	95
55.	KeyEvent	96
56.	Font	97
57.	Translator	98
58.	Conan Package Manager installieren	99
59.	App deployment Windows mit InnoSetup.....	105
60.	App deployment bei Qt	106

1. Qt einrichten

Qt kann mit Studenten Lizenz gratis auf folgender Homepage heruntergeladen werden:

Link: <http://www.qt.io/download/>

Hierzu muss zuerst ein Konto erstellt werden...

Qt Konto Michael Mislin:

Login: xxxxxxxxle@hotmail.com

Password: B.....-xx

... danach kann man das Execute herunter laden ...

Link: <http://www.qt.io/download-open-source/>

... mit welchem man dann die Software installieren kann.

WICHTIG:

Für Windows muss es eine Download Version sein in welcher das MinGW als Compiler bereits enthalten ist. Wenn dies nicht der Fall ist, muss der Compiler von Hand selbst installiert werden. Die im 2016 verfügbare Version 32 Bit läuft auch auf 64 Bit (Windows 10).

Windows Host

- > Qt 5.5.1 for Windows 64-bit (VS 2013, 823 MB) (info)
- > Qt 5.5.1 for Windows 32-bit (VS 2013, 804 MB) (info)
- > Qt 5.5.1 for Windows 32-bit (VS 2012, 747 MB) (info)
- > Qt 5.5.1 for Windows 32-bit (VS 2010, 725 MB) (info)
- > **Qt 5.5.1 for Windows 32-bit (MinGW 4.9.2, 1.0 GB) (info)**
- > Qt 5.5.1 for Android (Windows 32-bit, 1.1 GB) (info)
- > Qt 5.5.1 for Windows RT 32-bit (914 MB) (info)

Execute erstellen:

Mit den folgenden Link kann mehr darüber erfahren wie man GUI's mit Qt erstellt die ausserhalb von Qt lauffähig sind:

Setup Routine mit den DLL erstellen. DLL müssen unter Windows/System32 installiert werden. Dies kann mit einer Setup Routine gemacht werden.

Link: <http://www.computerbase.de/forum/showthread.php?t=1038942>

Execute innerhalb von Windows mit SDK „Visual Studio 10 Express“ oder das „Microsoft SDK 7.1“:

Link Anleitung:

<http://stackoverflow.com/questions/3791808/how-can-i-use-the-windows-sdk-with-qt-creator>

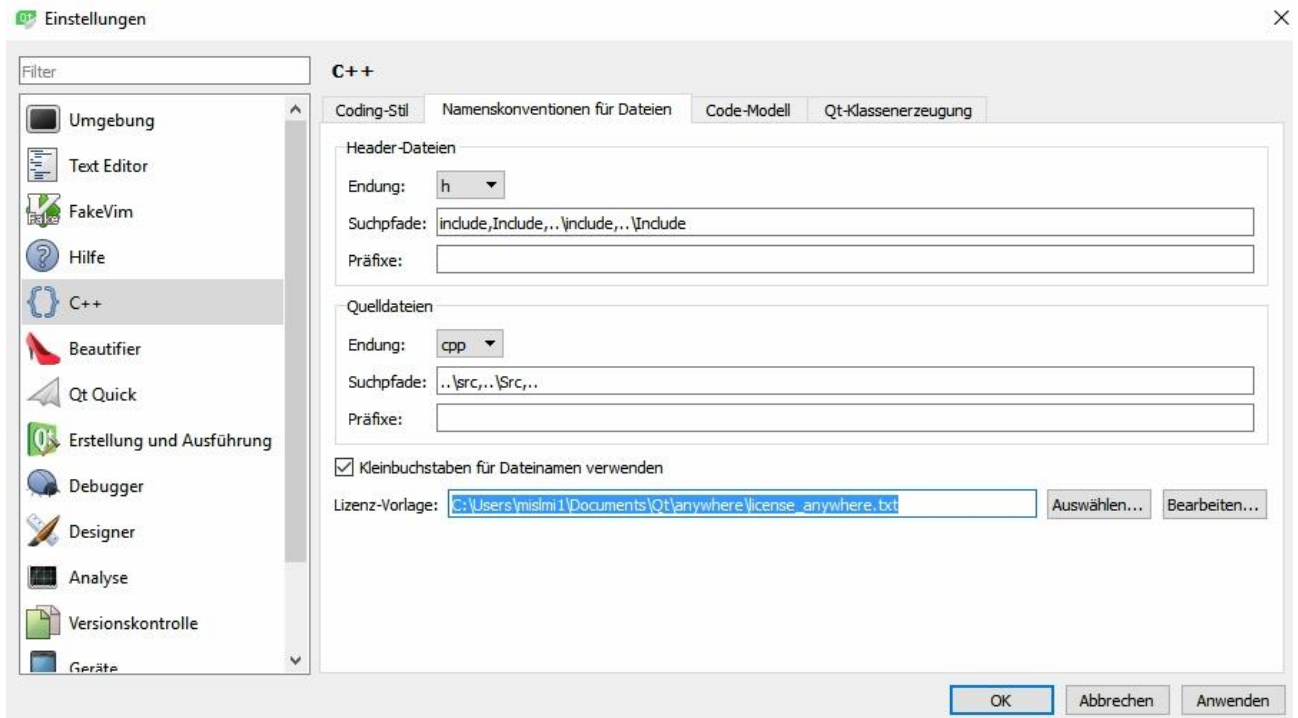
Link Download SDK Windows 10:

<https://dev.windows.com/en-us/downloads/windows-10-sdk>

2. Qt automatisch Kommentar-Header aktivieren

Ein Template für den Kommentar-Header der Source Dateien implementiert man automatisch wenn eine Datei „myHeader.txt“ mit dem Header erstellt wird und im Hauptverzeichnis des Projektes gespeichert wird. Man kann dann die Datei „myHeader.txt“ wie folgt im Qt aktivieren:

- Im Menu "Extras/Einstellungen/{}C++/Namenskonventionen für Dateien" ganz unten bei "Lizenz-Vorlage" die Datei ".\anywhere\ myHeader.txt" auswählen und speichern



WICHTIG! Der Kommentar-Header wird nicht Rückwirken in den Dateien vom Projekt eingefügt, dort muss der Text manuell eingefügt werden.

Beispiel für Open Source Kommentar-Header Qt:

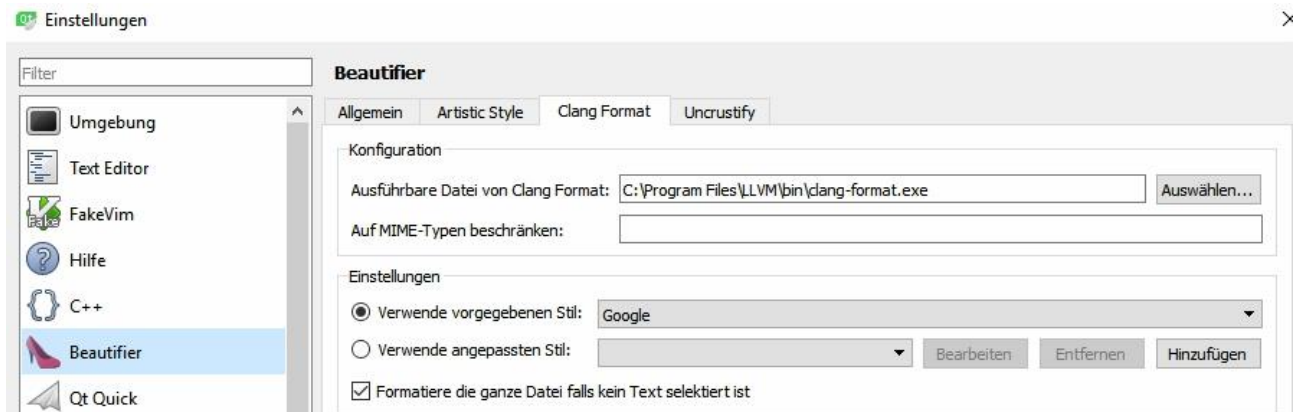
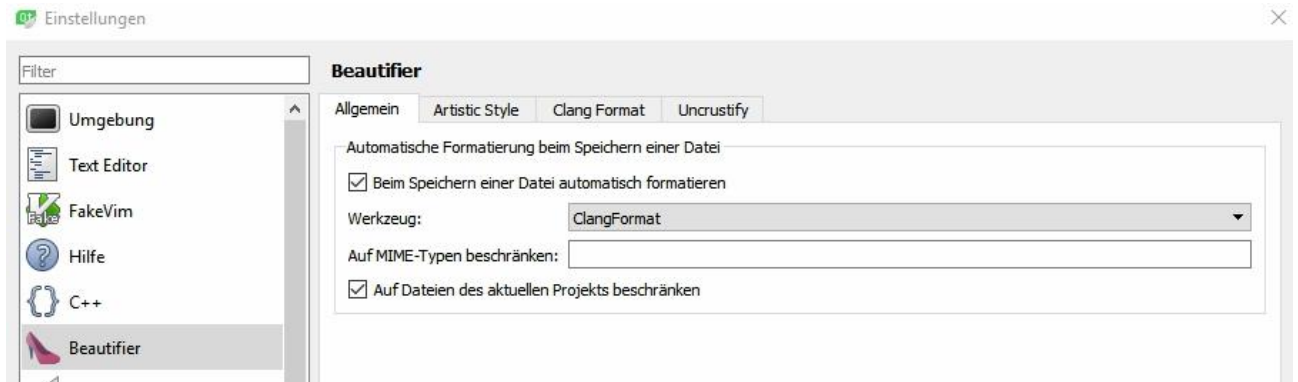
```
/* organisationName mySoftwareName
 *
 * This file is part of the mySoftwareName
 *
 * Copyright (C) 20nn <enterYourEmailAddressHere>
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 *
 * File name:  %FILENAME%
 * Created at: %DATE%
 */
```

Für mehr Umgebungsvariablen siehe auch folgender Link: <http://doc.qt.io/qtcreator/creator-tips.html> im Kapitel <Adding a License Header Template for C++ Code>.

3. Qt Beautifier aktivieren (Code Style)

Folgende Schritte durchführen um den CLang (Google) Codestyle für das Projekt im Qt zu aktivieren:

1. Download pre-build-binaries unter <http://releases.llvm.org/download.html>
2. Installieren auf dem Rechner
3. Aktivieren in Qt unter "Extras/Einstellungen/Beautifier" im Tab "Allgemein" und "Clang Format" die folgenden Einstellungen machen:



4. GitHub installieren

GitHub ermöglicht die Verwaltung eines Projektes automatisiert für einen oder mehrere Programmierer.

1. Download GitHub unter folgendem Link: <https://git-scm.com/downloads>
2. Installieren
3. Öffnen CLI (Commando Line Interpreter)
4. Eingeben „git --help“ und Enter drücken

Wenn Installation erfolgreich war und git als Umgebungsvariable installiert ist, sollten wie folgt dargestellt alle möglichen Befehle von GitHub aufgelistet werden:

```
C:\>git --help
usage: git [--version] [--help] [-C <path>] [-c name=value]
       [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
       [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
       [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
       <command> [<args>]

These are common Git commands used in various situations:


start a working area (see also: git help tutorial)
clone      Clone a repository into a new directory
init       Create an empty Git repository or reinitialize an existing one


work on the current change (see also: git help everyday)
add        Add file contents to the index
mv         Move or rename a file, a directory, or a symlink
reset      Reset current HEAD to the specified state
rm         Remove files from the working tree and from the index


examine the history and state (see also: git help revisions)
bisect     Use binary search to find the commit that introduced a bug
grep       Print lines matching a pattern
log        Show commit logs
show       Show various types of objects
status     Show the working tree status


grow, mark and tweak your common history
branch     List, create, or delete branches
checkout   Switch branches or restore working tree files
commit     Record changes to the repository
diff       Show changes between commits, commit and working tree, etc
merge      Join two or more development histories together
rebase     Reapply commits on top of another base tip
tag        Create, list, delete or verify a tag object signed with GPG

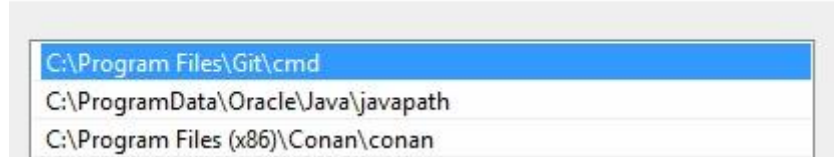

collaborate (see also: git help workflows)
fetch      Download objects and refs from another repository
pull       Fetch from and integrate with another repository or a local branch
push       Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.

C:\>
```

Wenn nicht muss der Befehl git unter <Systemumgebungsvariablen> von Hand wie folgt eingefügt werden:

Umgebungsvariable bearbeiten



```
C:\Program Files\Git\cmd
C:\ProgramData\Oracle\Java\javapath
C:\Program Files (x86)\Conan\conan
```

5. GitHub Repository erstellen

Im GitHub Repository kann ein Projekt geführt werden. Um ein Projekt zu eröffnen muss unter folgendem Link ein Account eröffnet werden.

Link: <https://github.com/join?source=header-home>

Der Account berechtigt public, als öffentliche Repositories zu erstellen. Soll das Repository „privat“ sein soll, muss bei GitHub bezahlt werden.

Eine alternative bietet die gratis Software „GitLab“, was eigentlich das Selbe wie „GitHub“ ist jedoch auf einem privaten Server installiert werden kann. Somit können auch die Repository auch nach eigenem Ermessen privat oder öffentlich gemacht werden.

Beim erstellen sollten die folgenden Dateien immer als Grundlage erstellt werden:

README.* In der Readme Datei soll das Software Projekt beschrieben werden. Hierzu sollten die folgenden Punkte falls zutreffend abgearbeitet werden:

Titel	Repository Titel
Aktuelle Version	Neuste Version und deren Beschreibung wie zum Beispiel die Fehlerbehebungen die in dieser Version gemacht wurde.
Versionsverlauf	Kompletter Versionsverlauf (*Optional)
Beschreibung	Beschreibung der Software oder Projektes.
Bedingungen	Mit welcher IDE wurde das Projekt erstellt, was werden für externe Bibliotheken verwendet, was für zusätzliche Tools wie zum Beispiel „conan“ müssen installiert werden und wie müssen diese angewendet werden.
Beispiele	Programmierbeispiel wie die Software oder das Programm verwendet werden kann.

LICENSE.* Unter welcher Lizenz wurde das Projekt erstellt.

.gitignore Diese Datei bietet die Möglichkeit Dateien die lokal generiert werden (wieso auch immer) aber nicht im Projekt in GitHub sein sollen bei einem Commit nicht in das Repository aufgenommen werden (ignoriert werden). Das Routenzeichen # kann Text auskommentiert werden.

Beispiel Aufbau:

```
# C++ objects and build libs
/include      #Ordner /lib und alle Dateien darin werden ignoriert
*.dll        #Dateien mit der Endung „.dll“ werden ignoriert

# Qt-Project
*.pro.user   #Die Qt Projekt Datei (ist persönlich und auf den lokalen Computer
              #bezogen) mit der Endung „.pro.user“ wird ignoriert
```


6. Linken mit Bibliotheken

Es gibt die Möglichkeit externe Bibliotheken (library) für bestimmte Funktionen wie zum Beispiel SSH Client mit dem eigenen Code zu verwenden. Dazu wird der Quellcode der externen Bibliothek mit Compiler gebaut (build). Das Bauen kann zu einer statischen oder einer dynamischen Bibliothek führen die je nach Betriebssystem unterschiedliche Namenskonventionen haben.

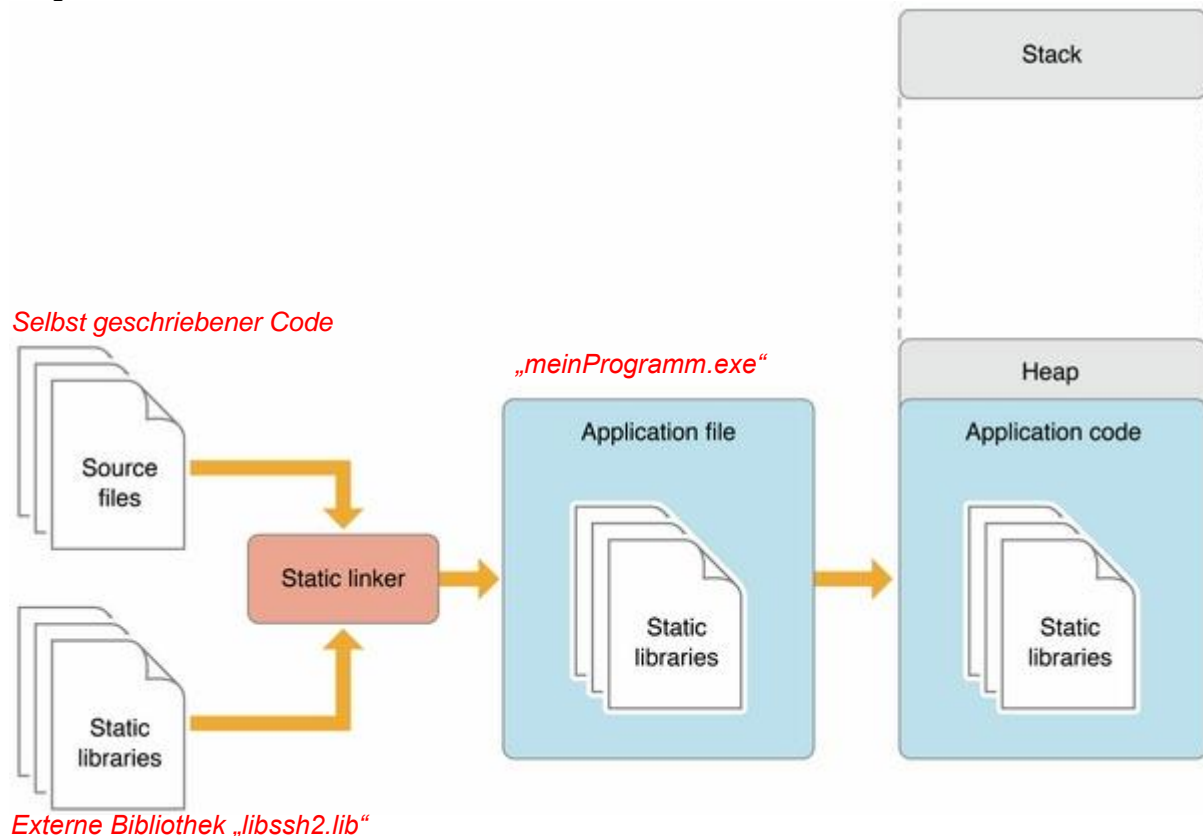
OS	Dynamische Bibliotheken	Statische Bibliotheken	Ordner nach Konvention	Weiterführende Links & Hilfe:
Windows	.dll	.lib	.\bin	
Linux	.so (.so.1)	.a	./lib	
Mac	.dylib or .so	.a	./lib	
iOS	.dylib or .so	.a (.la)	./lib	dynamic static
Android	.?	.?		

Sofern eine externe Bibliothek darauf ausgelegt ist, in Programmen als externe Bibliothek eingefügt verwendet zu werden, ist eine Programmierschnittstelle oder auch API (Application Programming Interface) genannt vorhanden.

Mit dem selbst erstellten Programm kann dann mittels der Methoden auf diese API's zugegriffen werden und die Bibliothek verwendet werden.

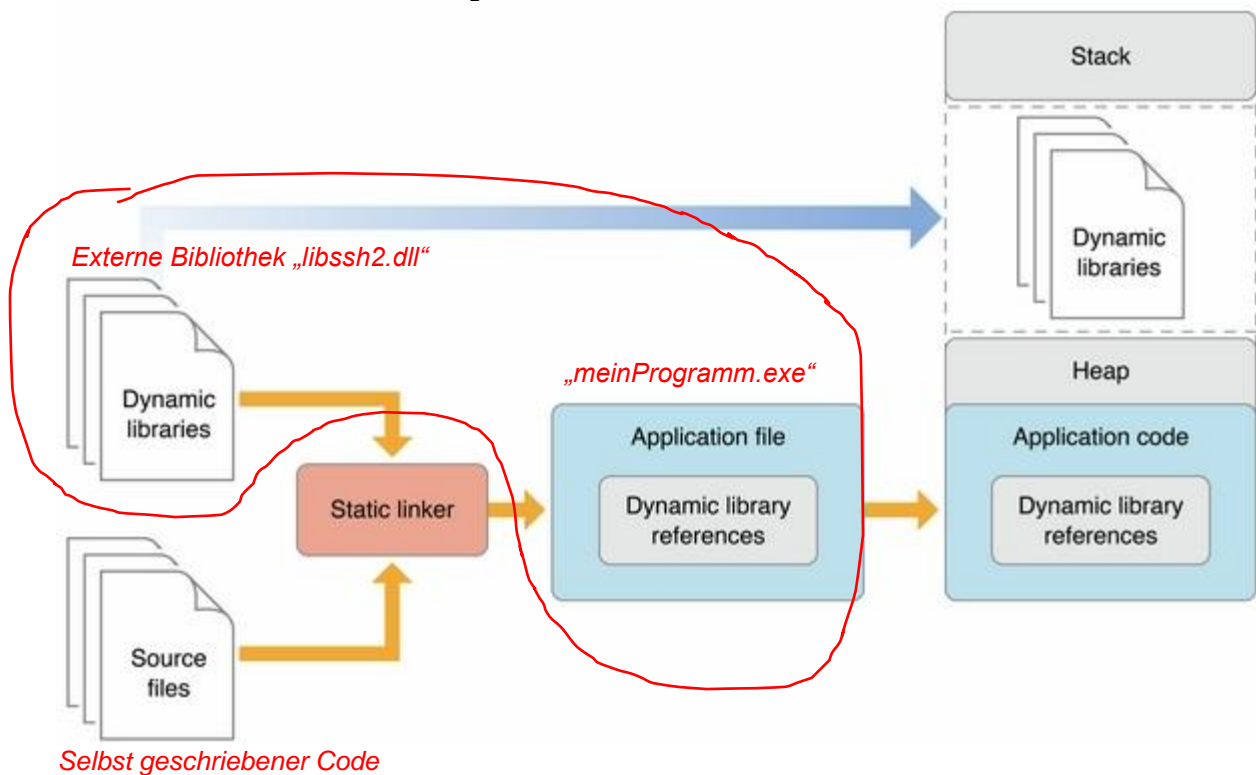
Funktionsweise statisch Linken

Beim statischen Linken wird die Bibliothek beim Builden direkt in die ausführbare Datei integriert. Zum Beispiel unter Windows besteht das Programm am Ende aus der Datei „meinProgramm.exe“, das Programm besteht also aus einer Datei.



Funktionsweise dynamisch Linken

Beim dynamischen Linken wird die Bibliothek eigenständig behalten und die API genutzt um die Methoden der externen Bibliothek zu verwenden. So besteht ein solches Programm nach dem Build unter Windows aus der Datei „meinProgramm.exe“ und „libssh2.dll“



WICHTIG: Beim dynamischen Linken muss normalerweise das OS selbst wissen wo die Bibliotheken sind. Der Dateityp und der Ort ist abhängig vom Betriebssystem und deshalb wird folgend für die wichtigsten OS Windows, Linux und Mac kurz aufgezeigt:

OS	Dateityp	Pfad
Windows	.dll	mit der ausführbaren Datei (.exe) zusammen
Linux	.so (.so.1)	System Bibliotheken Pfad, meisten: /usr/lib siehe auch unter folgenden Link: http://doc.qt.io/qt-5/linux-deployment.html
Mac	.dylib	mit der ausführbaren Datei zusammen

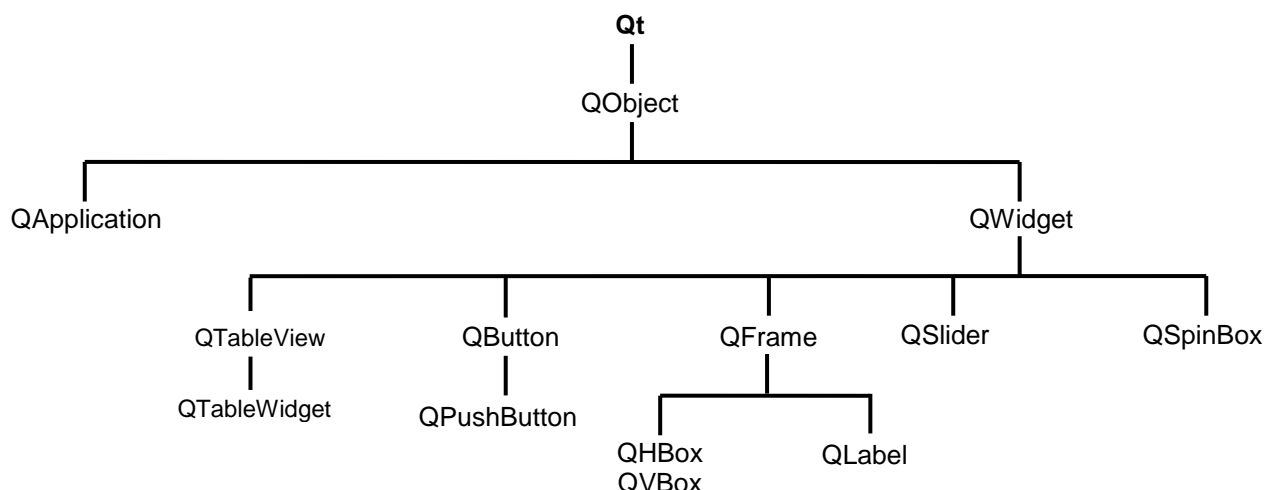
7. Begriffe und Grundlagen bei Qt

Siehe auch folgenden Link: <https://de.wikipedia.org/wiki/Steuerelement>
[https://en.wikipedia.org/wiki/Widget_\(GUI\)](https://en.wikipedia.org/wiki/Widget_(GUI))
https://en.wikipedia.org/wiki/List_of_graphical_user_interface_elements

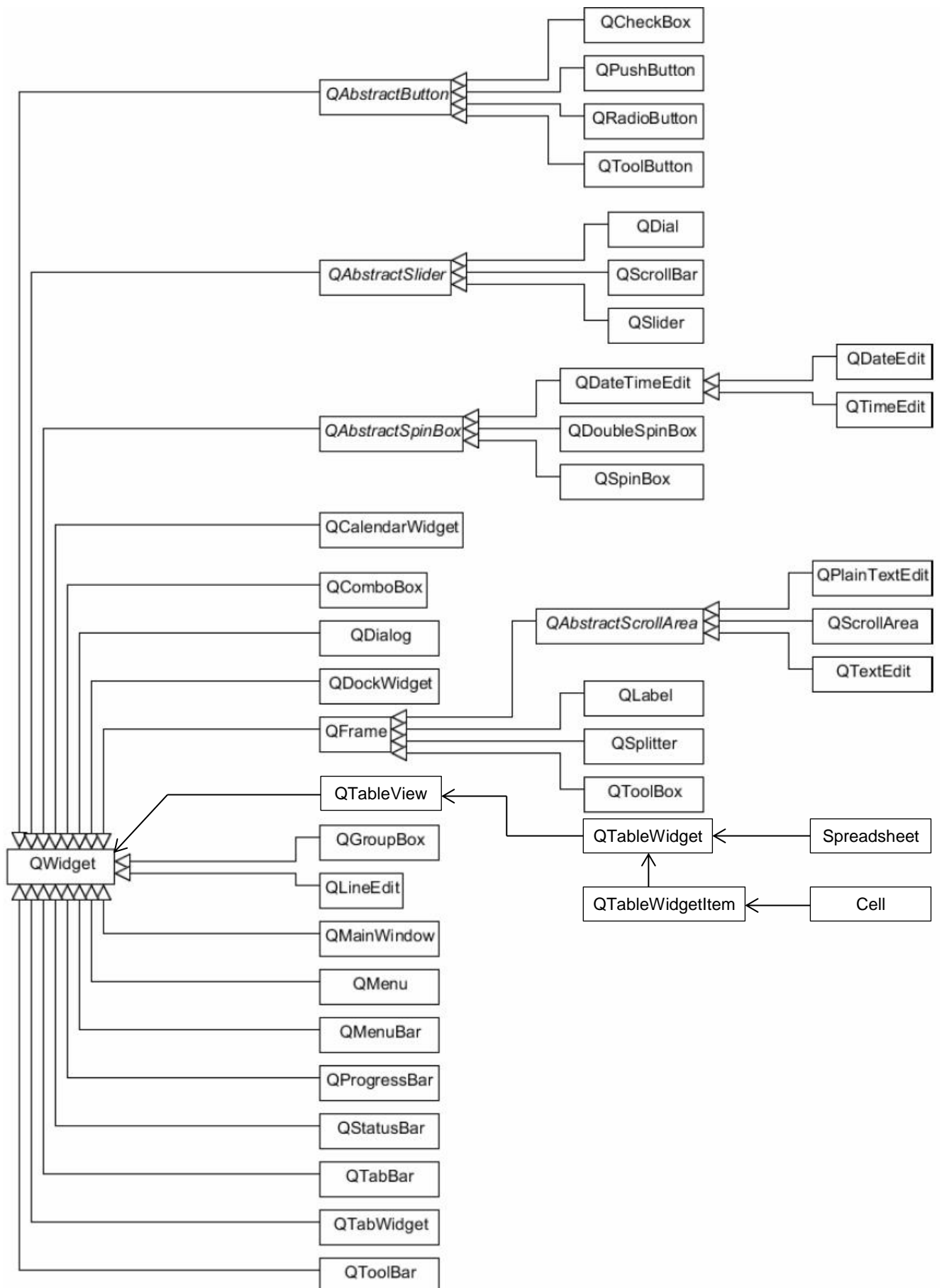
Window	Ein Fenster welches mit den typischen Menüs ergänzt werden kann
Widget	Kommt aus der Unix Terminologie und ist ein Visuelles Element in ein GUI. Ist vergleichbar mit „Control“ und „Container“ aus dem Windows Bereich. Widget setzt sich aus dem Wort Windows und Gadget zusammen.
Signal	Buttons, Menus, Scroll bars und Rahmen sind ebenfalls Widgets, diese bezeichnet man als sogenannte Item-Widgets. Ein Träger-Widget kann auch andere Widgets aufnehmen. Siehe hierzu auch Klassenableitung der Widgets in diesem Dokument Wir von Qt zur Verfügung gestellt um Aktionen innerhalb des GUI abzufangen. Folgende Beispiele gibt es dafür: <ul style="list-style-type: none">• anklicken eines Button• Drücken mit Maus innerhalb eines Widget• Loslassen der Maus innerhalb eines Widget• Wert hat in einem Eingabefeld geändert• - usw.
Slot	Ist eine auszuführende Funktion um eine Signal aufzufangen und den enthaltenen Code auszuführen. Qt stellt standardisierte Funktionen Slot's zur Verfügung oder aber der Programmierer erstellt sich seine eigenen Slot's mit dem entsprechenden Code.
Connector	Ist ein Objekt, welches Qt zur Verfügung stellt um Signale und Slots miteinander zu verbinden.
Socket	Ist eine Schnittstelle. Bei den Standard Schnittstellen wie einem seriellen COM Port kann immer nur ein GUI darauf zugreifen und danach ist die Schnittstelle für dieses GUI reserviert. Bei TCP/IP können jedoch zeitgleich verschiedene GUI's auf verschiedene Port auf den gleichen Server oder Client zugreifen was dem Programmierer die Möglichkeit gibt pro Server oder Client verschiedene Sockets (Verbindungen / Schnittstellen) zu definieren und zu verbinden.

8. Vererbungsbaum vom QT Klassen

Folgend ist die Verbindung und Ableitung der wichtigsten Objekte in Qt ersichtlich:

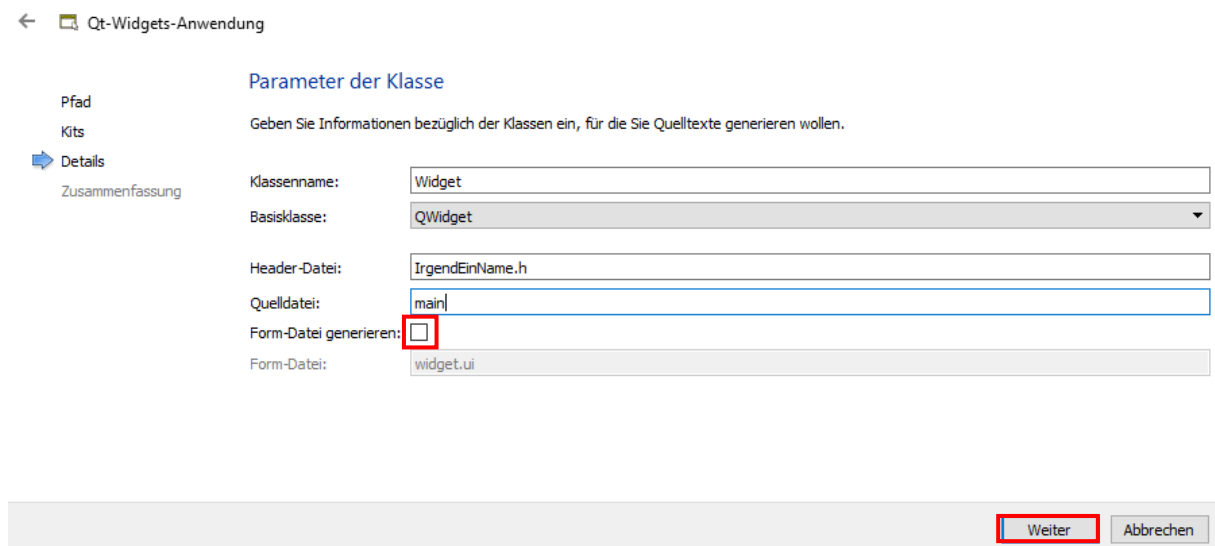
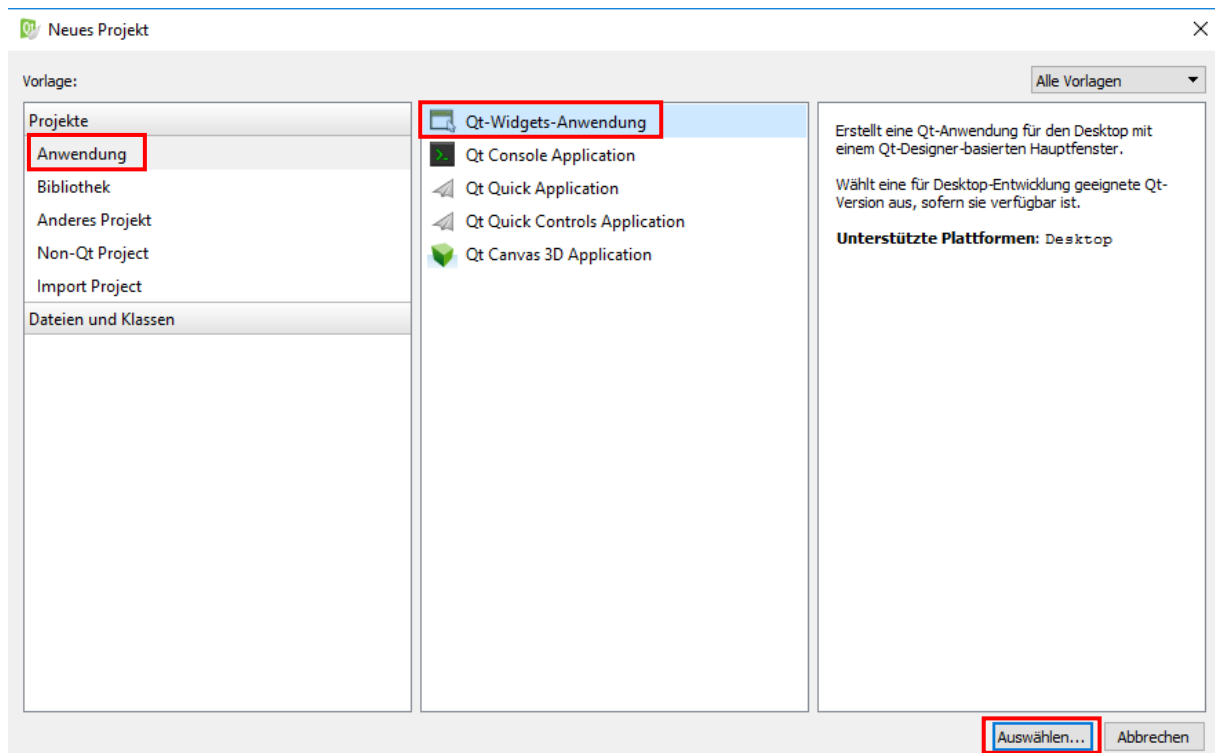


9. Übersicht der wichtigsten Widget's

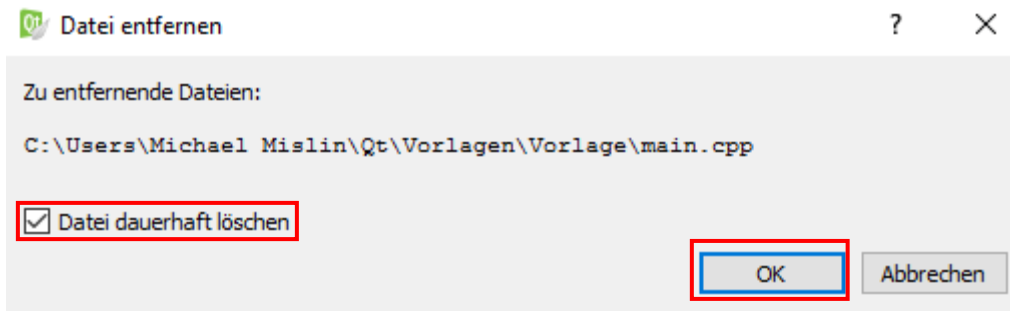


10. Erstellen neues Projekt

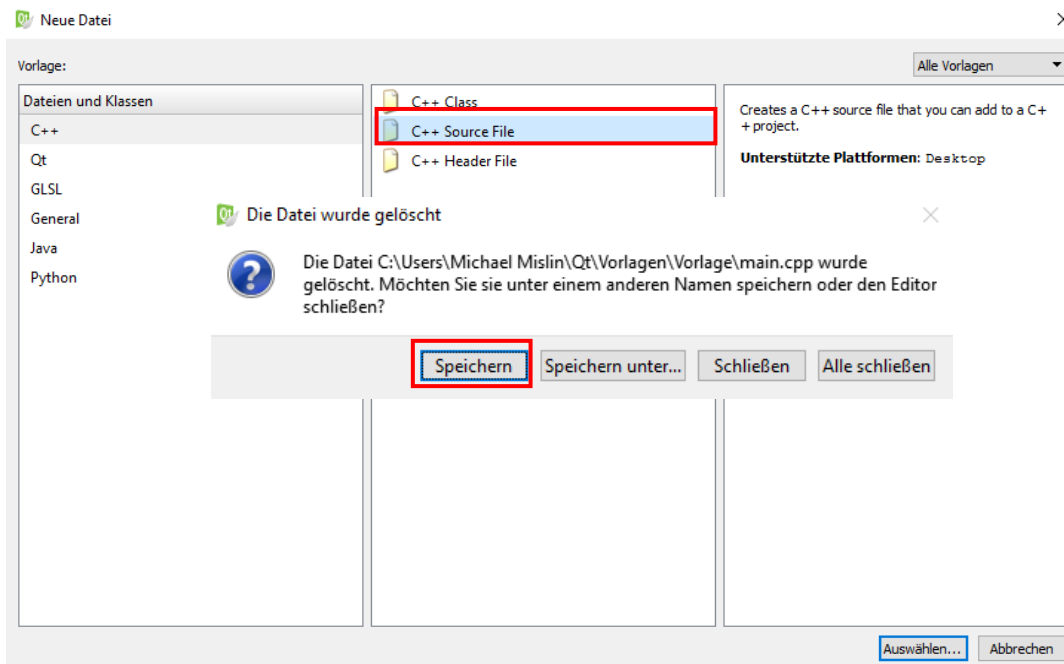
Wenn das Projekt als „QT-Widgets-Anwendungen“ wie folgt ausgewählt wird (siehe das folgende Bild) , erstellt der Qt Creator ein Grund Widget.



Nachdem das Projekt erstellt ist müssen die beiden beiden Source Dateien `xxx.h` und `xxx.cpp` entfernt werden indem man der linken Maustaste die Datei auswählt, „Dateien entfernen...“ auswählt und dann die Datei mittels „Datei dauerhaft löschen“ entfernt.



Danach muss wieder ein neues Source File „main.cpp“ eingefügt werden indem man mit der linken Maustaste den obersten Projektordner auswählt und dann „Hinzufügen...“ auswählt. Hier ein C++ Source File auswählen, den Namen main.cpp vergeben und „Speichern“



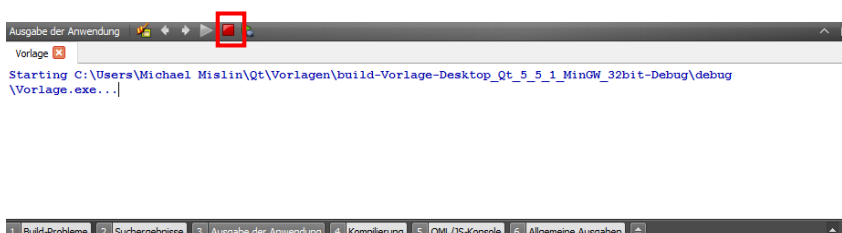
Nachdem das leere „main.cpp“ jetzt erstellt wurde kann das minimale Grundgerüst eines jeden Programmes wie folgt erstellt werden:

```
#include <QCoreApplication>

int main(int argc, char *argv[])
{
    QCoreApplication TEST1(argc, argv);           // Test1 ist der Name der Applikation

    return TEST1.exec();
}
```

Wichtig: Wenn das Projekt in dieser Ausführung kompiliert wird, muss es nach dem Starten unten wieder mit dem roten Stopp Knopf gestoppt werden!
Die Ansicht kann geöffnet werden indem man auf den Knopf „3, Ausgabe der Anwendung“ auswählt.



Erklärung zum Code vom Grundgerüst:

Code	Erklärung
<code>int main(int argc, char *argv[])</code>	Commando line Argumente werden von ausserhalb an C ++ an das Main übergeben
<code>QApplication nameAPP(argc, argv);</code>	Erstellt ein QAppliacion Objekt um alle Applikationsweite Ressourcen zu handhaben. Der QApplication Konstruktor braucht <i>argc</i> und <i>argv</i> weil Qt selbst Commando line Argumente verwendet kann.

11. Qmake Projekt Datei

Jedes Projekt das in Qt erstellt wird hat automatisch eine Projet-Datei mit der Syntax ***anyName.pro***. Über diese datei wird dem ***qmake*** mitgeteilt was alles in dem Projekt für Dateien enthalten sind, was für zusätzliche Qt Librarys (Qt += xxx) eingebunden sind, welche Ressourcen mittels Ressourcen Dateien eingebunden sind, usw.

Wenn zum Beispiel normale C++ V11 Syntax verwendet werden soll (auch int, double, ...usw. Variablen) dann muss zum Beispiel die Zeile ***CONFIG += c++11*** in die qmake Projekt Datei eingefügt werden.

Link im Netz: <http://doc.qt.io/qt-5/qmake-project-files.html>

Beispiel:

Project created by QtCreator 2016-05-06T15:36:57

QT += sql
QT += network
QT += core gui
QT += widgets serialport

CONFIG += console
CONFIG += c++11

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

TARGET = object_Serial_handling
TEMPLATE = app

RESOURCES += resources/images.qrc *# Path starts where project source files are located*

SOURCES += \
file.cpp \
fileadapt.cpp \
filecopy.cpp \
filegenerate.cpp \
fileread.cpp \
main.cpp \

HEADERS += \
file.h \
fileadapt.h \
filecopy.h \
filegenerate.h \
fileread.h \

12. Widget sktrukturiert

Eine Widget (Fenster) Instanz welche als Container für andere Control und Display Widgets arbeitet kann wie folgt im Main erstellt werden.

Code :

```
#include <QApplication>
#include <QWidget>

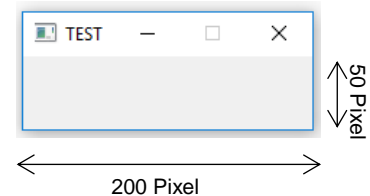
int main(int argc, char *argv[])
{
    QApplication TestApp(argc, argv);

    QWidget *testWindow = new QWidget;
    testWindow->setWindowTitle("TEST");
    testWindow->setFixedSize(200, 50);

    testWindow->show();

    return TestApp.exec();
}
```

Ausgabe:



Code Analyse

Code	Erklärung
<code>#include <QWidget></code>	Bibliothek in Projekt aufnehmen Link im Netz: http://doc.qt.io/qt-5/qwidget.html
<code>QWidget *testWindow = new QWidget;</code> oder <code>QWidget *testWindow = new QWidget(QWidget *parent = 0);</code>	Ein Fenster (Widget) mit dem Namen testWindow erstellen Beide Arten haben die selbe Funktion. Mit dem Sysntax (QWidget *parent = 0) wird zusätzlich das Parent (Übergeordnete) Widget definiert. Wenn = 0 ist dies ein Nullpointer und heisst es gibt kein Parent Widget.
<code>testWindow->setWindowTitle("TEST");</code>	Titel vom Fenster vergeben
<code>testWindow->setFixedSize(200, 50);</code>	Fenstergrösse mit x (waagrecht) und y (senkrecht) definieren. Grössenangabe senkrecht ist immer ohne die Titelleiste
<code>testWindow-> setGeometry(150,150,800,400) [xPos,yPos,XSize,YSize]</code>	Fensterposition 6 -grösse mit x (waagrecht) und y (senkrecht) definieren. Grössenangabe senkrecht ist immer ohne die Titelleiste
<code>testWindow->show();</code>	Anzeigen vom Fenster

13. Widget objektorientiert

Eine Widget (Fenster) Instanz bestehend aus Header und Source welche als Container für andere Control und Display Widgets arbeitet kann wie folgt im Main so oft wie gewünscht gleich erstellt werden.

Code :

main.cpp

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

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

    testWidget *newWidget= new testWidget ();
    newWidget ->show();

    return testAPP.exec();
}
```

Header (testWidget.h)

```
#include <QWidget>

class testWidget: public QWidget
{
    Q_OBJECT

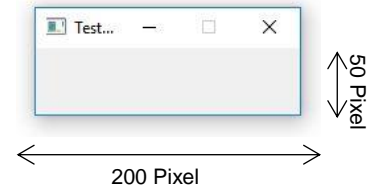
public:
    testWidget ();
};
```

Implementierung (testWidget.cpp)

```
#include <testWidget.h>

testWidget:: testWidget ()
{
    setWindowTitle("Test Widget");
    setFixedSize(200, 50);
}
```

Ausgabe:



14. Vorwärtsdeklaration (Forward declaration)

Wenn man eine eigene Header-Datei schreibst und dort die Klasse mit dem Syntax **`class testKlasse;`** in der Headerdatei definiert ist, muss man diese in anderen den Implementierung Dateien mit dem Syntax **`#include testKlasse;`** einbinden.

Es reicht aus für Header Dateien dem Compiler mit dem Schlüsselwort „class“ bekanntzugeben, dass es **`testKlasse`** gibt. Die Header Datei braucht nicht zu wissen, was die Library alles genau beinhaltet noch wie gross die Objekte sein müssen, wenn man alles Pointers von **`testKlasse`** verwendet. Die ist so, weil alle Pointer immer gleich gross sind und immer plattformabhängige den gleichen Speicherplatz auf der Festplatte einnehmen. Die Methoden wissen die werden dann erst in der *.cpp-Datei benötigt.

Der Punkt ist:

Wenn nun eine Header-Datei wiederum von einer anderen (dritten) Klasse verwendet wird, müsste der Compiler beim Bauen der dritten Objektdatei zunächst die Headerdatei einlesen und durch das `#include` auch die Datei **`testKlasse.h`**. Bei einer Vorwärtsdeklaration bleibt ihm das erspart. Noch wichtiger: Die dritte Klasse bleibt unabhängig von **`testKlasse`**, da sie diese direkt gar nicht nutzt.

Je größer und komplexer das Projekt wird, desto mehr verhindert man dadurch unnötige Abhängigkeiten und reduziert nebenbei die Compilerzeit weil der Compiler mit `#include` jedes Mal erneut die Klasse komplett inkludiert auch wenn dies bereits schon bei anderen Klassen x-mal gemacht wurde.

Beim vorwärts deklarieren geht es also um:

- Anzahl zu inkludierten Headern kürzer
- Probleme mit zyklischen Abhängigkeiten vermeiden

Welche Item-Widgets sollen vorwärts deklariert werden?

Hier gilt es so vorzugehen, dass alles was ausserhalb des Konstruktor „erreichbar“ sein soll vorwärts deklariert sein soll. Zum Biespiel macht es Sinn einen Button oder eine Eingabelinie in der ganzen Klasse verwenden zu können, wobei in den meisten Fällen ein Layout für die Anordnung nur beim erstellen des Widget gebraucht wird. Somit kann mann das Layout direkt im Konstruktor erstellen weil man danach keinen Zugriff mehr darauf braucht.

Folgend ein Beispiel wo die Vorwärtsdeklaration vom Item-Widget grün markiert ist und die deklaration des Layoutes im Kontruktor rot

Code :

main.cpp

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

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

    testWidget *newWidget= new testWidget ();
    newWidget ->show();

    return testAPP.exec();
}
```

header [testWidget.h]

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
class QPushButton;

class testWidget : public QWidget
{
    Q_OBJECT

public:
    testWidget(QWidget *parent = 0);
    void anyMethod() ;

private:
    QPushButton *testButton;
};

#endif // WIDGET_H
```

Implementierung [testWidget.cpp]

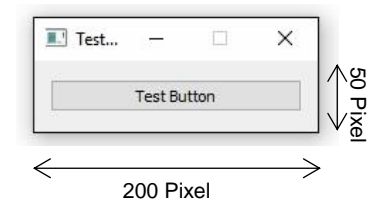
```
#include "widget.h"
#include <QPushButton>
#include < QHBoxLayout >

testWidget::testWidget(QWidget *parent)
: QWidget(parent)
{
    setWindowTitle("Test Widget");
    setFixedSize(200, 50);

    testButton = new QPushButton("Test Button");
    QHBoxLayout *testLayout = new QHBoxLayout();
    testLayout -> addWidget(testButton);
    this -> setLayout(testLayout);           // this ist überflüssig
    this -> show();                         // this ist überflüssig
}

void testWidget::anyMethod()
{
    testButton -> setText("my Text"); // Zugriff ist erlaubt!
    // testLayout -> kein Zugriff nach beenden Konstruktor!
}
```

Ausgabe:



15. Signal und Slot

Verschiedene Widgets von Qt senden „*Signale*“ zum Anzeigen, dass eine Bediener Aktion oder eine Änderung eines Status geschehen ist.

Als Beispiel sendet QPushButton ein *clicked()* Signal wenn der Bediener den Button betätigt.

Ein Signal kann mit einer Funktion verbunden sein (in dem Fall „*Slot*“ genannt), so dass wenn das Signal gesendet wird, die Funktion (der Slot) automatisch ausgeführt wird. Das heisst, sobald ein Signal ausserhalb eines Source Files zu einem anderen Source File übertragen werden muss, muss diese Funktion (der Slot) deklariert werden.

Signale sind „Botschaften“, die bei Eintreten eines Ereignisses abgegeben werden (emittiert). Ein Slot ist prinzipiell eine normale Funktion, die auf eine bestimmte Weise mit einem Signal verknüpft werden kann.

Slots und Signale „wissen“ zunächst nichts voneinander. Erst durch die Verknüpfung entsteht die eigentliche Programmlogik: Jedes Mal, wenn das Signal abgegeben wird, wird anschließend der verbundene Slot aufgerufen.

Wichtig:

- Ein Slot oder Signal wird nie im Header File implementiert
- In der Klasse muss immer als erstes **Q_OBJECT** implementiert sein
- Die Klasse muss **class myClass : public Widget** gesetzt werden
- Die Klasse muss immer **#include <QWidget>** beinhalten und kann NICHT mit *class QWidget;* vorwärts deklariert werden
- Signal und Slot müssen immer denselben Datentyp haben

Vorgehen:

- einfügen vom Q_OBJECT Makro
- nur wenn selbst erstellt einfügen signals: und erstellen der Signal Deklaration. Signale müssen nicht implementiert werden, sie verwenden automatisch das Schlüsselwort „emit“ beim Aufruf
- nur wenn selbst erstellt einfügen public-, protected- oder privat slots: und erstellen der Slot Prototypen
- Implementieren der Slots wie eine normale Methode
- Erstellen einer Verbindung

Syntax:

```
QObject :: connect( senderObject , SIGNAL(signal) ,receiverObject , SLOT(slot) ) ;
```

QObject :: connect

Fixe Syntax, Methode **connect** die von **QObject** abgeleitet ist

senderObject

Das Objekt welches das Signal sendet, zum Beispiel das Objekt *testButton* welches mit der Klasse QPushButton erstellt wird.

Objekt erstellen: QPushButton **testButton* = new QPushButton ("Zurück");

Es kann auch eine eigen Klasse aus einer Header Datei sein und in der Implementierung.cpp kann zum Beispiel über ein Button der Slot Button ausgeführt werden und im Slot Button ist der „emit“ aufruf des Signales. In diesem Fall wäre das Objekt *this*

SIGNAL

Fixe Syntax

signal

Das eigentliche Ereignis welches entweder selbst erstellt wird oder vom Widget zur Verfügung gestellt wird (siehe <http://doc.qt.io/qt-5/classes.html>)

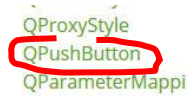
Deklaration ist im Header File unter **signals:** oder in der Referenz vom Widget Objekt (siehe <http://doc.qt.io/qt-5/classes.html>) im Kapitel „Public Slots“ zu finden.

Beispiel eigenes Signal in Header Datei:

```
signals:  
    emit myOwnSignal(variable1, variable2, ...);
```

Beispiel QPushButton aus der Klassenübersicht von Qt:

1. Suchen des QPushButton in der Klassenübersicht und auswählen



QProxyStyle
QPushButton
QMapParameterMapper

Link: <http://doc.qt.io/qt-5/qpushbutton.html>

2. Suchen der Slot in der Beschreibung und auswählen

> 14 public functions inherited from QPaintDevice

Public Slots

void showMenu()

5 public slots inherited from QAbstractButton

> 19 public slots inherited from QWidget

> 1 public slot inherited from QObject

Protected Functions

void initStyleOption(QStyleOptionButton *option) const

Link: <http://doc.qt.io/qt-5/qabstractbutton.html#public-slots>

3. Auswählen von welchem Objekt der Slot abgeleitet sein soll, in unserem Fall direkt von QPushButton, dann werden direkt darunter auch die entsprechenden zur Verfügung stehenden Signale aufgelistet

Public Slots

void	<code>animateClick(int msec = 100)</code>
void	<code>click()</code>
void	<code>setChecked(bool)</code>
void	<code>setIconSize(const QSize &size)</code>
void	<code>toggle()</code>

- › 19 public slots inherited from `QWidget`
- › 1 public slot inherited from `QObject`

Signals

void	<code>clicked(bool checked = false)</code>
void	<code>pressed()</code>
void	<code>released()</code>
void	<code>toggled(bool checked)</code>

- › 3 signals inherited from `QWidget`
- › 2 signals inherited from `QObject`

Protected Functions

Link: <http://doc.qt.io/qt-5/qabstractbutton.html#public-slots>

receiverObject

Das Objekt in welchem die Slot-Funktion ausgeführt werden soll. Dies kann zum Beispiel nur *this* sein oder ein Objekt *testLabel* sein welches mit der Klasse `QLabel` erstellt wird.

Objekt erstellen: `QLabel *testLabel = new QLabel("Hello Qt");`

Es kann auch eine eigene Klasse aus einer Header Datei sein und in der Implementierung.cpp dann die eigene Slot Funktion aufgerufen.. In diesem Fall wäre das Objekt *this*

SLOT

Fixe Syntax

slot

Die Slot Funktion welche entweder selbst erstellt wird oder vom Widget zur Verfügung gestellt wird (siehe <http://doc.qt.io/qt-5/classes.html>)

Deklaration ist im Header File unter **public- / protected- / private slots:** oder in der Referenz vom Widget Objekt (siehe <http://doc.qt.io/qt-5/classes.html>) im Kapitel „Public Slots“ zu finden.

In unsere Fall erstellen wir eine eigene Slot-Funktion in der Header Datei:

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

Implementiert wird dann die Slot-Funktion (Code) in der Implementierungsdatei .cpp wo da Objekt erstellt wird und auf welchem die Slot-Funktion einwirken soll.

Beispiel:

```
QObject :: connect( testButton, SIGNAL(clicked()) ,this , SLOT(onButtonClicked() ) ) ;
```

SIGNAL's:

Verschieden Qt Widgets wozu zum Beispiel der QPushButton, QSlider, QSpinBox, usw. gehören, senden (emit) Signale (Ereignisse, Geschehnisse) über einen Slot (Eingangsöffnung/Schlitz) an ein Empfänger Objekt wo es dann weiter Verarbeitet werden kann.

Die Qt Klassen der Widgets stellen verschiedene Signale zur Verfügung aber es besteht auch die Möglichkeit selber ein Signal zu erstellen.

Klasse	Signal	Erklärung
QPushButton	clicked()	Das BOOL Signal 1 wird übermittelt, wenn der Taster gedrückt wird und die Maus innerhalb des Schalters ist.
	pressed()	Das Signal wird übermittelt wenn der Taster gedrückt wird
	released()	Das Signal wird übermittelt wenn der Taster losgelassen wird
	toggled()	Das BOOL Signal 1 wird übermittelt, wenn der Schalter aktiviert wird und das BOOL Signal 0 wird übermittelt, wenn der Schalter wieder deaktiviert wird.

SLOT's:

Der Empfänger (Funktion) des Signals (Ereignis, Geschehnis) braucht einen Slot (Eingangsöffnung/Schlitz) um es empfangen zu können. Ein Slot wird im Header File deklariert und kann private oder public sein. Die Qt Klassen der Widgets stellen verschiedene Slots zur Verfügung aber es besteht auch die Möglichkeit selber einen Slot zu erstellen.

Klasse	Slot	Erklärung
QApplication	quit()	Sagt der Application exit mit return 0 Link: http://doc.qt.io/qt-5/qcoreapplication.html#public-slots
QWidget	setEnabled()	
QPushButton	setText()	

Beispiel Signal strukturiertes Programmieren (innerhalb vom selben Source File.cpp)

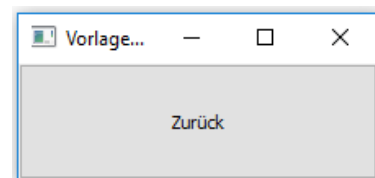
Code :

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

int main(int argc, char *argv[])
{
    QApplication TestApp(argc, argv);
    QPushButton *testButton = new QPushButton ("Zurück");
    QObject::connect(testButton, SIGNAL(clicked()),&TestApp, SLOT(quit()));

    testButton->show();
    return TestApp.exec();
}
```

Ausgabe:



Funktionsbeschreibung:

Wenn der Taster „**Zurück**“ gedrückt wird, dann wird das Programm mit dem Public SLOT „**quit()**“ der Klasse QApplication beendet

Analyse

Code	Erklärung
#include QPushButton	Bibliothek in Projekt aufnehmen Link im Netz: http://doc.qt.io/qt-5/qpushbutton.html
QObject::connect(testButton, SIGNAL(clicked()),&TestApp, SLOT(quit()));	Slot wenn man im selben Source File ist : <p>QObject::connect Mit connect werden zwei Objekte verbunden. QObject ist die oberste Klasse aller Qt Objekte, heisst an der Schreibweise :: ist ersichtlich, dass connect eine Methode von QObject (oberste Klasse in Qt) ist.</p> <p>(testButton, SIGNAL(.....),&TestApp, SLOT(.....)); Verbinden testButton & TestApp. Der SLOT von TestApp sendet ein Signal an das Widget testButton. Das SIGNAL entspricht einer Botschaft und der SLOT ist eine Funktion.</p> <p>clicked() Ist ein Signal auf welches reagiert werden soll und ist eine Klasse von QPushButton.</p> <p>quit() Ist eine Funktion die Child von QApplication ist</p>

Beispiel Signal objektorientiertes Programmieren

main.cpp

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

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

    QApplication TestApp(argc, argv);

    myQuit testObject1;
    testObject1.show();

    myQuit testObject2;
    testObject2.show();

    return TestApp.exec();
}
```

header.h

```
#ifndef MYQUIT_H
#define MYQUIT_H

#include <QWidget>
#include <QPushButton>
#include <QApplication>

class myQuit : public QWidget
{
    Q_OBJECT

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

};

#endif
```

implementation.cpp

```
#include "myQuit.h"

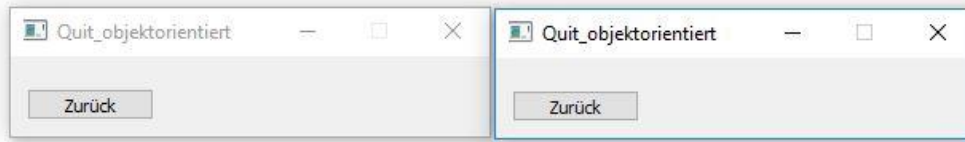
myQuit::myQuit(QWidget *parent):QWidget(parent){
    setFixedSize(300, 50);
    QPushButton *quitButton = new QPushButton("Zurück", this);
    quitButton->setGeometry(10,20,80,20);

    QApplication::connect( quitButton, SIGNAL(clicked()),
                           QApplication::instance(), SLOT(quit()));
}

myQuit::~myQuit(){
}
```

Funktionsbeschreibung:

Es werden 2 Objekte mit Taster erstellt. Wenn der Taster „**Zurück**“ bei einem von beiden gedrückt wird, dann wird das Programm mit dem Puplic SLOT „**quit()**“ der Klasse QApplication beide Fenster beendet.



myQuit testObjectn;

erstellen eines Objektes
testObject aus der Klasse
myQuit

testObjectn.show();

Anzeigen des Objektes

WICHTIG:

Bei Objekten von Klassen
den Punkt Operator nehmen,
bei Widget den Pfeil Operator
verwenden.

myQuit(QWidget *parent =0);

Konstruktor mit KEINEM Parent

~myQuit();

Dekonstruktor

myQuit::myQuit(QWidget *parent):QWidget(parent)

Konstruktor erzeugt Objekt und das Parent Attribut wird
mit dem eigenen Parent vom QWidget (Fenster)
initialisiert. In diesem Fall kein Parent.

setFixedSize(300, 50);

Fenster mit fixer Grösse erstellen

**QPushButton *quitButton = new QPushButton("Zurück",
this);**

Taster mit Beschriftung „Zurück“ erstellen. Mit „this“ wird
das QWidget als Parent vom Taster definiert und somit
wird wenn das QWidget geschlossen wird auch
automatisch der Taster geschlossen.

quitButton->setGeometry(10,20,80,20);

Taster im QWidget positionieren und Grösse festlegen

**QApplication::connect(quitButton, SIGNAL(clicked()),
QApplication::instance(),**

SLOT(quit()));

Verbindung vom Signal clicked() des Tasters mit der Slot
Funktion quit() vom TestApp. Mittels der statischen
Funktion QApplication *QApplication::instance() wird die
QApplication Instanz (die höchste in diesem Programm)
mitgegeben.

Beispiel Signal vorwärtsdeklariert mit eigenem Slot objektorientiertes Programmieren

main.cpp

```
#include "myQuit.h"

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

    QApplication TestApp(argc, argv);

    myQuit testObject1;
    testObject1.show();

    return TestApp.exec();
}
```

header.h

```
#ifndef MYQUIT_H
#define MYQUIT_H

#include <QWidget>
#include <QApplication>

class QPushButton;
class myQuit : public QWidget
{
    Q_OBJECT

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

private:
    QPushButton *quitButton;

signals:
    //INFO: use "clicked(bool checked)" signal

public slots:
    void mySlot (bool checked);
};

#endif
```

implementation.cpp

```
#include "myQuit.h"
#include <QPushButton>

myQuit::myQuit(QWidget *parent):QWidget(parent){
    setFixedSize(300, 50);
    quitButton = new QPushButton("Zurück", this);
    quitButton->setGeometry(10,20,80,20);
    quitButton->setCheckable(true);

    QApplication::connect(quitButton, SIGNAL (clicked(bool)),
                           this, SLOT (mySlot(bool)));
}

myQuit::~myQuit(){

}

void myQuit::mySlot (bool checked)
{
    if (checked) {
        quitButton->setText("Checked");
    }

    if (!checked) {
        quitButton->setText("Hello World");
    }
}
```

Funktionsbeschreibung:

Es wird beim ausführen ein Schalter mit dem Text „Zurück“ generiert. Der Taster hat jetzt den BOOL Wert 0



Beim ersten betätigen wird der Schalter Wert auf 1 gestellt und der Text Checked (Bool == 1) wird angezeigt



Beim ersten betätigen wird der Schalter Wert auf 1 gestellt und der Text Checked (Bool == 1) wird angezeigt



myQuit testObjectn;

erstellen eines Objektes
testObject aus der Klasse
myQuit

testObjectn.show();

Anzeigen des Objektes

WICHTIG:

Bei Objekten von Klassen
den Punkt Operator nehmen,
bei Widget den Pfeil Operator
verwenden.

class QPushButton;

Vorwärtsdeklaration der Klasse
QPushButton

myQuit(QWidget *parent = 0);

Konstruktor mit KEINEM Parent

~myQuit();

Dekonstruktor

private:

QPushButton *quitButton;

Pointer für den PushButton
weil vorwärts deklariert.

public slots:

void mySlot (bool checked);

Slot-Funktion deklarieren

#include <QPushButton>

Einfügen der QPushButton Library. Diese muss in der
Implementierungsdatei eingefügt werden damit die Datei
die Library inkludiert hat weil im Header „nur“ die Klasse
mit dem Schlüsselwort „class“ bekanntgegeben wurde
aber nicht die Klasse inkludiert wurde (vorwärts
deklariert)

myQuit::myQuit(QWidget *parent):QWidget(parent)

Konstruktor erzeugt Objekt und das Parent Attribut wird
mit dem eigenen Parent vom QWidget (Fenster)
initialisiert. In diesem Fall kein Parent.

setFixedSize(300, 50);

Fenster mit fixer Grösse erstellen

quitButton = new QPushButton("Zurück", this);

Taster mit Beschriftung „Zurück“ vorwärtsdeklariert
erstellen. Mit „this“ wird das QWidget als Parent vom
Taster definiert und somit wird wenn das QWidget
geschlossen wird auch automatisch der Taster
geschlossen.

quitButton->setGeometry(10,20,80,20);

Taster im QWidget positionieren und Grösse festlegen

quitButton->setCheckable(true);

Den Taster als Schalter definieren wobei folgendes gilt:
checked == 1, Schalter aktiv (blau hinterlegt)
not checked == 0, Schalter inaktiv (normal grau)

**QApplication::connect(quitButton, SIGNAL(clicked()),
this, SLOT(mySlot()));**

Verbindung vom Signal clicked() des Schalters mit der
eigenen Slot-Funktion „mySlot“.

void myQuit:: mySlot (bool checked) { ... }

Slot-Funktion welche ausgeführt wird sobald der Schalter
betätigt wird.

16. Widget & Item-Widgets Methoden

Qt stellt verschiedenen Funktionen für ein QWidget zur Verfügung. Diese können auch online unter dem folgenden Link angeschaut werden: <http://doc.qt.io/qt-5/search-results.html?q=QWidget>

Code	Widget	Beschreibung
<code>myWidget->setFixedSize(300, 50);</code>	fast alle Widgets	QWidget
<code>myWidget->show() ;</code>	fast alle Widgets	pass auf alle Widget's, sobald jedoch verschieden Widget's in einem Layout sind, so müssen das Show nur auf das finale Widget Objekt angewendet werden.
<code>myWidget->setWindowTitle("QString");</code>	QWidget	Titel vom Fenster vergeben
<code>myWidget -> setFixedSize(x,y);</code>		Grösse Fenster fixieren mit Pixel für x (waagrecht) und y (senkrecht)
<code>myWidget->setValue(int);</code>	QSlider QSpinBox QProgressBar	Passt auf alle Anwendungen bei denen ein INT oder Real Wert angezeigt oder ausgewertet wird.
<code>myWidget->setOrientation(Qt::Horizontal);</code>	QSlider QProgressBar	Anzeigen Horizontal
<code>myWidget->setOrientation(Qt::Vertical);</code>	QSlider QProgressBar	Anzeigen Vertikal
<code>myWidget->setRange(min int, max int);</code>	QSlider QSpinBox QProgressBar	Passt auf alle Anwendungen bei denen ein INT oder Real Wert angezeigt oder ausgewertet wird.
<code>myWidget ->setGeometry(x, y, w, h);</code>	fast alle Widgets	Grösse und Position auf dem Monitor, im Window, Widget oder Layout bestimmen
<code>myWidget ->setCheckable(true) ;</code>	QPushButton	Aus dem Taster wird ein Schalter gemacht. Taster: geht immer automatisch zurück, nur solange gedrückt wie der User die Maustaste drückt Schalter: Wechselt mit jedes Mal beim Anklicken die andere Position.
<code>mylayout ->addWidget(myButtonWidget) ;</code>	alle Widgets	ein Widget wie zum Bsp. ein QPushButton in ein Widget setzen
<code>mylayout ->addLayout(myButtonLayout) ;</code>		ein Layot mit Buttons in ein anderes Layout einfügen
<code>myWidget ->setLayout(myLayout) ;</code>	QHBoxLayout QVBoxLayout	ein fertiges Layout im Widget oder Window setzen
<code>myWidget -> setMinimumSize(640, 480);</code>	QWindows QWidgets	Kann nicht kleiner gemacht werden
<code>myQLineEdit ->setText(„my Text“);</code>	QLineEdit	Setzen vom Text im LineEdit Feld
<code>QString myStr = myQLineEdit ->Text();</code>	QLineEdit	Lesen vom Text in LineEdit

17. QString

Folgend sind die meist gebrauchten QString Methoden aufgeführt. Wenn eine Methode nicht aufgeführt ist, so kann man den folgenden Link verwenden: <http://doc.qt.io/qt-5/qstring.html>

Wandeln in QString:

```
QString myStr = QString::number(myInt)  
oder  
QString myStr;  
myStr.setNum(i);
```

Wandeln Int to QString

```
double myDouble = myStr.toDouble();
```

Wandeln Double to QString

Wandeln String in:

```
int myInt = xStr.toInt();  
oder  
bool ok = true ;  
int myInt1 = myStr.toInt(&ok, 10) //für dec  
int myInt 2= myStr.toInt(&ok, 16) //für hex  
if(!ok)  
....
```

Wandeln QString to Int

wenn mit &OK gearbeitet wird, so muss auch das OK erstellt werden!

```
double myDouble = QString::toDouble(myStr)
```

Wandeln QString to Double

Sonstige:

```
bool ok = true ;  
double myDouble = myStr.remove(QRegExp(".*")).toDouble(&ok);  
if(!ok)  
    //generate fault text in QLineEdit
```

Wandeln QString to Double und abschneiden ab Punkt nach hinten

18. CheckBox

Die CheckBox ist verwandt mit QPushButton und hat deshalb auch ungefähr die selben abgeleiteten Funktion und Methoden zur Verfügung.

Code Signal:

Code :

```
#include <QApplication>
#include <QCheckBox>

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

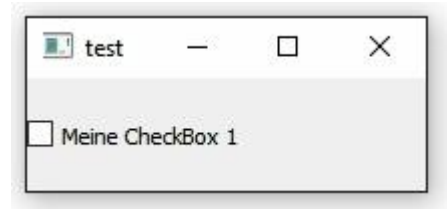
    QCheckBox *testCheckBox = new QCheckBox ("Meine CheckBox 1");

    QObject::connect(testCheckBox, SIGNAL(clicked()),
                    &TestApp, SLOT(quit()));

    testCheckBox->show();

    return TestApp.exec();
}
```

Ausgabe:



Code Analyse

Code	Erklärung
#include <QCheckBox>	Bibliothek in Projekt aufnehmen Link im Netz: http://doc.qt.io/qt-5/qcheckbox.html

Signale	Erklärung
clicked()	Wenn der PushButton gedrückt (aktiviert) wird, wird es gesendet. Zum Beispiel wenn der Maus Cursor innerhalb des PushButton ist.

Andere Funktionen / Methoden

Funktion	Erklärung
stateChanged()	Sobald der Status von der CheckBox in den anderen zustand wechselt wird dieses Signal gesendet. Es gibt folgende zweu Zustände: <ul style="list-style-type: none">▪ Checked▪ Unchecked

19. RadioButton

Ist wie die CheckBox zu verwenden, hat einfach ein paar Funktionen und Methoden weniger.

Code Signal:

Code :

```
#include <QApplication>
#include <QRadioButton>

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

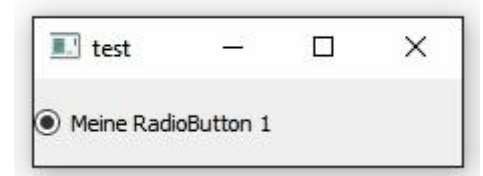
    QRadioButton *testRadioButton = new QRadioButton ("Meine RadioButton 1");

    QObject::connect(testRadioButton, SIGNAL(clicked()),
                    &TestApp, SLOT(quit()));

    testRadioButton->show();

    return TestApp.exec();
}
```

Ausgabe:



Code Analyse

Code	Erklärung
#include <QRadioButton>	Bibliothek in Projekt aufnehmen Link im Netz: http://doc.qt.io/qt-5/qradiobutton.html

SIGNALS:

Signale	Erklärung
clicked()	Wenn der PushButton gedrückt (aktiviert) wird, wird es gesendet. Zum Beispiel wenn der Maus Cursor innerhalb des PushButton ist.

Andere Funktionen / Methoden

Funktion	Erklärung
stateChanged()	Sobald der Status von der CheckBox in den anderen zustand wechselt wird dieses Signal gesendet. Es gibt folgende zweu Zustände: <ul style="list-style-type: none">▪ Checked▪ Unchecked

20. QPushButton

Siehe auch Link: <http://pyqt.sourceforge.net/Docs/PyQt4/qpushbutton.html>

QPushButton sendet ein *clicked()* Signal wenn der Bediener den Button betätigt. Das Signal sendet einen INT Wert zurück, der als „state“ anzuschauen ist und kein BOOL.

Über den Slot wird das Signal beim Betätigen übertragen und entsprechend die programmierte Aktion ausgeführt.

Code Signal (innerhalb vom selben Source File):

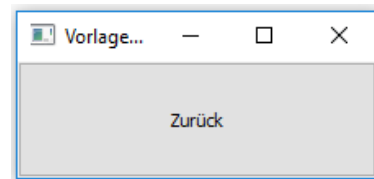
Code :

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

int main(int argc, char *argv[])
{
    QApplication TestApp(argc, argv);
    QPushButton *testButton = new QPushButton ("Zurück");
    QObject::connect(testButton, SIGNAL(clicked()),&TestApp, SLOT(quit()));

    testButton->show();
    return TestApp.exec();
}
```

Ausgabe:



Code Analyse

Code	Erklärung
<code>#include <QPushButton></code>	Bibliothek in Projekt aufnehmen Link im Netz: http://doc.qt.io/qt-5/qpushbutton.html
<code>QPushButton *testButton = new QPushButton ("Zurück");</code>	PushButton mit der Beschriftung „Zurück“ erstellen
<code>QObject::connect(testButton, SIGNAL(clicked()),&TestApp, SLOT(quit()));</code> <code>connect(testButton,SIGNAL(clicked()), QApplication::instance(), SLOT(quit()));</code>	Verbinden vom SIGNAL clicked mit dem SLOT quit(). Beendet das Programm.
<code>testButton->show();</code>	Anzeigen vom PushButton

SIGNALS:

Signale	Erklärung
<code>clicked()</code>	Wenn der PushButton gedrückt (aktiviert) wird, wird es gesendet. Zum Beispiel wenn der Maus Cursor innerhalb des PushButton ist.
<code>pressed()</code>	Sendet wenn Button gedrückt wird
<code>released()</code>	Sendet wenn Button losgelassen wird
<code>toogled()</code>	Sendet immer wenn der Button den Zustand wechsel aber er muss als <code>setcheckable()</code> (als Schalter) definiert sein! <code>checked = true</code> wenn Schalter gedrückt <code>checked = false</code> wenn Schalter zurückgesetzt

Andere Funktionen / Methoden

Funktion	Erklärung
<code>anyButton->setCheckable(true);</code>	Mit <code>setCheckable</code> wird der PushButton zu einem : <code>true</code> = Schalter <code>false</code> = Taster

21. Slider

Das Widget ermöglicht das Einstellen eines Wertes über einen Schieberegler in einem definierten Bereich.

Code :

```
#include <QApplication>
#include <QHBoxLayout>
#include <QSlider>

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

    QWidget *testWindow = new QWidget();
    testWindow->setWindowTitle("Enter your age");

    QSlider *testSlider = new QSlider;
    testSlider->setRange(0,100);

    QObject::connect(testSlider, SIGNAL(valueChanged(int)),
                     testWindow, SLOT(setValue(int)));
    testSlider->setValue(77);

    QHBoxLayout *testLayout=new QHBoxLayout;

    testLayout->addWidget(testSlider);
    testWindow->setLayout(testLayout);

    testWindow->show();

    return TestApp.exec();
}
```

```
... QSlider *testSlider = new QSlider(Qt::Horizontal,0);
...
```

```
... QSlider *testSlider = new QSlider(Qt::Vertical,0);
...
```

Ausgabe:

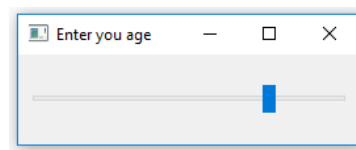


Wenn beim Erstellen des QSliders nichts zur Ausrichtung angegeben wird, dann wird automatisch ein vertikaler Slider erstellt.

Wichtig:

Die Ausrichtung kann nicht mit dem QHBoxLayout oder QVBoxLayout gemacht werden!

Das Widget Slider wird nur im QWidget angezeigt, wenn es mittels einem Layout dem QWidget zugeordnet wird!



Wenn die Ausrichtung definiert sein soll, so hat dies über eine Funktion von Qt zu geschehen, in diesem Fall Horizontal



Wenn die Ausrichtung definiert sein soll, so hat dies über eine Funktion von Qt zu geschehen, in diesem Fall Vertical

Code Analyse

Code	Erklärung
#include <QSlider>	Bibliothek in Projekt aufnehmen Link im Netz: http://doc.qt.io/qt-5/qslider.html
QSlider *testSlider = new QSlider(Qt::Horizontal,0);	Erstellen eines neuen Widget Slider Horizontal
testSlider->setRange(0,100);	Einen Bereich min, max zuordnen
QObject::connect(testSlider, SIGNAL(valueChanged(int)), testWindow, SLOT(setValue(int)));	SLOT und SIGNAL definieren
testSlider->setValue(77);	Voreinstellen des Sliders auf den Wert 77

Signal:	
actionTriggered(int action)	<p>Eine Aktion wird auf dem Slider ausgeführt. Mögliche Aktionen sind:</p> <p> SliderSingleStepAdd SliderSingleStepSub SliderPageStepAdd SliderPageStepSub SliderToMinimum SliderToMaximum SliderMove </p>
rangeChanged(int min, int max)	Der Slider Range wurde geändert
sliderMoved(int value)	Der Slider wurde mit der Maus bewegt
sliderPressed()	Den Slider mit der Maus angeklickt
sliderReleased()	Den Slider mit der Maus losgelassen
valueChanged(int value)	Der Wert vom Slider hat geändert

22. ProgressBar

QProgressBar ist eine visuelle Anzeige auf wieviel der Wert zwischen dem Minimum und Maximum steht.

Code :

```
#include <QApplication>
#include <QProgressBar>
#include <QSlider>

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

    QWidget window;
    window.setFixedSize(200, 80);

    QProgressBar *progressBar = new QProgressBar(&window);
    progressBar->setRange(0, 100);
    progressBar->setValue(0);
    progressBar->setGeometry(10, 10, 180, 30);

    QSlider *slider = new QSlider(&window);
    slider->setOrientation(Qt::Horizontal);
    slider->setRange(0, 100);
    slider->setValue(0);
    slider->setGeometry(10, 40, 180, 30);

    window.show();

    QObject::connect(slider, SIGNAL (valueChanged(int)),
        progressBar, SLOT (setValue(int)));

    return app.exec();
}

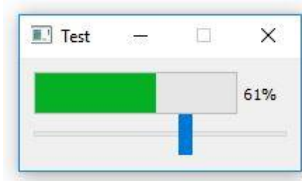
...
QProgressBar *progressBar = new QProgressBar (Qt::Horizontal);
...
```

```
...
QProgressBar *progressBar = new QProgressBar (Qt::Vertical);
...
```

Ausgabe:

Wenn beim Erstellen des QSliders nichts zur Ausrichtung angegeben wird, dann wird automatisch ein vertikaler Slider erstellt.

Wenn nichts zur Ausrichtung angegeben wird, so wird der Balken horizontal angezeigt und dazu standardmässig die % Anzeige.

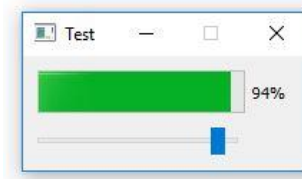


Wichtig:

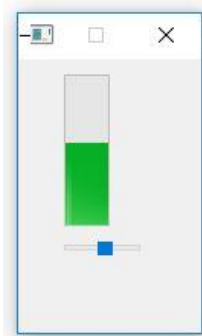
Die Ausrichtung kann nicht mit dem QHBoxLayout oder QVBoxLayout gemacht werden!

Das Widget Slider wird nur im QWidget angezeigt, wenn es mittels einem Layout dem QWidget zugeordnet wird!

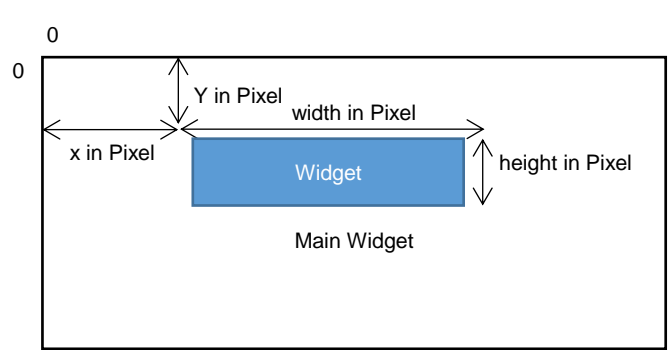
Wenn die Ausrichtung definiert sein soll, so hat dies über eine Funktion von Qt zu geschehen, in diesem Fall Horizontal mit %Anzeige.



Wenn die Ausrichtung definiert sein soll, so hat dies über eine Funktion von Qt zu geschehen, in diesem Fall Vertical. Die Prozentanzeige wird nicht mehr angezeigt. Diese muss speziell wieder erstellt werden.



Code Analyse

Code	Erklärung
<code>#include <QProgressBar></code>	Bibliothek in Projekt aufnehmen Link: http://doc.qt.io/qt-5/qprogressbar.html
<code>QProgressBar *myProgressBar = new QProgressBar(&window);</code>	Erstellen eines neuen ProgressBar bei dem der Parent das QWidget window ist
<code>myProgressBar ->setValue(n);</code>	Wert vorgeben der angezeigt werden soll
<code>myProgressBar ->setOrientation(Qt::Horizontal);</code> <code>myProgressBar ->setOrientation(Qt::Vertical);</code>	Ausrichtung angeben, ohne setOrientation ist es automatisch horizontal
<code>myProgressBar ->setRange(min, max);</code>	Der Wertebereich den der ProgressBar haben soll angeben
<code>int i = myProgressBar ->value();</code>	Auslesen des aktuellen Wertes in eine Variable
<code>myProgressBar ->setGeometry(x, y, w, h);</code>	Positionieren des Widget 

Signal:

`valueChanged()` Sendet wenn der Wert ändert

Slot:

<code>reset()</code>	ProgressBar wird zurückgesetzt, es wird kein wert mehr angezeigt bis ein neuer geschrieben wird.
<code>setMaximum(int maximum)</code>	ProgressBar setzen max Wert
<code>setMinimum(int minimum)</code>	ProgressBar setzen min Wert
<code>setOrientation(Qt::Orientation)</code>	Ausrichtung setzen
<code>setRange(int minimum, int maximum)</code>	ProgressBar setzen min und max Wert
<code>setValue(int value)</code>	ProgressBar setzen aktueller Wert

23. SpinBox

Die Widgets von Qt senden „*Signale*“ zum Anzeigen, dass eine Bediener Aktion oder eine Änderung

Code :

```
#include <QApplication>
#include <QHBoxLayout>
#include <QSpinBox>

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

    QWidget *testWindow = new QWidget;
    testWindow->setWindowTitle("Enter you age");

    QSpinBox *testSpinBox = new QSpinBox;
    testSpinBox->setRange(0, 100);

    QObject::connect(testSpinBox, SIGNAL(valueChanged(int)),
        « Empfänger », SLOT(setValue(int)));

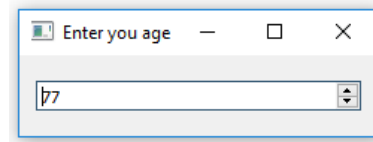
    testSpinBox->setValue(77);

    QHBoxLayout *testLayout=new QHBoxLayout;
    testLayout->addWidget(testSpinBox);
    testWindow->setLayout(testLayout);

    testWindow->show();

    return TestApp.exec();
}
```

Ausgabe:



Wichtig:

Das Widget SpinBox wird nur im QWidget angezeigt, wenn es mittels einem Layout dem QWidget zugeordnet wird!

Code Analyse

Code	Erklärung
<pre>#include <QSpinBox> #include <QDoubleSpinBox></pre>	Bibliothek in Projekt aufnehmen Link im Netz: http://doc.qt.io/qt-5/qspinbox.html Link im Netz: http://doc.qt.io/qt-5/qdoublespinbox.html
<pre>QSpinBox *testSpinBox = new QSpinBox(0, 100); QDoubleSpinBox *testSpinBox = new QDoubleSpinBox ();</pre>	Erstellen eines neuen Widget SpinBox mit min & max Wert INT Erstellen eines neuen Widget SpinBox nicht mit min max Werten möglich! setRange und setDecimal verwenden.
<pre>testSpinBox->setRange(0, 100); // Int und Double testSpinBox-> setDecimals(2); // nur für Double testSpinBox -> setSingleStep(0.1) // Int und Double</pre>	Einen Bereich min, max setzen, bei DoubleSpinBox die Anzahl Dezimal Stellen setzen und bestimmen Schrittgrösse.
<pre>testSpinBox -> setMinimum(1);</pre>	Den min Bereich setzen
<pre>mySpinBox2 -> setMaximum(99);</pre>	Den max Bereich setzen
<pre>QObject::connect(testSpinBox, SIGNAL(valueChanged(int)), testWindow, SLOT(setValue(int)));</pre>	SLOT und SIGNAL definieren
<pre>testSpinBox->setValue(77);</pre>	Voreinstellen der SpinBox auf den Wert 77
<pre>int i = testSpinBox ->value();</pre>	Auslesen vom Wert

Signal :

valueChanged(int i)
valueChanged(const QString &text)

Der INT Wert in der SpinBox hat sich geändert
Der QString in der SpinBox hat sich geändert

Slot :

setValue(int val)

Der INT Wert setzen

24. Label

Mit einem Label kann in einem Widget ein Text (mehrzeilig), ein Bild, ein QPixmap, ein QString, eine INT Zahl oder ein Film angezeigt werden. Um ein Label in einem Widget auf einer bestimmten Stelle oder Position einzufügen muss ein Layout verwendet.

Code :

```
#include <QApplication>
#include <QLabel>

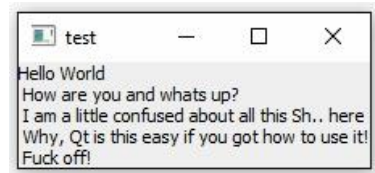
int main (int argc, char *argv[]){
    QApplication VorlagenNamen(argc, argv);

    QLabel *testLabel = new QLabel("Hello World"
                                   "\n\rHow are you and whats up? "
                                   "\n\rI am a little confused about all this Sh.. here"
                                   "\n\rWhy, Qt is this easy if you got how to use it!"
                                   "\n\rFuck off!");

    testLabel->show();

    return VorlagenNamen.exec();
}
```

Ausgabe:



Wichtig:

Bei Merzeiligen Text muss mit den Steuersignalen
 \n (newline)
 \r (carriage return)
im QString gearbeitet werden, wobei immer die
Reihenfolge \n\r verwendet wird.

Wichtig zu Wissen:

- QPicture = ist eine QPainter Klasse welches in Qt „gezeichnet“ wurde.
- QPixmap = ist um ein Pixel orientiertes Bild (jpg, bmp, png, ...) anzuzeigen
- QImage = ist gemacht um I/O und direkten Pixel Zugriff und Manipulation

Code Analyse

Code	Erklärung
#include QLabel	Bibliothek in Projekt aufnehmen Link im Netz: http://doc.qt.io/qt-5/qlabel.html
QLabel *testLabel = new QLabel("Hello Qt");	Erstellen eines neuen Labels mit dem Text „Hello Qt“
testLabel->show();	Anzeigen vom Label ohne positionieren in einem Widget

Signal:

linkActivated(const QString &link)
linkHovered(const QString &link)

Signal wird gesendet wenn der Link angeklickt wird
Signal wird gesendet wenn der User mit der Maus über / um den Link herum geht.

Slot:

clear()
setText();
setMovie(QMovie *movie)
setNum(int num)
setNum(double num)
setPicture(const QPicture &picture)
setPixmap(const QPixmap &)

Löscht den gesamten Label inhalt (egal was es ist)
Schreibt einen QString als Text in das Label

Schreibt eine INT Zahl als Text in das Label
Schreibt eine DOUBLE Zahl als Text in das Label
Zeigt ein DrawEvent Picture an
Zeigt ein Pixelbezogenes (jpg, bmp, png, ...) Bild an

25. QLineEdit

QLineEdit ist ein Ein- und Ausgabefeld welches jedoch nur QString aufnehmen kann. Ganzzahlen und Gleitpunktzahlen müssen jeweils in QString gewandelt werden.

Code :

main.cpp

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

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

    Widget myWidget;
    myWidget.setWindowTitle("Widget");
    myWidget.show();

    return a.exec();
}
```

Header widget.h

```
#include <QWidget>
#include <QObject>

class QGridLayout;
class QLineEdit;
class QPushButton;

class Widget : public QWidget
{
    Q_OBJECT

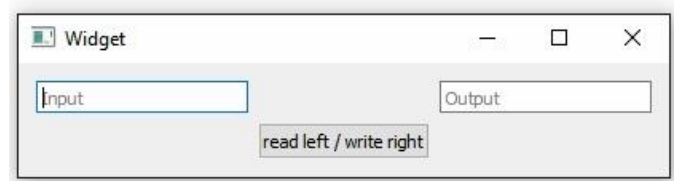
public:
    Widget(QWidget *parent = 0);

private:
    QGridLayout *myGridLayout;
    QLineEdit *myLineEdit1;
    QLineEdit *myLineEdit2;
    QPushButton *myPushButton;

    QString stringLine1;
    QString stringLine2;

public slots:
    void slotWriteRead(bool check);
};
```

Ausgabe:



Implementierung widget.cpp

```
include "widget.h"

#include <QGridLayout>
#include <QLineEdit>
#include <QPushButton>
#include <QString>

Widget::Widget(QWidget *parent) : QWidget(parent)
{
    myGridLayout = new QGridLayout();
    myLineEdit1 = new QLineEdit();
    myLineEdit2 = new QLineEdit();
    myPushButton = new QPushButton("read left / write right");
    myPushButton -> setCheckable(1);

    myGridLayout -> addWidget(myLineEdit1, 0, 0);
    myGridLayout -> addWidget(myLineEdit2, 0, 2);
    myGridLayout -> addWidget(myPushButton, 1, 1);

    setLayout(myGridLayout);

    myLineEdit1 -> setPlaceholderText("Input");
    myLineEdit2 -> setPlaceholderText("Output");
    myLineEdit2 -> setReadOnly(1);

    connect(myPushButton, SIGNAL(clicked(bool)),
            this, SLOT(slotWriteRead(bool)));
}

void Widget::slotWriteRead(bool check)
{
    if(check)
    {
        myPushButton -> setText("delete line");
        stringLine1 = myLineEdit1 -> text();
        myLineEdit2 -> setText(stringLine1);
    }
    if(!check)
    {
        myPushButton -> setText("read left / write right");
        myLineEdit1 -> clear();
        myLineEdit2 -> clear();
    }
}
```

Code

```
#include <QLineEdit>
```

```
myQLineEdit ->setText(„my Text“);
QString myStr = myQLineEdit ->Text();
myLineEdit -> setInputMask(QStringMask);
```

```
myLineEdit -> setMaxLength(maxLengthInt);
myLineEdit -> setPlaceholderText(myString);
myLineEdit -> setReadOnly(1) //true = read only - false = default
myString = myLineEdit -> hasSelectedText();
myLineEdit -> clear();
```

Wandeln QString to & to QString

```
QString myStr = QString::number(myInt)
oder
QString myStr;
myStr.setNum(i);
```

```
double myDouble = myStr.toDouble();
```

```
int myInt = xStr.toInt();
oder
bool ok = true ;
int myInt1 = myStr::toInt(&ok, 10) //für dec
int myInt 2= myStr::toInt(&ok, 16) //für hex
if(!OK)
....
```

```
double myDouble = QString::toDouble(myStr)
```

```
bool ok = true ;
double myDouble = myStr.remove(QRegExp(".*")).toDouble(&ok);
if(!ok)
    //generate fault text in QLineEditit
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qlineedit.html>

Text in Feld schreiben

Text aus Feld auslesen

Im Feld eine Eingabe Maske erzwingen. Für mehr siehe auch Link: <http://doc.qt.io/qt-5/qlineedit.html#inputMask-prop>

Maximale Länge vom Text definieren

Platzhalter wenn leer definieren (grau im Feld)

Feld auf „nur lesen“ setzen

Ausgewählter Text (im Feld blau) auslesen

Feldlöschen

Wandeln Int to QString

Wandeln Double to QString

Wandeln QString to Int

Wandeln QString to Double

Wandeln QString to Double und abschneiden ab Punkt nach hinten

Signal:

```
cursorPositionChanged(int old, int new)
editingFinished()
```

```
returnPressed()
selectionChanged()
textChanged(const QString &text)
```

```
textEdited(const QString &text)
```

Das Signal wird ausgelöst wenn der Cursor sich im Feld bewegt

Das Signal wird ausgelöst wenn die Eingabe im Feld mit Enter bestätigt wird oder man das Feld mit der Cursor verlässt

Das Signal wird ausgelöst wenn die Eingabe im Feld mit Enter bestätigt wird

Das Signal wird ausgelöst wenn der ausgewählte Text ändert

Das Signal wird ausgelöst wenn sich der Text im Feld ändert. Die Eingabe kann mit newText() aufgefangen werden.

Das Signal wird ausgelöst wenn der Text im Feld geändert wird. Die Eingabe kann mit newText() aufgefangen werden.

Slot:

```
clear()
copy()
cut()
```

```
paste()
redo()
selectAll()
setText(myQString)
undo()
```

Löscht die eingabe im Feld

Kopiert die Eingabe zum Zwischenspeicher falls eines vorhanden ist

Kopiert die Eingabe zum Zwischenspeicher falls eines vorhanden ist und löscht die Eingabe im Feld

Fügt den Inhalt aus dem Zwischenspeicher in das Feld ein

Führt die letzte Handlung im Feld nochmal aus

Wählt den Text im Feld aus und markiert ihn
schreibt den Text in das Feld

Macht die letzte Eingabe im Feld rückgängig

26. TextEdit

QTextEdit ist ein Text Editorfeld, welches wie ein normaler Editor genutzt werden kann. So kann man Texte reinschreiben, kopieren und ersetzen, Bilder, Tabellen und Listen einfügen usw. Ausserdem wird automatisch ein Zeilenumbruch eingefügt oder ein ScrollBar erstellt, sobald der Text die Grösse des Feldes überschreitet.

Code :

main.cpp

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

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

    return a.exec();
}
```

Header widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QObject>

class QPushButton;
class QTextEdit;

class Widget : public QWidget
{
    Q_OBJECT

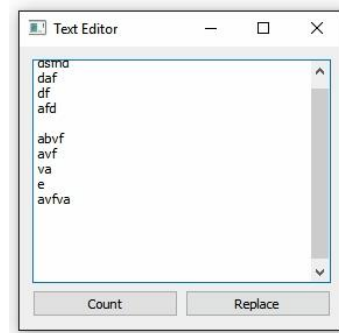
public:
    Widget(QWidget *parent = 0);

private:
    QPushButton *countButton;
    QPushButton *replaceButton;
    QTextEdit *textEditField;

public slots:
    void slotButtonCount();
    void slotButtonReplace();
};

#endif // WIDGET_H
```

Ausgabe:



Implementierung widget.cpp

```
#include "widget.h"

#include <QPushButton>;
#include <QTextEdit>;
#include <QVBoxLayout>;
#include <QFormLayout>
#include <QGridLayout>

Widget::Widget(QWidget *parent) : QWidget(parent)
{
    countButton = new QPushButton("Count");
    replaceButton = new QPushButton("Replace");
    textEditField = new QTextEdit();

    QFormLayout *formLayoutInput = new QFormLayout();
    QGridLayout *gridLayoutButton = new QGridLayout();
    QVBoxLayout *mainVBoxLayout = new QVBoxLayout();
    gridLayoutButton -> addWidget(countButton, 0, 0);
    gridLayoutButton -> addWidget(replaceButton, 0, 1);
    mainVBoxLayout -> addWidget(textEditField);
    mainVBoxLayout -> addLayout(gridLayoutButton);

    this -> setLayout(mainVBoxLayout);
    this -> setWindowTitle("Text Editor");

    connect(countButton, SIGNAL(clicked()),
            this, SLOT(slotButtonCount()));

    connect(replaceButton, SIGNAL(clicked()),
            this, SLOT(slotButtonReplace()));
}

void Widget::slotButtonCount()
{
}

void Widget::slotButtonReplace()
{
}
```


Code

```
#include <QTextEdit>
```

```
textEditField -> insertPlainText("Hallo du schöne Welt\r\n"
                                "Wie gehts es so?");
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qtextedit.html>

Plain Text einfügen (Text Editor)

\r = CR (carriage return)

\n = NL (new line)

Kann auch verwendet werden um auf CSV, txt, usw in einen Stream zu lesen und dann den Stream in das TextEdit Feld einzufügen.

Signal:

```
copyAvailable(bool yes)
currentCharFormatChanged(const
QTextCharFormat &f)
cursorPositionChanged()
redoAvailable(bool available)
selectionChanged()
textChanged()
undoAvailable(bool available)
```

Slot:

```
append(const QString &text)
clear()
copy()
cut()
insertHtml(const QString &text)
insertPlainText(const QString &text)
paste()
redo()
scrollToAnchor(const QString &name)
selectAll()
setAlignment(Qt::Alignment a)
setCurrentFont(const QFont &f)
setFontFamily(const QString &fontFamily)
setFontItalic(bool italic)
setFontPointSize(qreal s)
setFontUnderline(bool underline)
setFontWeight(int weight)
setHtml(const QString &text)
setPlainText(const QString &text)
setText(const QString &text)
setTextBackgroundColor(const QColor &c)
setTextColor(const QColor &c)
undo()
zoomIn(int range = 1)
zoomOut(int range = 1)
```

27. ComboBox

QComboBox ist ein ausklappbares Menu das nur mit Eingabe vom Typ QString gefüllt werden kann.

Code :

```
#include <QApplication>
#include <QComboBox>
#include <QString>

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

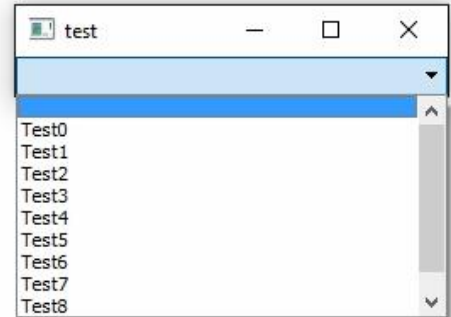
    QComboBox *testComboBox = new QComboBox();
    QString myStr = "";

    testComboBox -> addItem("");      // einfügen einer leeren Zeile

    for(int i = 0; i < 10; i++){
        {
            myStr = "Test" + QString::number(i);
            testComboBox -> addItem(myStr);
        }
    }
    testComboBox -> show();

    return VorlagenNamen.exec();
}
```

Ausgabe:



Code

```
#include QComboBox
```

```
QString tmp = QString::number(myInt)
oder
QString myStr;
myStr.setNum(i);
QString tmp = ComboBox -> currentText()
testComboBox -> addItem(myStr);
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qcombobox.html>

Wandeln Int to QString

Lesen aus ComboBox

In ComboBox schreiben, Zeile für Zeile möglich (For Schleife)

Signal:

activated(int index)

activated(const QString &text)

currentIndexChanged(int index)

currentIndexChanged(const QString &text)

currentTextChanged(const QString &text)

editTextChanged(const QString &text)

highlighted(int index)

highlighted(const QString &text)

Signal wird gesendet sobald der User Element aus der ComboBox auswählt. Die gesendete INT entspricht der Position in der ComboBox und nicht dem Wert!

Signal wird gesendet sobald der User ein QString Element aus der ComboBox ausgewählt wird.

Signal wird gesendet sobald das INT Element gewechselt wird

Signal wird gesendet sobald das QString Element gewechselt wird

Signal wird gesendet sobald der Text im Element durch den User wechselt

Signal wird gesendet sobald der Text in der LineEdit oben wechselt

Signal wird gesendet sobald ein INT Element aus der Liste blau hinterlegt ist wenn man mit der Maus darüber fährt

Signal wird gesendet sobald ein QString Element aus der Liste blau hinterlegt ist wenn man mit der Maus darüber fährt

Slot:

clear()

clearEditText()

setCurrentIndex(int index)

setCurrentText(const QString &text)

setEditText(const QString &text)

Löscht den gesamten Inhalt der ComboBox (alle Zeilen)

Löscht nur den Inhalt der ComboBox LineEdit Zeile

Gibt den Wert zurück den eine Ausgewählte Zeile in der ComboBox hat. Am Beispiel „Test 5“ oben wäre dies 7, wenn kein Element ausgewählt ist wird -1 zurückgemeldet.

Gibt den angezeigten Inhalt der EditLine zurück

Schreibt den angezeigten Inhalt der EditLine

28. ScrollArea

Code

```
#include <QApplication>
#include <QVBoxLayout>
#include <QLabel>
#include <QScrollArea>

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

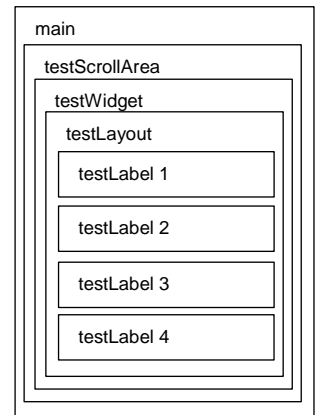
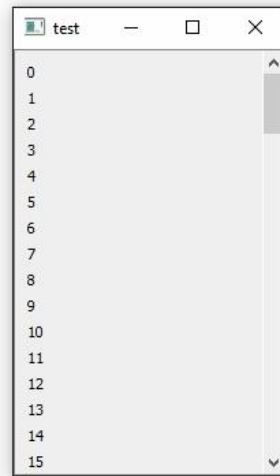
    QWidget *testWidget = new QWidget();
    QVBoxLayout *testLayout = new QVBoxLayout();
    QScrollArea *testScrollArea = new QScrollArea();

    for(int i = 0; i < 100; i++){
        {
            QLabel *testLabel = new QLabel();
            testLabel->setNum(i);
            testLayout->addWidget(testLabel);
        }
    }
    testWidget -> setLayout(testLayout);
    testScrollArea -> setWidget(testWidget);

    testScrollArea->show();

    return VorlagenNamen.exec();
}
```

Erklärung



Aufbau:

Beim ScrollArea muss der schrittweise Aufbau entsprechend eingehalten werden. Anstelle der Labels können auch Buttons, LineEdit oder sonst irgendwas verwendet werden.

```
#include <QApplication>
#include <QVBoxLayout>
#include <QLabel>
#include <QScrollArea>

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

    QWidget *mainWidget = new QWidget();
    QVBoxLayout *mainLayout = new QVBoxLayout();

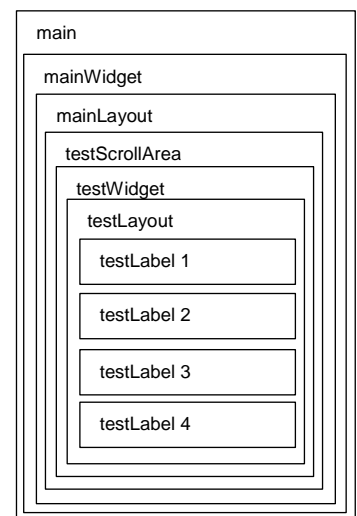
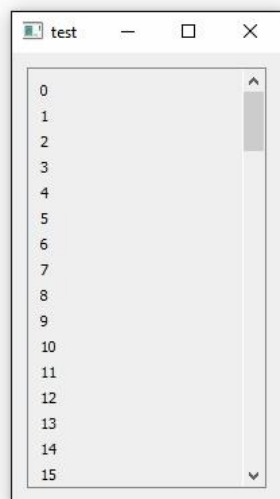
    QWidget *testWidget = new QWidget();
    QVBoxLayout *testLayout = new QVBoxLayout();
    QScrollArea *testScrollArea = new QScrollArea();

    for(int i = 0; i < 100; i++){
        {
            QLabel *testLabel = new QLabel();
            testLabel->setNum(i);
            testLayout-> addWidget(testLabel);
        }
    }
    testWidget -> setLayout(testLayout);
    testScrollArea -> setWidget(testWidget);

    mainLayout -> addWidget(testScrollArea);
    mainWidget -> setLayout(mainLayout);

    mainWidget->show();

    return VorlagenNamen.exec();
}
```



Code

```
#include <QApplication>
#include <QVBoxLayout>
#include <QLabel>
#include <QScrollArea>
#include <QString>

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

    QWidget *mainWidget = new QWidget();
    QVBoxLayout *mainLayout = new QVBoxLayout();
    QLabel *testLabel = new QLabel();
    QWidget *testWidget = new QWidget();
    QVBoxLayout *testLayout = new QVBoxLayout();
    QScrollArea *testScrollArea = new QScrollArea();
    QString testStr = "\n\rTest";

    for(int i = 0; i < 100; i++){
        {
            testStr = testStr + "\n\rTest";
            testLabel->setText(testStr);
        }
        testLayout -> addWidget(testLabel);
        testWidget -> setLayout(testLayout);
        testScrollArea -> setWidget(testWidget);

        mainLayout -> addWidget(testScrollArea);
        mainWidget -> setLayout(mainLayout);

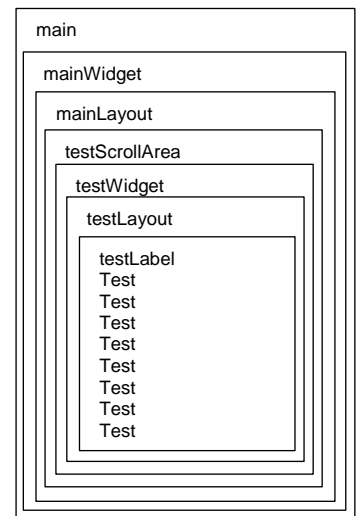
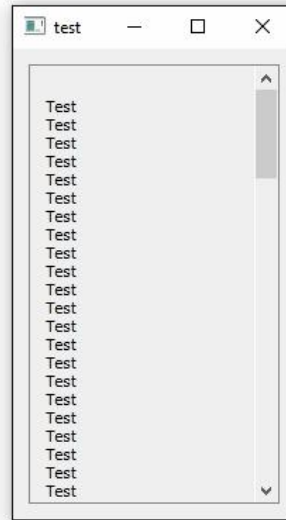
        mainWidget->show();

        return VorlagenNamen.exec();
    }
}
```

Code:

```
#include <QComboBox>
```

Erklärung



Aufbau:

Es kann auch ein kompletter Text in nur einem Label mit einem ScrollArea versehen werden aber auch hier ist die richtige Reihenfolge wichtig.

Beschreibung:

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qscrollarea.html>

<http://doc.qt.io/qt-5/qabstractscrollarea.html#scrollContentsBy>

Für das ScrollArea stehen keine Signale oder Slot's zur Verfügung was schliesslich ja auch nicht wirklich Sinn machen würde.

29. TableWidget

Mit TableWidget können Tabellen erstellt werden. Eine Tabellen Zelle kann auch mit Widget gefüllt werden. Um Tabelle mit Daten (QString) zu füllen müssen QTableWidgetItem's erstellt werden.

Code :

main.cpp

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

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

    w.show();

    return a.exec();
}
```

Header widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QObject>

class QTableWidgetItem;
class QTableWidgetItem;
class QPushButton;
class QLineEdit;
class QSpinBox;
class QLabel;

class Widget : public QWidget
{
    Q_OBJECT

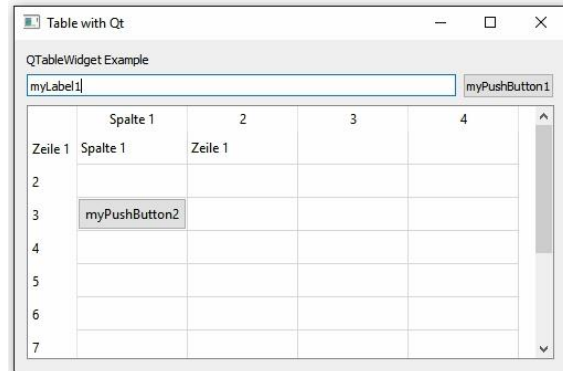
public:
    Widget(QWidget *parent = 0);

private:
    QTableWidgetItem *myTableWidgetItem;
    QTableWidgetItem *myItem1;
    QTableWidgetItem *myItem2;
    QPushButton *myPushButton1;
    QPushButton *myPushButton2;
    QLineEdit *myLineEdit1;
    QLabel *myLabel1;

public slots:
    void slotClicked(bool myClicked);
};

#endif // WIDGET_H
```

Ausgabe:



Implementierung widget.cpp

```
#include "widget.h"

#include <QTableWidget>
#include <QTableWidgetItem>
#include <QPushButton>
#include <QLineEdit>
#include <QLabel>
#include <QHBoxLayout>
#include <QVBoxLayout>

Widget::Widget(QWidget *parent) : QWidget(parent)
{
    this -> setMinimumSize(500, 300);
    this -> setWindowTitle("Table with Qt");

    myTableWidgetItem = new QTableWidgetItem("Hallo");
    myItem1 = new QTableWidgetItem();
    myItem2 = new QTableWidgetItem();
    myPushButton1 = new QPushButton("myPushButton1");
    myPushButton2 = new QPushButton("myPushButton2");
    myLineEdit1 = new QLineEdit("myLabel1");
    myLabel1 = new QLabel("QTableWidget Example");

    QHBoxLayout *myHBoxLayout1 = new QHBoxLayout();
    QVBoxLayout *myVBoxLayout1 = new QVBoxLayout();

    myVBoxLayout1 -> addWidget(myLabel1);
    myHBoxLayout1 -> addWidget(myLineEdit1);
    myHBoxLayout1 -> addWidget(myPushButton1);
    myVBoxLayout1 -> addLayout(myHBoxLayout1);
    myVBoxLayout1 -> addWidget(myTableWidgetItem);

    //Füllen Tabelle
    myItem2 -> setText("World");
    myTableWidgetItem -> setItem(0,0,myItem1); // (Zeile,Spalte,Item)
    myTableWidgetItem -> setItem(0,1,myItem2); // (Zeile,Spalte,Item)
    myTableWidgetItem -> setCellWidget(2,0,myPushButton2); //
    (Zeile,Spalte,Widget)

    //Beschriftung (Header) Tabelle Spalten und Zeilen
    myItem1 -> setText("Spalte 1");
    myTableWidgetItem -> setHorizontalHeaderItem(0, myItem1);
    myItem2 -> setText("Zeile 1");
    myTableWidgetItem -> setVerticalHeaderItem(0, myItem2);

    this -> setLayout(myVBoxLayout1);

    connect(myPushButton1, SIGNAL(clicked(bool)),
            this, SLOT(slotClicked(bool)));
}
```

```
void Widget::slotClicked(bool myClicked)
{
    //Löschen Inhalt exklusiv Beschriftung (Header)
    myTableWidget -> clear();
}
```

Code

```
#include <QTableWidget>
#include <QTableWidgetItem> (siehe QTableWidgetItemItems)

myTableWidget = new QTableWidgetItem(10, 4, this);
myTableWidget = new QTableWidgetItem(10, 4);
myTableWidget = new QTableWidgetItem(this);
myTableWidget -> setRowCount(10);
myTableWidget -> setColumnCount(5);
int tmpRow = myTableWidget -> rowCount();
int tmpColumn = myTableWidget -> columnCount();
QTableWidgetItem *myItem = new QTableWidgetItem(myString);

myTableWidget -> setItem(row - 1, column - 1, myItem);

myTableWidget -> clearContents();

myTableWidget -> clear();

myTableWidget -> setHorizontalHeaderItem(column - 1, myItem);
myTableWidget -> setVerticalHeaderItem(row - 1, myItem);
myTableWidget -> setCellWidget(row - 1, column - 1, myQLabel);

myTableWidget -> removeCellWidget(row - 1, column - 1);

qDebug() << myTableWidget -> cellWidget(row - 1, column - 1);

myTableWidget -> insertRow(1);

myTableWidget -> insertColumn(1);

myTableWidget -> removeRow(1);
myTableWidget -> removeColumn(1);
horizontalHeader()->setResizeMode(QHeaderView::Stretch);
verticalHeader()->setResizeMode(QHeaderView::Stretch);
tableWidget->horizontalHeader()->setResizeMode(8,
QHeaderView::Fixed);
table->resizeColumnsToContents(); //oder nur Column
table->resizeRowsToContents(); //oder nur Row
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qtablewidget.html>

Erstellen Tabelle mit Parent (Zeilen, Spalten, Parent)

Erstellen Tabelle ohne Parent (Zeilen, Spalten)

Erstellen Tabelle nur mit Parent (Parent)

Anzahl Zeilen der Tabelle setzen

Anzahl Spalten der Tabelle setzen

Zählen der aktuellen Anzahl Zeilen in der Tabelle

Zählen der aktuellen Anzahl Spalten in der Tabelle

Erstellen eine Item welches in die Tabelle eingefügt werden kann, siehe auch Kapitel QTableWidgetItem

Einfügen des Item in die Tabelle mit (Zeile, Spalte, QString).

Tabelle Zeile und Spalte startet bei 0, siehe auch Kapitel QTableWidgetItem

Kompletten Tabelleninhalt löschen mit aktiven Selection auf der

Tabelle aber braucht einen Selection auf eine Zelle wo ein Item gesetzt ist sonst Absturz.

Löscht auch die Headers und ersetzt diese nicht durch Nummern!

Kompletten Tabelleninhalt löschen mit Header (werden durch Nummern ersetzt) und aktiven Selection auf der Tabelle

Selections sind QList's (Bereiche), siehe auch

setRangeSelected() und selectedRanges()

Ersetzen der Spaltenbenennung (standard Zahlen) mit irgend einem Text eines Item's

Ersetzen der Zeilenbenennung (standard Zahlen) mit irgend einem Text eines Item's

Einfügen eines Widgets in einer Tabelle (Zeile, Spalte, myWidget).

Achtung:

Ein erstelltes Widget kann nur einmal verwendet werden. Wenn zum Bsp ein myButton1 im Layout verwendet wird und dieser dann in die Tabelle eingefügt wird, so verschwindet er aus dem Layout!

Löschen eines Widgets in einer Tabelle (Zeile, Spalte). Achtung, Objekt wird komplett gelöscht, heisst falls es noch gebraucht wird müsste man es wieder erstellen!

Gibt das Widget zurück in der Tabelle (Zeile, Spalte) mit dem Typ und der Adresse vom Objekt.

Beispiel: QSpinBox(0x1616eda8)

Eine neue zusätzliche Zeile in die Tabelle einfügen (wo?)

Eine neue zusätzliche Spalte in die Tabelle einfügen (wo?)

Eine Zeile aus der Tabelle löschen (welche?)

Eine Spalte aus der Tabelle löschen (welche?)

Zellegrösse nach Spaltennamen anpassen

Zellegrösse nach Zeilenamen anpassen

Zellegrösse nach Spaltennamen FIX anpassen in einer bestimmter Spalte

Siehe auch <http://doc.qt.io/qt-5/qtableview.html> unter Public Slots

Bsp. mit Slot: https://wiki.qt.io/How_to_Use_QTableWidget

Bsp. Zellengrösse: <http://www.codeprogress.com/cpp/libraries/qt/QTableWidgetResizeColumnWidth.php>

30. QTableWidgetItem

QTableWidgetItem sind Inhalt für die QWidget Tabellen. Nur so können in die Tabelle Daten eingefügt werden.

Code :

Ausgabe:

siehe unter QWidget !

Code

```
#include <QTableWidgetItem>
```

```
QTableWidgetItem *myItem = new  
QTableWidgetItem(myString);  
myQWidget -> setItem(row - 1, column - 1, myItem);  
  
myItem -> setText("Hallo");  
QString myString2 = myItem -> text();
```

```
int myRow = myItem -> row();
```

```
int myColumn = myItem -> column();
```

```
myItem1 = myItem2 -> clone();  
myItem -> setIcon(":/icons/remotedata_red.png");
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qtablewidgetitem.html>

Erstellen eine Item welches in die tabelle eingefügt werden kann.

Einfügen des Item in die Tabelle mit (Zeile, Spalte, QString). Tabelle Zeile und Spalte startet bei 0.

Setzen des QString im Item.

Auslesen des Textes (QString) im Item.

Gibt die aktuelle Zeile der Tabelle als INT zurück in welcher das Item eingebettet ist. Rückgabewert -1 heisst Item ist nicht vorhanden.

Gibt die aktuelle Spalte der Tabelle als INT zurück in welcher das Item eingebettet ist. Rückgabewert -1 heisst Item ist nicht vorhanden.

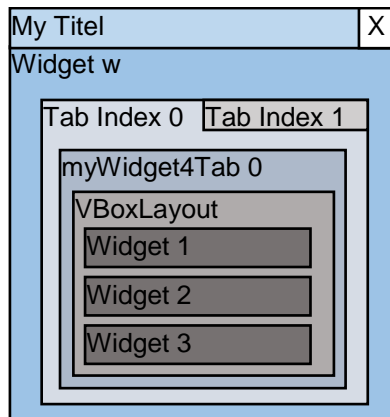
Kopieren von Inhalt Item2 in das Item1.

Einfügen eines Icons, die Grösse kann mit iconSize() definiert werden.

31. TabWidget

Ein TabWidget wird in ein TrägerWidget eingefügt. Jedes Tab braucht selber ebenfalls ein Widget um wieder Widgets aufnehmen zu können.

Skizze:



Code :

main.cpp

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

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

    return a.exec();
}
```

widget.h Header

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QObject>

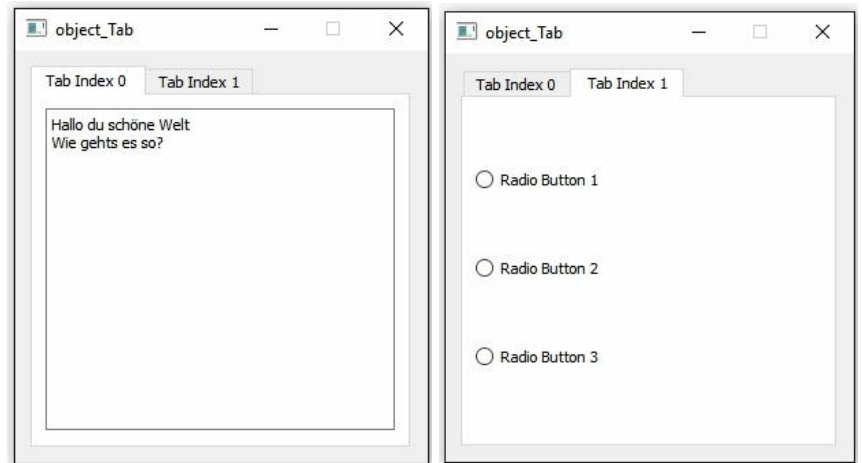
class QTabWidget;
class QVBoxLayout;
class QTextEdit;
class QRadioButton;

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);

private:
    QTabWidget *myTabWidget;
    QWidget *myWidget4Tab0;
    QWidget *myWidget4Tab1;
    QVBoxLayout *myVBoxLayout1;
    QVBoxLayout *myVBoxLayout2;
    QVBoxLayout *myVBoxLayout3;
    QTextEdit *myTextEdit1;
    QRadioButton *myRadioButton1;
    QRadioButton *myRadioButton2;
```

Ausgabe:



widget.cpp Implementierung

```
#include "widget.h"

#include <QTabWidget>
#include <QVBoxLayout>
#include <QTextEdit>
#include <QRadioButton>
#include <QDebug>

Widget::Widget(QWidget *parent) : QWidget(parent)
{
    this->setFixedSize(300, 300);

    myTabWidget = new QTabWidget();
    myWidget4Tab0 = new QWidget();
    myWidget4Tab1 = new QWidget();
    myVBoxLayout1 = new QVBoxLayout();
    myVBoxLayout2 = new QVBoxLayout();
    myVBoxLayout3 = new QVBoxLayout();
    myTextEdit1 = new QTextEdit();
    myRadioButton1 = new QRadioButton("Radio Button 1");
    myRadioButton2 = new QRadioButton("Radio Button 2");
    myRadioButton3 = new QRadioButton("Radio Button 3");

    // Widget für Tab mit "Index 0" erstellen
    myVBoxLayout1 -> addWidget(myTextEdit1);
    myTextEdit1 -> insertPlainText("Hallo du schöne Welt\r\n"
                                   "Wie gehts es so?");
    // \r = CR (carriage return)
```



```

        QRadioButton *myRadioButton3;

public slots:
    void slotOnChanged(int tabNo);
};

#endif // WIDGET_H

// \n = NL (new line)
myWidget4Tab0 -> setLayout(myVBoxLayout1);

// Widget fur Tab mit "Index 1" erstellen
myVBoxLayout2 -> addWidget(myRadioButton1);
myVBoxLayout2 -> addWidget(myRadioButton2);
myVBoxLayout2 -> addWidget(myRadioButton3);
myWidget4Tab1 -> setLayout(myVBoxLayout2);

// zwei Tab's mit vorbereiteten Widgets erstellen
myTabWidget -> addTab(myWidget4Tab0, "Tab Index 0");
myTabWidget -> addTab(myWidget4Tab1, "Tab Index 1");

// Tab in "this" einfügen und Layout setzen
myVBoxLayout3 -> addWidget(myTabWidget);
this -> setLayout(myVBoxLayout3);

// Signal->Slot auslösen wenn Tab gewechselt wird
connect(myTabWidget, SIGNAL(currentChanged(int)),
        this, SLOT(slotOnChanged(int)));

// Index vom aktivem Tab ermitteln
int currentIndex = myTabWidget -> currentIndex();
qDebug() << "Aktives Tab ist: " << currentIndex;
}

void Widget::slotOnChanged(int tabNo)
{
    // Index vom neu gewählten Tab ausgeben
    qDebug() << "Aktives Tab ist: " << tabNo;
}

```

Code

```
#include <QTabWidget>
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qtabwidget.html>

```
myTabWidget = new QTabWidget();
```

```
myTabWidget -> addTab(myWidget4Tab0, "Tab Index 0");
```

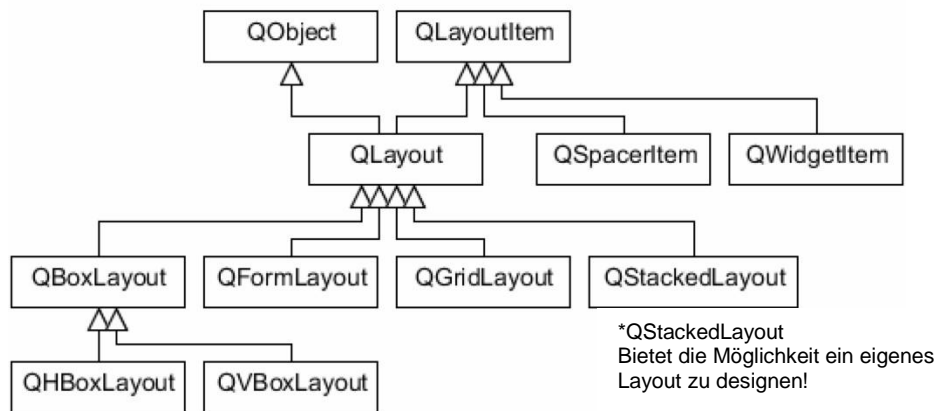
Erstellen eines TabWidget Objektes

Erstellen Tab mit zugeordnetem Widget und QString für die Benennung vom Tab (QWidget, QString)

Siehe Beispiel im Netz: <http://www.codeprogress.com/cpp/libraries/qt/qtQTabWidgetAddTab.php>
<http://doc.qt.io/qt-5/qtwidgts-dialogs-tabdialog-example.html>

32. Layout

Um ein Item-Widget auf einem Träger-Widget organisiert und strukturiert anzuzeigen, so muss dieses zuerst einem Layout zugeordnet werden. Hierzu gilt der folgende Klassenstruktur von Qt:



32.1. Kein Layout

Wenn man kein Layout auswählt, so muss alles selbst mittels Pixelposition usw. konstruiert werden. Wie man folgend erkennen kann ist dies relativ auswendig und mühsam. Ausserdem muss beachtet werden, dass diese Anwendung meist nur in einem Window oder Widget mit einer fixen Grösse Sinn macht ausser es werden die Grösse und Positionen aller enthaltenen Item-Widgets ebenfalls dynamisch programmiert.

Code :

Implementierung (myImplement.cpp)

```
#include <QLineEdit>
#include <QPushButton>

Widget::Widget(QWidget *parent) : QWidget(parent)
{
    QLabel *label1 = new QLabel("Label 1:");
    QLabel *label2 = new QLabel("Label 2:");
    QLabel *label3 = new QLabel("Label 3:");
    QLabel *label4 = new QLabel("Label 4:");
    QPushButton *button1 = new QPushButton("Button 1");

    label1 -> setParent(this);
    label1 -> setGeometry(10,10,50,20);
    label1 -> show();

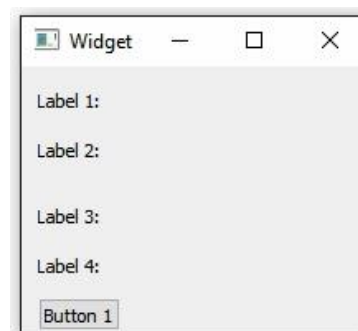
    label2 -> setParent(this);
    label2 -> setGeometry(10,40,50,20);
    label2 -> show();

    label3 -> setParent(this);
    label3 -> setGeometry(10,80,50,20);
    label3 -> show();

    label4 -> setParent(this);
    label4 -> setGeometry(10,110,50,20);
    label4 -> show();

    button1 -> setParent(this);
    button1 -> setGeometry(10,140,50,20);
    button1 -> show();
}
```

Ausgabe:



32.2. QHBoxLayout

ItemWidgets werden in einer Reihe nebeneinander angeordnet

Code :

```
#include <QApplication>
#include <QHBoxLayout>
#include <QSlider>
#include <QSpinBox>

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

    QWidget *testWindow = new QWidget;
    testWindow->setWindowTitle("Enter you age");

    QSpinBox *testSpinBox = new QSpinBox;
    QSlider *testSlider = new QSlider(Qt::Horizontal,0);
    testSpinBox->setRange(0, 100);
    testSlider->setRange(0,100);

    QObject::connect(testSpinBox, SIGNAL(valueChanged(int)),
                     testSlider, SLOT(setValue(int)));
    QObject::connect(testSlider, SIGNAL(valueChanged(int)),
                     testSpinBox, SLOT(setValue(int)));
    testSpinBox->setValue(77);

    QHBoxLayout *testLayout=new QHBoxLayout;
    testLayout->addWidget(testSpinBox);
    testLayout->addWidget(testSlider);
    testWindow->setLayout(testLayout);

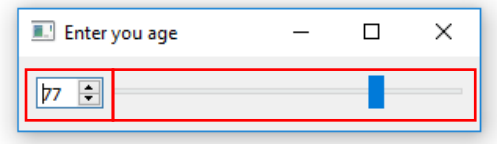
    testWindow->show();

    return TestApp.exec();
}
```

Code

```
#include QHBoxLayout
```

Ausgabe:



Wichtig:

Die beiden Widget SpinBox und Slider werden der Grösse des Fensters automatisch angepasst.

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qhboxlayout.html>

32.3. QVBoxLayout:

ItemWidgets werden in einer Spalte übereinander angeordnet

Code :

```
#include <QApplication>
#include <QVBoxLayout>
#include <QSlider>
#include <QSpinBox>

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

    QWidget *testWindow = new QWidget;
    testWindow->setWindowTitle("Enter you age");

    QSpinBox *testSpinBox = new QSpinBox;
    QSlider *testSlider = new QSlider(Qt::Horizontal,0);
    testSpinBox->setRange(0, 100);
    testSlider->setRange(0,100);

    QObject::connect(testSpinBox, SIGNAL(valueChanged(int)),
                     testSlider, SLOT(setValue(int)));
    QObject::connect(testSlider, SIGNAL(valueChanged(int)),
                     testSpinBox, SLOT(setValue(int)));
    testSpinBox->setValue(77);

    QVBoxLayout *testLayout=new QVBoxLayout;
    testLayout->addWidget(testSpinBox);
    testLayout->addWidget(testSlider);
    testWindow->setLayout(testLayout);

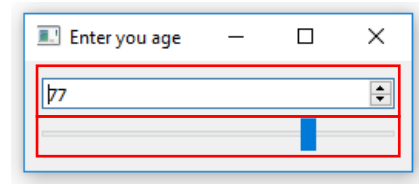
    testWindow->show();

    return TestApp.exec();
}
```

Code

```
#include QHBoxLayout
```

Ausgabe:



Wichtig:

Die beiden Widget SpinBox und Slider werden der Grösse des Fensters automatisch angepasst.

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qvboxlayout.html>

32.4. QHBoxLayout & QVBoxLayout kombiniert

H- und QVBoxLayout können auch miteinander kombiniert werden

Code :

```
#include <QApplication>
#include <QVBoxLayout>
#include <QHBoxLayout>
#include <QSlider>
#include <QSpinBox>

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

    QWidget *testWindow = new QWidget;
    testWindow->setWindowTitle("Enter you age");

    QSpinBox *testSpinBox1 = new QSpinBox;
    QSlider *testSlider1 = new QSlider(Qt::Horizontal,0);

    QSpinBox *testSpinBox2 = new QSpinBox;
    QSlider *testSlider2 = new QSlider(Qt::Horizontal,0);

    QVBoxLayout *testLayoutV1=new QVBoxLayout;
    QVBoxLayout *testLayoutV2=new QVBoxLayout;
    QHBoxLayout *testLayoutH=new QHBoxLayout;

    //Vertikal Layout die zwei Widgets zurodenen
    testLayoutV1->addWidget(testSpinBox1);
    testLayoutV1->addWidget(testSlider1);
    testLayoutV2->addWidget(testSpinBox2);
    testLayoutV2->addWidget(testSlider2);

    //Horizontal Layout das erte mal das Vertikale Layout zuordnen
    testLayoutH->addLayout(testLayoutV1);

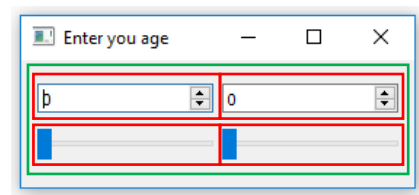
    //Horizontal Layout das zweite mal das Vertikale Layout zuordnen
    testLayoutH->addLayout(testLayoutV2);

    testWindow->setLayout(testLayoutH);

    testWindow->show();

    return TestApp.exec();
}
```

Ausgabe:



Horizontales Layout

Vertikales Layout

Code Analyse H- & QVBoxLayout

Code	Erklärung
<code>QHBoxLayout *testLayout=new QHBoxLayout;</code>	Erstellen eines Layout Horizontal
<code>QVBoxLayout *testLayout=new QVBoxLayout;</code>	Erstellen eines Layout Vertikal
<code>testLayout->addWidget(testSpinBox);</code>	Erstes Widget SpinBox dem Layout zuordnen
<code>testLayout->addWidget(testSlider);</code>	Zweites Widget Slider dem Layout zuordnen
<code>testWindow->setLayout(testLayout);</code>	Dem Haupt Widget das Layout zuordnen

Code

#include QHBoxLayout

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qvboxlayout.html>

32.5. QFormLayout

Mit dem FormLayout werden Zeile für Zeile eingefügt wobei die Möglichkeit besteht in jeder Zeile maximal zweien Item-Widgets einzufügen. Ausserdem besteht die Möglichkeit ohne QLabel mittels eines String Labels zu erstellen.

Wichtig: Es können **max. 2 Spalten** eingefügt werden. Ein String Label immer nur ganz links in der ersten Spalte erstellt werden kann. Wenn man mehrere Labels braucht, so muss dies mittels dem inkludieren von QLabel und daraus erstellten Item-Widget Objekten gemacht werden.

Es entsteht automatisch eine Tabelle in welcher mit verschiedenen Optionen die Darstellung angepasst werden kann:

- [setLabelAlignment\(\)](#)
- [setFormAlignment\(\)](#) bietet wiederum Formatierungsmöglichkeiten: [Qt::AlignHCenter](#), [Qt::AlignTop](#), [::AlignLeft](#), usw...
- [setFieldGrowthPolicy\(\)](#)
- [setRowWrapPolicy\(\)](#)

Dies ergibt die Möglichkeit das Layout den verschrienen Betriebssysteme wie Linux, Windows, Max OS anzupassen

Beispiel: `formLayout -> setRowWrapPolicy(QFormLayout::DontWrapRows);`
`formLayout -> setFieldGrowthPolicy(QFormLayout::FieldsStayAtSizeHint);`
`formLayout -> setFormAlignment(Qt::AlignHCenter | Qt::AlignTop);`
`formLayout -> setLabelAlignment(Qt::AlignLeft);`

Code :

```
#include <QApplication>
#include <QFormLayout>
#include <QLabel> //nur Notwendig fuer Label3 und 4!
#include <QPushButton>
#include <QLineEdit>

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

    QWidget *testWindow = new QWidget;
    testWindow -> setWindowTitle("Test FormLayout");

    QPushButton *button1 = new QPushButton("Button 1");
    QPushButton *button2 = new QPushButton("Button 2");
    QLineEdit *lineEdit1 = new QLineEdit();
    QLineEdit *lineEdit2 = new QLineEdit();
    QLabel *label3 = new QLabel("Label 3:");
    QLabel *label4 = new QLabel("Label 4:");

    QFormLayout *formLayout = new QFormLayout();
    formLayout -> addRow("Label 1: ", button1);
    formLayout -> addRow("Label 2: ", button2);
    formLayout -> addRow(label3, lineEdit1);
    formLayout -> addRow(label4, lineEdit2);

    testWindow->setLayout(formLayout);

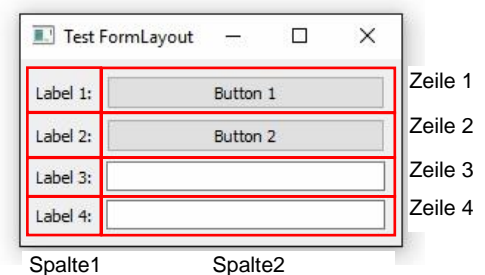
    testWindow->show();

    return TestApp.exec();
}
```

Code

#include QFormLayout

Ausgabe:



WICHTIG:

include QLabel ist nicht Notwendig wenn das Label wie in Zeile 1 und 2 vergeben wird!

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qformlayout.html>

32.6. QGridLayout

Das GridLayout ist mehr oder weniger dasselbe wie das FormLayout wobei beim Grid Layout viel weniger Formatierungsmöglichkeiten bestehen. Jedoch lassen sich beim QGridLayout mehr als nur zwei Spalten Item-Widgets einfügen.

Wichtig : Im GridLayout lassen sich KEINE QHBoxLayout, QVBoxLayout, FormLayout und auch GridLayout einfügen.

Code :

```
#include <QApplication>
#include <QGridLayout>
#include <QLabel>
#include <QPushButton>
#include <QLineEdit>

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

    QWidget *testWindow = new QWidget;
    testWindow -> setWindowTitle("Test FormLayout");

    QPushButton *button1 = new QPushButton("Button 1");
    QPushButton *button2 = new QPushButton("Button 2");
    QLineEdit *lineEdit1 = new QLineEdit();
    QLineEdit *lineEdit2 = new QLineEdit();
    QLabel *label1 = new QLabel("Label 3:");
    QLabel *label2 = new QLabel("Label 3:");
    QLabel *label3 = new QLabel("Label 3:");
    QLabel *label4 = new QLabel("Label 4:");

    QGridLayout *gridLayout = new QGridLayout();
    gridLayout -> addWidget(label1, 0, 0);
    gridLayout -> addWidget(button1, 0, 1);
    gridLayout -> addWidget(label2, 1, 0);
    gridLayout -> addWidget(button2, 1, 1);
    gridLayout -> addWidget(label3, 2, 0);
    gridLayout -> addWidget(lineEdit1, 2, 1);
    gridLayout -> addWidget(label4, 3, 0);
    gridLayout -> addWidget(lineEdit2, 3, 1);

    testWindow->setLayout(gridLayout);

    testWindow->show();

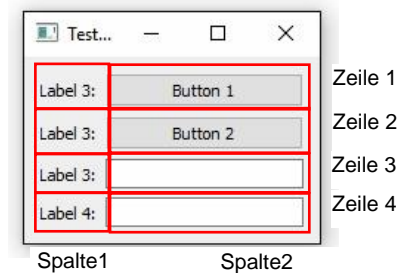
    return TestApp.exec();
}
```

Code

```
#include QGridLayout
```

```
gridLayout -> addWidget(label1, 0, 0);
```

Ausgabe:



WICHTIG:

include QLabel ist nicht Notwendig wenn das Label wie in Zeile 1 und 2 vergeben wird!

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qgridlayout.html>

Syntax: addWidget(*Item-Widget*, *Zeile*, *Spalte*)

32.7. ContentMargin (Inhalt Rahmen)

Sind Item-Widgets einem Träger-Widget zugeteilt so hat das Träger-Widget einen Inhaltsrahmen den man mittels der folgenden Methode „einstellen“ kann. Dies wird gemacht „nachdem“ dem Layout die Widgets zugeordnet wurden.

Code :

```
myWidget -> setLayout(testLayoutH);
```

```
myWidget -> show();
```

```
myWidget -> setLayout(testLayoutH);
```

```
myWidget -> setContentsMargins(20, 30, 40, 50);
```

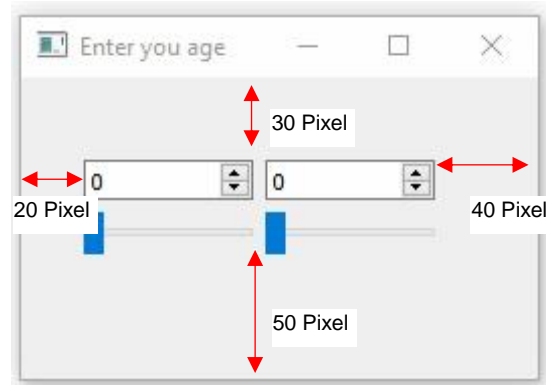
```
testWindow -> show();
```

Ausgabe:

ohne setContentMargin:



mit setContentMargin:



32.8. Spacing (Abstand zwischen den Item-Widgets)

Um den Abstand zwischen den Item-Widget innerhalb eines Layout zu bestimmen muss die „Spacing“ Methode eingesetzt werden. Diese wird dem zwischen allen Item-Widgets den definierten Abstand einstellen.

Code :

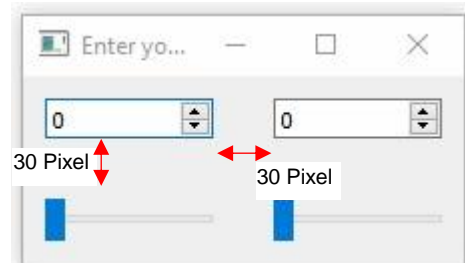
```
testLayoutH -> setSpacing(30);
```

```
testWindow -> setLayout(testLayoutH);
```

```
testWindow -> show();
```

Ausgabe:

mit setSpacing:



32.9. Layout Methoden von QBoxLayout

Siehe auch Link: <http://doc.qt.io/qt-5/qboxlayout.html>

Code

```
boxLayout -> addStretch ( int stretch = 0 )
```

```
boxLayout -> insertStretch ( int index, int stretch = 0 )
```

Erklärung

Im Layout wird der Gesamte verfügbare Platz benutzt.

33. Frame

Ein Frame Widget wird benutzt um den das Kanten Design vom einen Window, Widget oder Item-Widget gegenüber dem Rest anders zu stylen. So besteht die Möglichkeit die Elemente im Frame Widget wie folgt zu definieren:

- Plain (flach, ist die Standardeinstellung in Qt)
- Raised (erhoben, alle Elemente im Frame werden 3D angehoben)
- Sunken (gesenkt, alle Elemente im Frame werden 3D abgesenkt)
-usw (siehe Link: <http://doc.qt.io/qt-5/qframe.html#details>)

Wichtig: Je nach ItemWidget ist es nicht ganz klar wie man den Frame anwenden muss.

Code :

main.cpp

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

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

    MainWindow myMainWindow;
    myMainWindow.show();

    return a.exec();
}
```

Header mainwindow.h

```
#include <QMainWindow>

class QGridLayout;
class QLabel;

class MainWindow : public QMainWindow
{
    Q_OBJECT

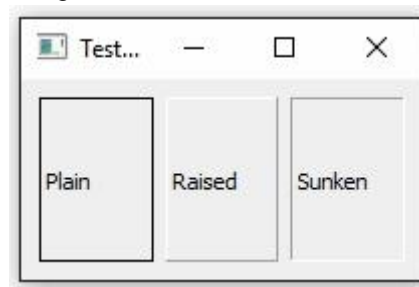
public:
    MainWindow();

private:
    QGridLayout *myLayout;
    QLabel *myLabel1;
    QLabel *myLabel2;
    QLabel *myLabel3;
};
```

Code

```
#include <QFrame>
```

Ausgabe:



Implementierung mainwindow.cpp

```
#include "mainwindow.h"

#include <QGridLayout>
#include <QLabel>
#include <QWidget>

MainWindow::MainWindow()
{
    setWindowTitle("Test Frame");

    QWidget *myWidget = new QWidget();
    setCentralWidget(myWidget);

    myLayout = new QGridLayout();
    myLabel1 = new QLabel("Plain");
    myLabel2 = new QLabel("Raised");
    myLabel3 = new QLabel("Sunken");

    myLabel1 -> setFrameStyle( QFrame::Panel | QFrame::Plain );
    myLabel2 -> setFrameStyle( QFrame::Panel | QFrame::Raised );
    myLabel3 -> setFrameStyle( QFrame::Panel | QFrame::Sunken );

    //Zeile 0
    myLayout -> addWidget(myLabel1, 0, 0);
    myLayout -> addWidget(myLabel2, 0, 1);
    myLayout -> addWidget(myLabel3, 0, 2);

    myWidget -> setLayout(myLayout);
    myWidget -> show();
}
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qframe.html>

34. MesseageBox

Eine MessageBox sind informative Fenster mit Buttons von welchen es vier verschieden Standard ausföhrungen gibt. Die MessageBox können jedoch auch beliebig angepasst werden wie auch unten im letzten Beispiel zu sehen ist.

Beispiele:

#include <QMessageBox>

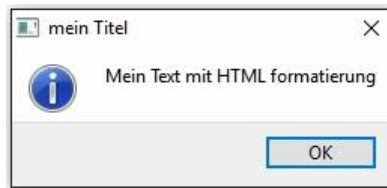
Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qmessagebox.html>

Standard « Information » erstellen

Folgender Code kann irgendwo eingefügt werden um eine Info Box anzuzeigen :

```
QMessageBox::information(this,"mein Titel", "Mein Text mit HTML formatierung");
```



Standard « Question » erstellen

Folgender Code kann irgendwo eingefügt werden um eine Info Box anzuzeigen :

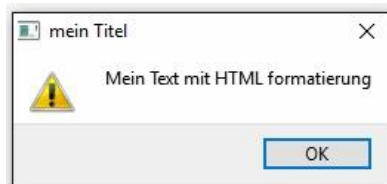
```
QMessageBox::question(this,"mein Titel", "Mein Text mit HTML formatierung");
```



Standard « Warning » erstellen

Folgender Code kann irgendwo eingefügt werden um eine Info Box anzuzeigen :

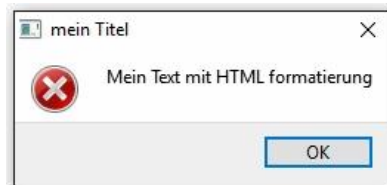
```
QMessageBox::warning(this,"mein Titel", "Mein Text mit HTML formatierung");
```



Standard « Critical » erstellen

Folgender Code kann irgendwo eingefügt werden um eine Info Box anzuzeigen :

```
QMessageBox::critical(this,"mein Titel", "Mein Text mit HTML formatierung");
```



Icon MessageBox setzen

```
#include <QIcon>
```

MessageBox designen

```
#include <QIcon>
```

Info:

Pixmap ist in QIcon enthalten!

Icon an Message Box anhängen:

Wichtig: Das Icon muss über eine Resource dem Projekt angehängt werden, siehe auch unter Kapitel Icon.

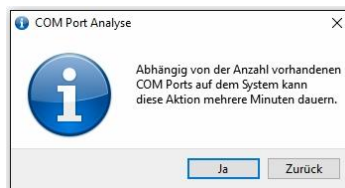
```
QMessageBox myMsgBox;  
myMsgBox.setWindowIcon(QIcon(":/icons/information.png"));
```

oder

```
this -> setWindowIcon(QIcon(":/icons/information.png"));
```

Eigene MessageBox designen

Wichtig: Das Icon und das Bild links vom Text muss über eine Resource dem Projekt angehängt werden, siehe auch unter Kapitel Icon und QPixmap.



Deklaration Header

```
private:  
    QPushButton *myButton;  
  
public slots:  
    void mySlotQMessageBox (bool clicked);
```

Connector im Konstruktor

```
QObject::connect(myButton, SIGNAL(clicked(bool)),  
    this, SLOT(slotQMessageCheckCOMPort(bool)));
```

***Slot ***

```
void meineKlasse::mySlotQMessageBox(bool clicked)  
{  
    QMessageBox msgBox;  
    msgBox.setWindowTitle(tr("Mein Titel"));  
    msgBox.setText(tr("\r\nMein \r\n Text."));  
    msgBox.setIconPixmap(QPixmap(":/icons/information.png"));  
    msgBox.setWindowIcon(QIcon(":/icons/information.png"));  
    msgCheckCOMButtonYes = msgBox.addButton(tr("Ja"), QMessageBox::YesRole);  
    msgCheckCOMButtonNo = msgBox.addButton(tr("Zurück"), QMessageBox::NoRole);  
    msgBox.exec();  
  
    if (msgBox.clickedButton() == msgCheckCOMButtonYes)  
    {  
        qDebug() << "Yes";  
    }  
  
    if (msgBox.clickedButton() == msgCheckCOMButtonNo)  
    {  
        qDebug() << "No"; //Not in use  
    }  
}
```

35. Validator

Der Validator kann auf Eingabe Text QString oder Zahlen vom Typ INT und DOUBLE angewendet werden und prüft ob der Text oder die Zahl gültig (valid) oder ungültig (invalid) ist. Der Validator wird üblicherweise auf den Widgets QLineEdit, SpinBox und ComboBox verwendet.

Beispiel im Netz: <http://ynonperek.com/course/qt/validation.html>

35.1. QLineEdit / QString

Mit dieser Art wird der IntValidator direkt auf das QLineEdit angewendet. Durch diese Anwendung wird der Validator den User davon beschützen überhaupt einen falschen oder zu Grossen String einzugeben.

Code :

main.cpp

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

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

    return a.exec();
}
```

Header widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QObject>

class QLineEdit;
class QRegExpValidator;

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);

private:
    QLineEdit *myLineEdit1;
    QLineEdit *myLineEdit2;
    QRegExpValidator *myStringValidator1;
    QRegExpValidator *myStringValidator2;
};

#endif // WIDGET_H
```

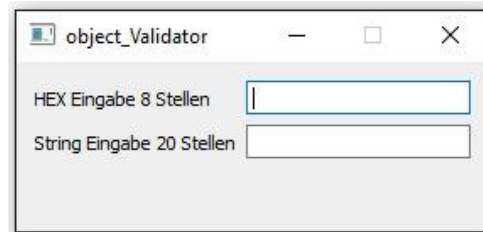
Code

```
#include <QValidator>
#include <QRegExp>

Subklassen von QValidator:
#include <QRegExpValidator>
```

```
"[0-9A-F]{1,8}"
"[0-9A-Za-z]{1,20}"
```

Ausgabe:



Implementierung widget.cpp

```
#include "widget.h"
#include <QLineEdit>
#include <QFormLayout>
#include <QRegExpValidator>
#include <QRegExp>

Widget::Widget(QWidget *parent) : QWidget(parent)
{
    this -> setFixedSize(280, 100);

    //Validator Beispiel vorgehen 1
    myLineEdit1 = new QLineEdit();
    QRegExp myRegExp("[0-9A-F]{1,8}");
    myStringValidator1 = new QRegExpValidator(myRegExp);
    myLineEdit1 -> setValidator(myStringValidator1);

    //Validator Beispiel vorgehen 2
    myLineEdit2 = new QLineEdit();
    myStringValidator2 = new QRegExpValidator(QRegExp ("[0-9A-Za-z]{1,20}"));
    myLineEdit2 -> setValidator(myStringValidator2);

    //Erstellen Layout
    QFormLayout *myLayout = new QFormLayout();
    myLayout -> addRow("HEX Eingabe 8 Stellen", myLineEdit1);
    myLayout -> addRow("String Eingabe 20 Stellen", myLineEdit2);

    this -> setLayout(myLayout);
}
```

Erklärung

Bibliothek in Projekt aufnehmen

Validator Link im Netz:

<http://doc.qt.io/qt-5/qvalidator.html>

QRegExp Link im Netz:

<http://doc.qt.io/qt-5/qregexp.html>

Subklassen von QValidator

QRegExpValidator Link im Netz:

<http://doc.qt.io/qt-5/qregexpvalidator.html>

Zahlen 0 – 9, Grossbuchstaben A – F, Stellen min. 1 max. 8

Zahlen 0 – 9, Grossbuchstaben A – Z & Kleinbuchstaben a – z, Stellen min. 1 max. 20

35.2. QLineEdit / Integer

Mit dieser Art wird der IntValidator direkt auf das QLineEdit angewendet. Durch diese Anwendung wird der Validator den User davon beschützen überhaupt einen falschen Wert einzugeben. Im folgend Fall kann somit keine Wert kleiner als -1000 oder grösser 1000 eingegeben werden.

Im Falle eines Doubles muss „nur“ QDoubleValidator inkludiert werden.

Code :

main.cpp

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

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

    return a.exec();
}
```

Header widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QObject>

class QLineEdit;
class QIntValidator;

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);

private:
    QLineEdit *myIntLineEdit;
    QIntValidator *myIntValidator;
};

#endif // WIDGET_H
```

Code

```
#include <QValidator>
```

Subklassen von QValidator:

```
#include <QIntValidator>
```

```
myIntValidator = new QIntValidator(-1000, 1000);
```

```
myDoubleValidator = new QDoubleValidator(-5.00,
5.00, 2);
```

Ausgabe:



Implementierung widget.cpp

```
#include "widget.h"
#include <QLineEdit>
#include <QFormLayout>
#include <QIntValidator>

Widget::Widget(QWidget *parent) : QWidget(parent)
{
    this -> setFixedSize(150, 100);

    //Validator Beispiel vorgehen 1
    myIntLineEdit = new QLineEdit();
    myIntValidator = new QIntValidator(-1000, 1000);
    myIntLineEdit -> setValidator(myIntValidator);

    //Erstellen Layout
    QFormLayout *myLayout = new QFormLayout();
    myLayout -> addRow("Zahl +/-1000", myIntLineEdit);

    this -> setLayout(myLayout);
}
```

Erklärung

Bibliothek in Projekt aufnehmen

Validator Link im Netz:

<http://doc.qt.io/qt-5/qvalidator.html>

Subklassen von QValidator

QIntValidator Link im Netz:

<http://doc.qt.io/qt-5/qintvalidator.html>

Setzen eines INT Validators auf min. -1000 und max. 1000

Setzen eines DOUBLE Validators auf min. -5.00 und max. 5.00 mit zwei dezimal Stellen.

35.3. Validate Resultat selbst

In diesem Beispiel wird der Validator auf den Text der Eingabe selbst angewendet. Auf diese Weise kann man etwas falsche eingeben und über die Funktion validate() abfangen welchen Status die Eingabe hat.

Folgende INT oder QString Rückgaben sind möglich:

Konstante	Value	Beschreibung
QValidator::Invalid	0	Ungültig
QValidator::Intermediate	1	Der String hat einen pausable Zwischenwert
QValidator::Acceptable	2	Gültig

Code :

main.cpp

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

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

    return a.exec();
}
```

Header widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QObject>

class QLineEdit;
class QIntValidator;
class QLabel;

class Widget : public QWidget
{
    Q_OBJECT

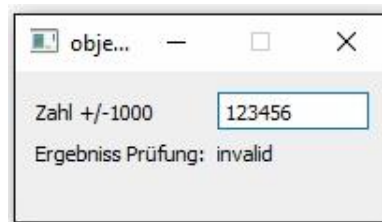
public:
    Widget(QWidget *parent = 0);

private:
    QLineEdit    *myIntLineEdit;
    QIntValidator *myIntValidator;
    QLabel       *myLabel;

public slots:
    void slotTextedChanged(QString myInput);
};

#endif // WIDGET_H
```

Ausgabe:



Implementierung widget.cpp

```
#include "widget.h"
#include <QLineEdit>
#include <QFormLayout>
#include <QIntValidator>
#include <QLabel>
#include <QString>
#include <QDebug>

Widget::Widget(QWidget *parent) : QWidget(parent)
{
    this -> setFixedSize(200, 80);

    myIntLineEdit = new QLineEdit();
    myLabel = new QLabel();
    myIntValidator = new QIntValidator(-1000, +1000);

    QFormLayout *myLayout = new QFormLayout();
    myLayout -> addRow("Zahl +/-1000", myIntLineEdit);
    myLayout -> addRow("Ergebniss Prüfung:", myLabel);

    this -> setLayout(myLayout);

    connect(myIntLineEdit, SIGNAL(textChanged(QString)),
            this, SLOT(slotTextedChanged(QString)));
}

void Widget::slotTextedChanged(QString myInput)
{
    int myInt = myInput.toInt();
    QString myString;
    int stateValid = myIntValidator -> validate(myInput, myInt);

    if(stateValid == 0)
    {
        myLabel -> setText("invalid");
    }

    if(stateValid == 1)
    {
        myLabel -> setText("intermediate value");
    }
}
```

```

if(stateValid == 2)
{
    myLabel -> setText("is valid");
}
}

```

Code

```

#include <QValidator>
#include <QRegExp>

```

Subklassen von QValidator:

```

#include <QIntValidator>
#include <QDoubleValidator>
#include <QRegExpValidator>

```

```
myIntValidator = new QIntValidator(-1000, +1000);
```

```
int stateValid = myIntValidator -> validate(myInput, myInt);
```

Erklärung

Bibliothek in Projekt aufnehmen

Validator Link im Netz: <http://doc.qt.io/qt-5/qvalidator.html>

QRegExp Link im Netz: <http://doc.qt.io/qt-5/qregexp.html>

Subklassen von QValidator

QIntValidator Link im Netz: <http://doc.qt.io/qt-5/qintvalidator.html>

QDoubleValidator Link im Netz: <http://doc.qt.io/qt-5/qdoublevalidator.html>

QRegExpValidator Link im Netz: <http://doc.qt.io/qt-5/qregexpvalidator.html>

erstellen eines Validators mit einem Range min -1000 und max 1000

anwenden des Validators auf myInput (=QString von QLineEdit) und myInt (= Int Wert)

36. Mask

Mit dem setzen einer Maske kann man eine Validation machen und den User dazu zwingen kann sich an vorgaben zu halten. Dies wird mit der Funktion `inputMask()` gemacht welche Member der Klasse `QLineEdit` ist. Folgende Platzhalter stehen zur Verfügung:

Platzhalter	Bedeutung
A	ASCII alphabetic character required. A-Z, a-z.
a	ASCII alphabetic character permitted but not required.
N	ASCII alphanumeric character required. A-Z, a-z, 0-9.
n	ASCII alphanumeric character permitted but not required.
X	Any character required.
x	Any character permitted but not required.
9	ASCII digit required. 0-9.
0	ASCII digit permitted but not required.
D	ASCII digit required. 1-9.
d	ASCII digit permitted but not required (1-9).
#	ASCII digit or plus/minus sign permitted but not required.
H	Hexadecimal character required. A-F, a-f, 0-9.
h	Hexadecimal character permitted but not required.
B	Binary character required. 0-1.
b	Binary character permitted but not required.
>	All following alphabetic characters are uppercased.
<	All following alphabetic characters are lowercased.
!	Switch off case conversion.
\	Use \ to escape the special characters listed above to use them as separators.

Code :

main.cpp

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

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

    return a.exec();
}

Header widget.h

#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QObject>

class QLineEdit;

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);

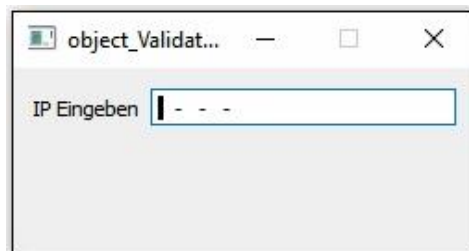
private:
    QLineEdit *myLineEdit;
};
#endif // WIDGET_H
```

Code

```
#include <QLineEdit>
```

```
myLineEdit -> setInputMask("00D-00D-00D-00D");
```

Ausgabe:



Implementierung widget.cpp

```
#include "widget.h"
#include <QLineEdit>
#include <QFormLayout>

Widget::Widget(QWidget *parent) : QWidget(parent)
{
    this -> setFixedSize(250, 100);

    //Erstellen Mask
    myLineEdit = new QLineEdit();
    myLineEdit -> setInputMask("00D-00D-00D-00D");

    //Erstellen Layout
    QFormLayout *myLayout = new QFormLayout();
    myLayout -> addRow("IP Eingeben", myLineEdit);

    this -> setLayout(myLayout);
}
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qlineedit.html#inputMask-prop>

Setzen der Input Maske

37. Completer

Der Completer vervollständigt bei der Eingabe in eines Strings den Text.

Code :

main.cpp

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

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

    return a.exec();
}
```

Header widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QObject>

class QLineEdit;
class QCompleter;

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);

private:
    QLineEdit *myLineEdit;
    QCompleter *myCompleter;
};

#endif // WIDGET_H
```

Code

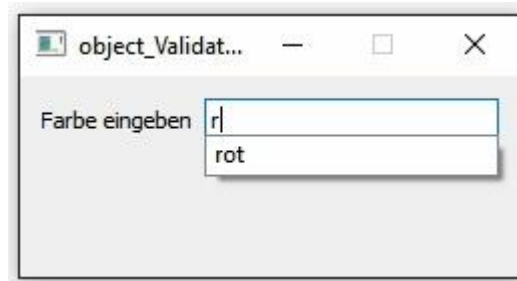
```
#include <QCompleter>

QStringList myStringList;
myStringList << "blau" << "grün" << "rot" << "gelb"
<< "schwarz";

myCompleter = new QCompleter(myStringList,
myLineEdit);

myLineEdit -> setCompleter(myCompleter);
```

Ausgabe:



Implementierung widget.cpp

```
#include "widget.h"
#include <QLineEdit>
#include <QCompleter>
#include <QStringList>
#include <QFormLayout>

Widget::Widget(QWidget *parent) : QWidget(parent)
{
    this -> setFixedSize(250, 100);

    //Erstellen der Kompletierungswörter
    QStringList myStringList;
    myStringList << "blau" << "grün" << "rot" << "gelb" << "schwarz";

    //Erstellen Kompletierung
    myLineEdit = new QLineEdit();
    myCompleter = new QCompleter(myStringList, myLineEdit);

    myLineEdit -> setCompleter(myCompleter);

    //Erstellen Layout
    QFormLayout *myLayout = new QFormLayout();
    myLayout -> addRow("Farbe eingeben", myLineEdit);

    this -> setLayout(myLayout);
}
```

Erklärung

Bibliothek in Projekt aufnehmen
Link im Netz: <http://doc.qt.io/qt-5/qcompleter.html>

StringList für Kennwörter erstellen

Erstellen des Completers

Setzen des Completers

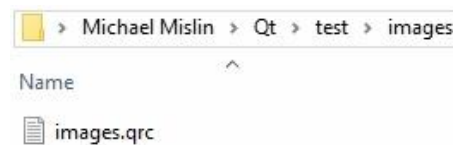
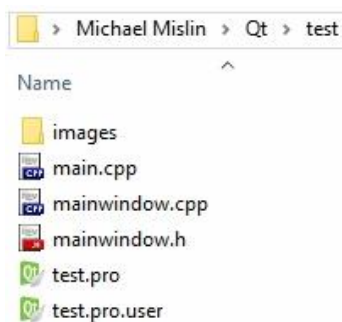
38. Icon

Um ein QIcon in einem Window, Widget, MessageBox, ToolBar, MenuAction, ... usw einzufügen muss dem Projekt zuerst die Ressource zur Verfügung stehen. Dazu muss mittels einer Ressourcen-Datei im Projekt Ordner über die qmake Projekt Datei (*projektName.pro*) gesagt werden welche Ressourcen es gibt und wo diese liegen.

Es können alle Pixelorientierte Dateien wie JPG, ICO, PNG, ... usw verwendet werden jedoch eignet sich PNG am besten da bei diesen der Hintergrund transparent ist.

Das vorgehen ist wie folgt:

1. Irgendwo im Projektordner mit den Source Dateien eine Datei mit dem Namen „*anyName.qrc*“ erstellen. In diesem Fall heisst die Datei „*images.qrc*“ und liegt im Projekt im Ordner „images“.

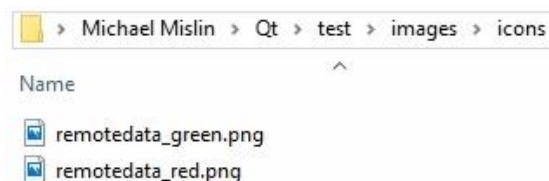
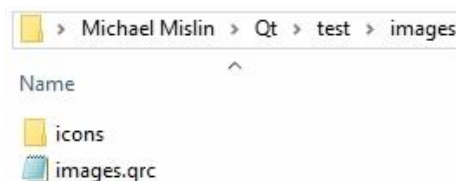


2. Den folgenden Code in die Datei „*images.qrc*“ einfügen wobei die Zeilen die mit `<file>` beginnen auf die Icons zeigen. Der Pfad ist ausgehend vom Ort wo die „*images.qrc*“ Datei liegt.

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>icons/remotedata_green.png</file>
    <file>icons/remotedata_red.png</file>
</qresource>
</RCC>
```

Bei diesem Beispiel werden die zwei Icons „*remotedata_green.png*“ und „*remotedata_red.png*“ eingefügt

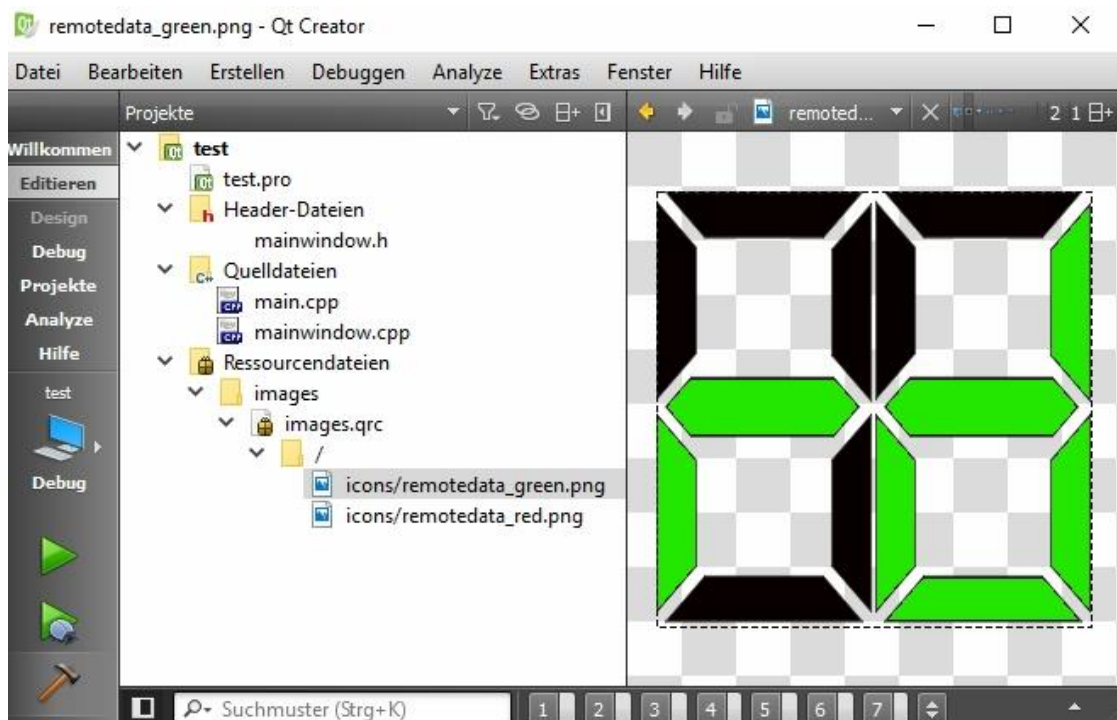
3. Einen Ordner „*icons*“ bei der Datei „*images.qrc*“ erstellen und dort die gewünschten Icons reinkopieren.



4. Jetzt die Ressourcen Datei in der qmake Projekt Datei „*projektName.pro*“ bekannt machen indem der Pfad ausgehend vom Projektordner (hier <test>) auf Datei „*images.qrc*“ zeigt.

```
#-----  
# Project created by QtCreator 2016-05-31T15:56:37  
#-----  
  
QT      += core gui  
  
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets  
  
TARGET = test  
TEMPLATE = app  
  
RESOURCES += images/images.qrc  
  
SOURCES += main.cpp \  
          mainwindow.cpp  
  
HEADERS += \  
          mainwindow.h
```

5. Speichern und neu das qmake ausführen und danach sollte in der Projekt Übersicht die Dateien wie folgt aufgezeigt werden.



Beispiel:

Code :

main.cpp

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

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

    MainWindow myMainWindow;
    myMainWindow.show();

    return a.exec();
}
```

Header mainwindow.h

```
#include <QMainWindow>

class QGridLayout;
class QLabel;
class QLineEdit;
class QProgressBar;

class MainWindow : public QMainWindow
{
    Q_OBJECT

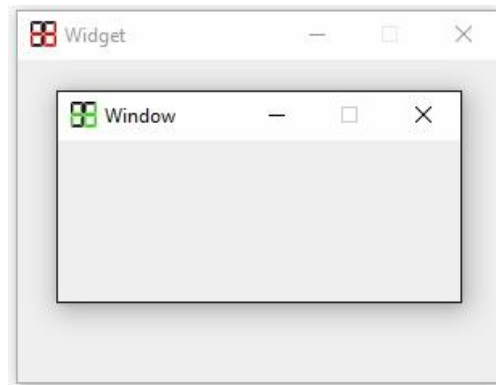
public:
    MainWindow();
};
```

Code

```
#include <QIcon>
```

```
setWindowIcon(QIcon(":/icons/remotedata_red.png"));
```

Ausgabe:



Implementierung mainwindow.cpp

```
#include "mainwindow.h"

#include <QIcon>
#include <QWidget>

MainWindow::MainWindow()
{
    this -> setWindowTitle("Window");
    this -> setFixedSize(250, 100);
    this -> setWindowIcon(QIcon(":/icons/remotedata_green.png"));

    QWidget *myWidget = new QWidget();
    myWidget -> setWindowTitle("Widget");
    myWidget -> setFixedSize(300, 200);
    myWidget -> setWindowIcon(QIcon(":/icons/remotedata_red.png"));

    //setCentralWidget(myWidget);

    myWidget -> show();
}
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qicon.html>

Wichtig ist zu beachten, dass der Pfad immer ausgehend vom Projektordner (hier <test>) auf die „myRessource.qrc“ Datei zeigt.

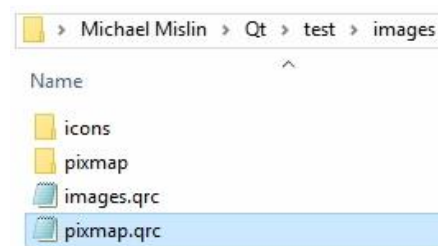
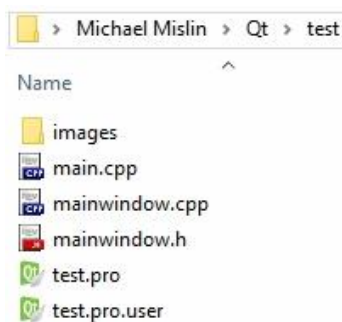
39. Pixmap / Image (Bilder einfügen)

Um bei Qt Bilder einzufügen muss die Methode Pixmap verwendet werden. Die Randbedingungen sind die gleichen wie bei QIcon, heisst es können alle Pixelorientierte Dateien wie JPG, ICO, PNG, ... usw verwendet werden. Im Gegensatz zu QIcon ist die Klasse von QPixmap ausgelegt die Bilder auch zu manipulieren.

Ausgehend davon, dass die Icons wie im vorherigen Kapitel beschrieben eingefügt wurden, muss man folgende Schritte machen um der qmake Projektdatei die Bilder bekannt zu machen:

Das vorgehen ist wie folgt:

1. Irgendwo im Projektordner mit den Source Dateien eine Datei mit dem Namen „*anyName.qrc*“ erstellen. In diesem Fall heisst die Datei „*pixmap.qrc*“ und liegt im Projekt im Ordner „images“.

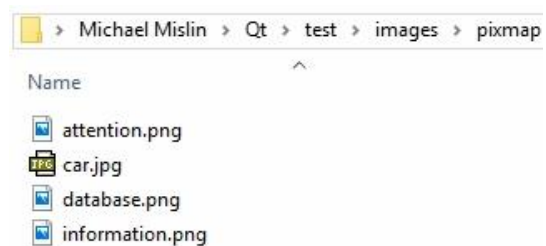
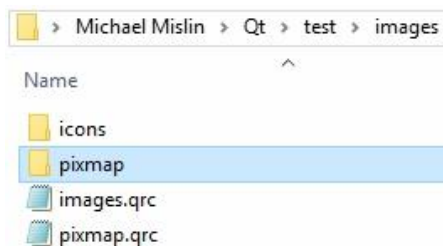


2. Den folgenden Code in die Datei „*pixmap.qrc*“ einfügen wobei die Zeilen die mit `<file>` beginnen auf die Icons zeigen. Der Pfad ist ausgehend vom Ort wo die „*pixmap.qrs*“ Datei liegt.

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
  <file>pixmap/attention.png</file>
  <file>pixmap/database.png</file>
  <file>pixmap/information.png</file>
  <file>pixmap/car.jpg</file>
</qresource>
</RCC>
```

Bei diesem Beispiel werden vier Dateien eingefügt.

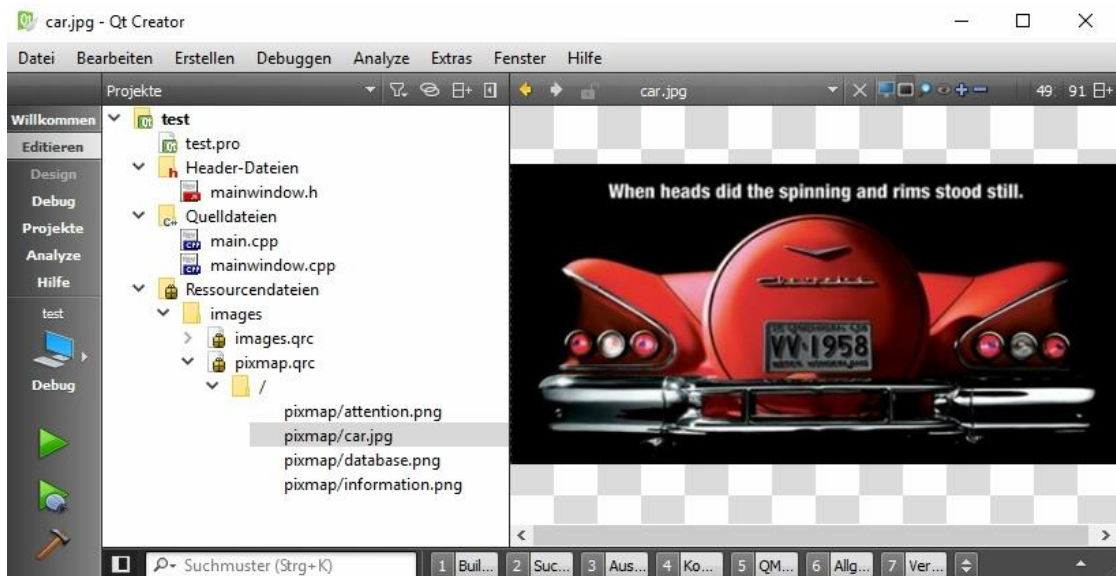
3. Einen Ordner „*pixmap*“ bei der Datei „*pixmap.qrc*“ erstellen und dort die gewünschten Bilder reinkopieren.



4. Jetzt die Ressourcen Datei in der qmake Projekt Datei „*projektName.pro*“ bekannt machen indem der Pfad ausgehend vom Projektordner (hier <test>) auf Datei „*images.qrc*“ zeigt.

```
#-----  
# Project created by QtCreator 2016-05-31T15:56:37  
#-----  
  
QT      += core gui  
  
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets  
  
TARGET = test  
TEMPLATE = app  
  
RESOURCES += images/images.qrc  
RESOURCES += images/pixmap.qrc  
  
SOURCES += main.cpp \  
          mainwindow.cpp  
  
HEADERS += \  
          mainwindow.h
```

5. Speichern und neu das qmake ausführen und danach sollte in der Projekt Übersicht die Dateien wie folgt aufgezeigt werden.



Beispiel mit QPixmap:

Code :

main.cpp

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

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

    MainWindow myMainWindow;
    myMainWindow.show();

    return a.exec();
}
```

Header mainwindow.h

```
#include <QMainWindow>

class MainWindow : public QMainWindow
{
    Q_OBJECT

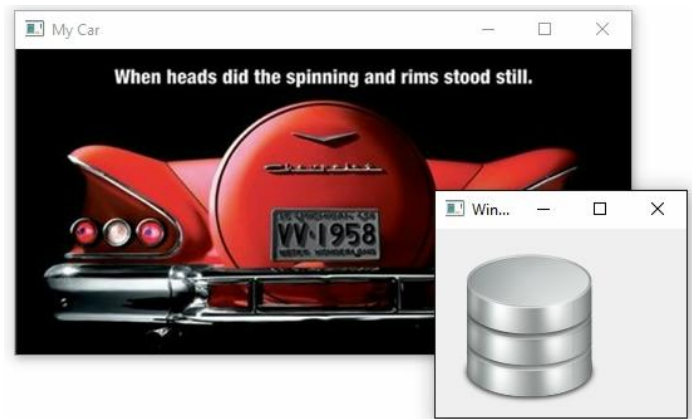
public:
    MainWindow();
};
```

Code

```
#include <QPixmap>
```

```
QLabel *pixmapLabel = new QLabel();
QPixmap mypixmap(":/pixmap/car.jpg");
pixmapLabel->setPixmap(mypixmap);
```

Ausgabe:



Wichtig: Das QPixmap Objekt darf nicht mit als Pointer mit * erstellt werden, weil es sonst nicht mit dem Pfad funktioniert!

Implementierung mainwindow.cpp

```
#include "mainwindow.h"

#include <QPixmap>
#include <QWidget>
#include <QLabel>
#include <QHBoxLayout>

MainWindow::MainWindow()
{ this -> setWindowTitle("Window");

    //mit einem Label in einem Widget
    QWidget *myWidget = new QWidget();
    myWidget -> setWindowTitle("Widget");
    QLabel *widgetLabel = new QLabel();
    QPixmap widgetPixmap(":/pixmap/database.png");
    widgetLabel -> setPixmap(widgetPixmap);

    QHBoxLayout *widgetLayout = new QHBoxLayout();
    widgetLayout -> addWidget(widgetLabel);
    myWidget -> setLayout(widgetLayout);
    myWidget -> show();

    //..oder im Window
    setCentralWidget(myWidget);

    //mit einem Label und abhängig
    QLabel *pixmapLabel = new QLabel();
    QPixmap labelPixmap(":/pixmap/car.jpg");
    pixmapLabel -> setPixmap(labelPixmap);
    pixmapLabel -> setWindowTitle("My Car");
    pixmapLabel -> show();
}
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qpixmap.html>

1. Erstellen eines Label objektes
2. Erstellen eine QPixmap Objektes aber ohne * (Pointer)
3. Dem Label das QPixmap zuordnen

Ab hier kann das Label ganz normal innerhalb von Layouts usw. eingesetzt werden!

Beispiel mit QImage:

Code :

main.cpp

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

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

    MainWindow myMainWindow;
    myMainWindow.show();

    return a.exec();
}
```

Header mainwindow.h

```
#include <QMainWindow>

class MainWindow : public QMainWindow
{
    Q_OBJECT

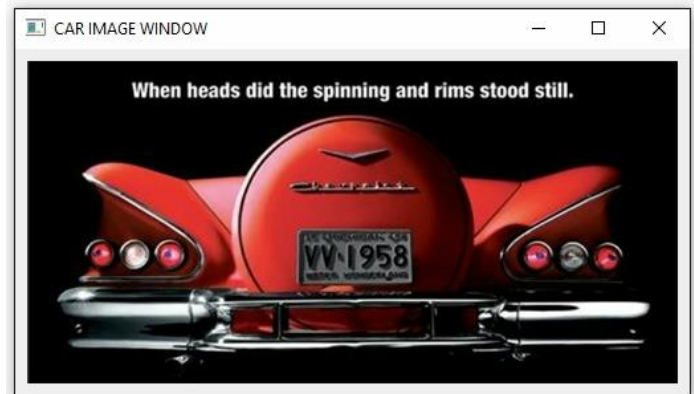
public:
    MainWindow();
};
```

Code

```
#include <QImage>
```

```
QImage image(":/pixmap/car.jpg");
QLabel *myLabel = new QLabel();
myLabel -> setPixmap(QPixmap::fromImage(image));
```

Ausgabe:



Wichtig:

Das QPixmap Objekt darf nicht mit als Pointer mit * erstellt werden, weil es sonst nicht mit dem Pfad funktioniert!

Implementierung mainwindow.cpp

```
#include "mainwindow.h"

#include <QImage>
#include <QWidget>
#include <QLabel>
#include <QHBoxLayout>

MainWindow::MainWindow()
{
    this -> setWindowTitle("CAR IMAGE WINDOW");

    QWidget *myWidget = new QWidget();

    QImage image(":/pixmap/car.jpg");
    QLabel *myLabel = new QLabel();
    myLabel -> setPixmap(QPixmap::fromImage(image));

    QHBoxLayout *myLayout = new QHBoxLayout();

    myLayout -> addWidget(myLabel);
    myWidget -> setLayout(myLayout);

    setCentralWidget(myWidget);
}
```

Erklärung

Bibliothek in Projekt aufnehmen

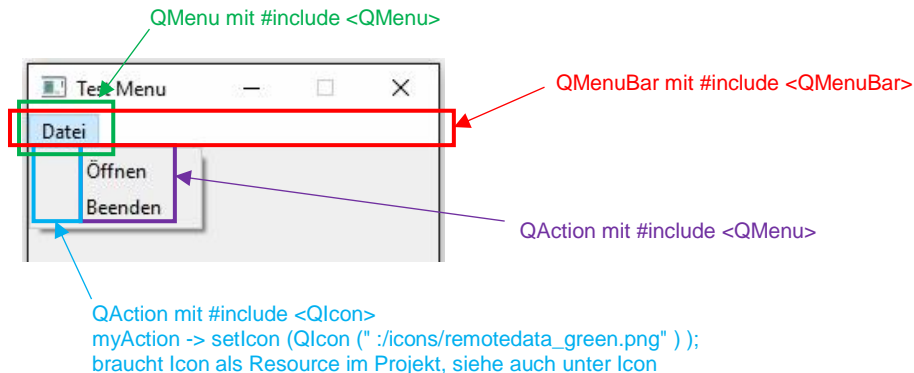
Link im Netz: <http://doc.qt.io/qt-5/qimage.html>

1. Erstellen eines Label objektes
2. Erstellen eine QPixmap Objektes aber ohne * (Pointer)
3. Dem Label das QPixmap zuordnen

Ab hier kann das Label ganz normal innerhalb von Layouts usw. eingesetzt werden!

40. MainWindow

Ein MainWindow ist ein Objekt mit den folgenden Optionen:



Code :

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

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

    MainWindow myMainWindow;
    myMainWindow.show();

    return a.exec();
}

#include <QMainWindow>

class QMenuBar;
class QMenu;

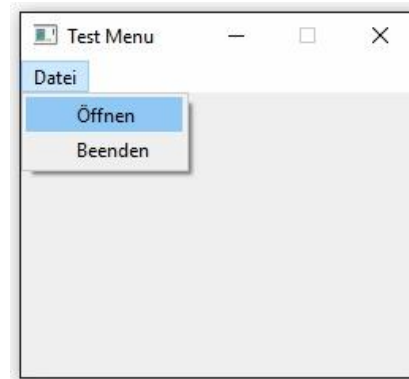
class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow();

private:
    QMenuBar *myMenuBar;
    QMenu *fileMenu;
    QAction *openAction;
    QAction *closeAction;

public slots:
    void openSlot();
};
```

Ausgabe:



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

#include <QMenuBar>
#include <QMenu>
#include <QApplication> //für Systemfunktionen wie quit()

MainWindow::MainWindow()
{
    setWindowTitle("Test Menu");
    setFixedSize(250, 200);

    Widget *myWidget = new Widget();
    setCentralWidget(myWidget); // setzen des Widget im Zentrum

    //leere Menü Leiste erstellen
    myMenuBar = this -> menuBar();

    //Menü „Datei“ aufbauen
    fileMenu = new QMenu("Datei"); //leeres Menü erstellen
    myMenuBar -> addMenu(fileMenu); //Menü an Menü Leiste
                                    anhängen

    //Menüeinträge mit Connector erstellen
    openAction = fileMenu -> addAction("Öffnen");
    QObject::connect(openAction, SIGNAL(triggered()),
                     this, SLOT(openSlot()));

    closeAction = fileMenu -> addAction("Beenden");
    QObject::connect(closeAction, SIGNAL(triggered()),
                     QApplication::instance(), SLOT(quit()));
}
```

```
#include <QObject>
#include <QWidget>
```

```
class Widget : public QWidget
{
public:
    Widget();
};
```

Code

```
#include <QMenuBar>
#include <QMenu> // including QAction
```

```
#include <QApplication>
```

```
#include <QIcon>
```

```
void openSlot();
```

```
myMenuBar = this -> menuBar();
```

```
fileMenu = new QMenu("Datei");
```

```
myMenuBar -> addMenu(fileMenu);
```

```
openAction = fileMenu -> addAction("Öffnen");
closeAction = fileMenu -> addAction("Beenden");
```

```
openAction = fileMenu -> addAction(QIcon(":/icons/open.png"),
    " Öffnen ");
closeAction = fileMenu -> addAction(QIcon(":/icons/exit.png"),
    " Beenden ");
```

```
QObject::connect(openAction,SIGNAL(triggered()),
    this,SLOT(openSlot()));
```

```
QObject::connect(closeAction,SIGNAL(triggered()),
    QApplication::instance(),SLOT(quit()));
```

```
void MainWindow::openSlot()
```

```
{
    //any code
}
```

```
#include "widget.h"
```

```
Widget::Widget()
```

```
{
}
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qmainwindow.html>

Beispiel im Netz: <http://doc.qt.io/qt-5/qtwidgets-mainwindows-menus-example.html>

<http://zetcode.com/gui/qt4/menusandtoolbars/>

<http://doc.qt.io/qt-5/qtwidgets-mainwindows-application-example.html>

<http://doc.qt.io/qt-5/qtwidgets-mainwindows-mainwindow-example.html>

Wenn Grundfunktionen wie quit(), aboutQt(), usw verwendet werden sollen. Siehe auch Link Beispiel.

Wenn für die Aktionen Icons angezeigt werden sollen

Jede Aktion die nicht mit einer Systemfunktion wie quit() verbunden ist, muss jeweils mit einem eigenen Slot verbunden werden. Im Slot kann dann jeweils der Code ausgeführt werden der gewünscht ist, wie zum Beispiel auch das öffnen eines neuen Widgets mit einem QDialog.

Erstellen Menü Leiste

Erstellen neues Menu mit Namen Datei

Menu in die Menü-Leiste eintragen

Button (=QAction) in das Menu eintragen

Buttons (=Actions) der MenuBar zuordnen und Icon geben. Der Text wird in mToolTip angezeigt.

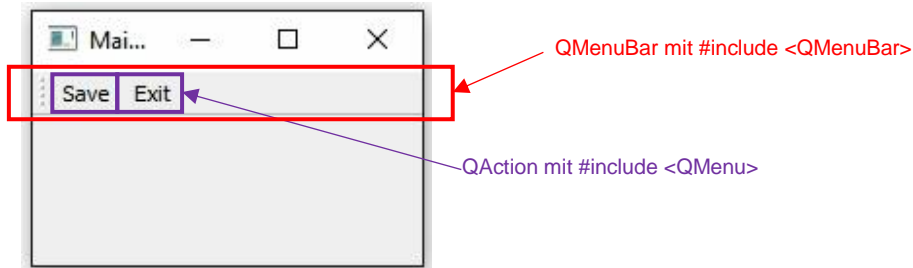
Wichtig: Icon müssen mit Ressourcen Datei dem Projekt zugeführt werden, siehe auch im Kapitel Icon.
Signal-Slot mit System Signal und eigenem Slot

Signal-Slot mit System Signal und System Slot quit() von QApplication.

41. Tool-Bar und Tool-Button

Beim Tool-Button handelt es sich um die Buttons, welche beim Window im Header in einem Tool-Bar angezeigt werden können.

Mit Text:



Code Signal (innerhalb vom selben Source File):

Code :

```
main.cpp

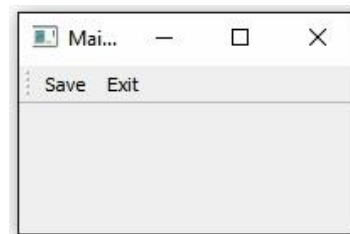
#include "mainwindow.h"
#include "widget.h"
#include <QApplication>

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

    MainWindow w;
    w.show();

    return a.exec();
}
```

Ausgabe:



Header mainwindows.h

```
#include <QMainWindow>
#include <QObject>

class QMenuBar;
class QMenu;
class QToolBar;

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);

private:
    //ToolBar MainWindow
    QToolBar *myToolBar;
    QAction *actionToolSave;
    QAction *actionToolExit;

signals:

public slots:
    void saveSlot();
};
```

Implementierung mainwindow.cpp

```
#include "mainwindow.h"

#include <QApplication>
#include <QMenu>
#include <QToolBar>

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent)
{
    setWindowTitle("MainWindow");
    setMinimumSize(200,100);

    myToolBar = new QToolBar();
    myToolBar = this ->addToolBar("Meine ToolBar");

    actionToolSave = myToolBar -> addAction("Save");
    actionToolExit = myToolBar -> addAction("Exit");

    connect(actionToolSave,SIGNAL(triggered()),
            this,SLOT(saveSlot()));

    connect(actionToolExit,SIGNAL(triggered()),
            QApplication::instance(),SLOT(quit()));
}

void MainWindow::saveSlot()
{
}
```

Code

```
#include <QToolBar>
#include <QIcon>
#include <QMenu> // including QAction

#include <QApplication>

#include <QIcon>
void saveSlot();

myToolBar = new QToolBar();
myToolBar = this ->addToolBar("Meine ToolBar");

actionToolSave = myToolBar -> addAction("Save");
actionToolExit = myToolBar -> addAction("Exit");

actionToolSave = myToolBar -> addAction(QIcon(":/icons/save.png"),
                                         "Save");
actionToolExit = myToolBar -> addAction(QIcon(":/icons/exit.png"),
                                         "Exit");

connect(actionToolSave,SIGNAL(triggered()),
        this,SLOT(saveSlot()));

connect(actionToolExit,SIGNAL(triggered()),
        QApplication::instance(),SLOT(quit()));
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qtoolbar.html>

Beispiel im Netz:

<http://zetcode.com/gui/qt4/menusandtoolbars/>

Wenn Grundfunktionen wie quit(),aboutQt(), usw verwendet werden sollen. Siehe auch Link Beispiel.

Wenn für die Aktionen Icons angezeigt werden sollen
Jede Aktion die nicht mit einer Systemfunktion wie quit() verbunden ist, muss jeweils mit einem eigenen Slot verbunden werden. Im Slot kann dann jeweils der Code ausgeführt werden der gewünscht ist, wie zum Beispiel auch das öffnen eines neuen Widgets mit einem QDialog.

Erstellen neues ToolBar Objekt

ToolBar dem MainWindow zufügen

ToolButtons (=Actions) der ToolBar zuordnen und Text geben.

ToolButtons (=Actions) der ToolBar zuordnen und Icon geben.
Der Text wird in mToolTip angezeigt.

Wichtig: Icon müssen mit Ressourcen Datei dem Projekt zugeführt werden, siehe auch im Kapitel Icon.
Signal-Slot mit System Signal und eigenem Slot

Signal-Slot mit System Signal und System Slot quit() von QApplication.

42. MouseEvent

Mit dem Mouse Event kann die aktuelle Position des Cursor im aufgerufenen Widget / Window ausgegeben werden.

Wichtig : Standart ist, dass es egal ist welche Maustaste gedrückt wird! Wenn dies nicht der Fall sein soll, so muss man die Maustaste mit `QMouseEvent::button()` definieren.

Code:

main.cpp

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

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

    Widget w;
    w.show();

    return a.exec();
}

Header widget.h

#include <QObject>
#include <QWidget>

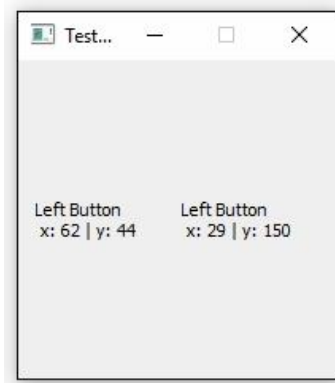
class QLabel;
class QHBoxLayout;

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget();
    void mouseDoubleClickEvent(QMouseEvent *event);

private:
    QLabel *leftLabel;
    QLabel *rightLabel;
    QHBoxLayout *myLayout;
    int x1 = 0, y1 = 0;
};
```

Ausgabe:



Implementierung widget.cpp

```
#include "widget.h"
#include <QLabel>
#include <QMouseEvent>
#include <QHBoxLayout>
#include <QDebug>

Widget::Widget()
{
    setMouseTracking(true);

    setWindowTitle("Test MouseEvent");
    setFixedSize(200, 200);

    leftLabel = new QLabel("Initialisierung\n\rlinks");
    rightLabel = new QLabel("Initialisierung\n\rrechts");
    myLayout = new QHBoxLayout();

    myLayout -> addWidget(leftLabel);
    myLayout -> addWidget(rightLabel);
    this -> setLayout(myLayout);
    this -> show();
}

void Widget::mouseDoubleClickEvent(QMouseEvent *event)
{
    x1 = event -> x();
    y1 = event -> y();
    qDebug() << x1 << " " << y1;

    QString strX1 = QString::number(x1);
    QString strY1 = QString::number(y1);
    QString strLabel = "Left Button\n\rx: " + strX1 + " | y: " + strY1;

    if(event->buttons() & Qt::LeftButton)
    {
        leftLabel -> setText(strLabel);
    }
    if(event->buttons() & Qt::RightButton)
    {
        rightLabel -> setText(strLabel);
    }
    // emit anySignal();
}
```

Code

```
#include <QMouseEvent>
```

```
mouseGrabber()
```

```
emit anySignal();
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qmouseevent.html>

bringt das aktuelle gültige Widget / Window wieder in den Vordergrund

mit dem Schlüsselwort „emit“ kann man im QMouseEvent jede beliebige erreichbare Funktion aufrufen!

Wenn die zum Beispiel mit dem painterEvent verbunden wird, so kann mit der Maus gezeichnet werden.

Events:

```
void mousePressEvent(QMouseEvent *event);
```

Sendet Event wenn Taste gedrückt wird

```
void mouseReleaseEvent(QMouseEvent *event);
```

Sendet Event wenn Taste losgelassen wird

```
void mouseMoveEvent(QMouseEvent *event)
```

Sendet Event wenn Cursor sich innerhalb eines Widget / Window bewegt oder dieses verlässt.

```
void mouseDoubleClickEvent (QMouseEvent *event)
```

Sendet Event wenn Taste doppelgedrückt wird

QMouseEvent::button()

Link im Netz: <http://doc.qt.io/qt-5/qt.html#MouseButton-enum>

43. PaintEvent und Painter

Wenn in Qt gezeichnet werden soll, so werden der Painter verwendet für das Zeichnen an sich und das paintEvent um über Signal die Slots auszulösen, zum Beispiel auf Knopfdruck etwas zeichnen.

Code :

main.cpp

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

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

    Widget w;
    w.show();

    return a.exec();
}
```

Header widget.h

```
#include <QObject>
#include <QWidget>

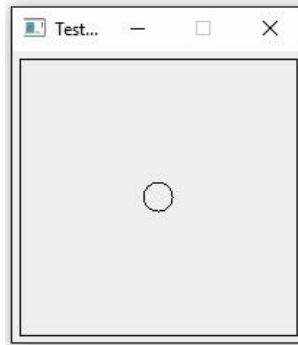
class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget();
    void paintEvent(QPaintEvent *event);
    void anyFunction();

private:
    int edgeGap = 5;
    int radiusArc = 20;
    int radiantStart = 0;
    int radiantStop = 5760;
};
```

emit anySignal();

Ausgabe:



Implementation widget.cpp

```
#include "widget.h"

#include <QPainter>
// #include <QPaintEvent> // in diesem Fall nicht notwendig!
#include <QDebug>

Widget::Widget()
{
    setWindowTitle("Test PainterEvent");
    setFixedSize(200, 200);

    this -> show();
}

void Widget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);

    painter.drawRect(0 + edgeGap, 0 + edgeGap,
        this -> width() - (2 * edgeGap),
        this -> height() - (2 * edgeGap));

    int widgetCenterX = width() - ( width()/2 )
        + (radiusArc/2 );
    int widgetCenterY = height() - ( height()/2 )
        + (radiusArc/2 );

    painter.drawArc(widgetCenterX, widgetCenterY,
        radiusArc, radiusArc,
        radiantStart, radiantStop);

    // emit anySignal();
}
```

mit dem Schlüsselwort „emit“ kann man aus dem paintEvent jede beliebige erreichbare Funktion aufrufen!

Wichtig : Auf **this ->** vor jedem width und height kann verzichtet werden, weil es sich auf das aktuelle Widget bezieht. Wenn jedoch die Breite und Höhe eines anderen Objektes benutzt werden soll, so muss anstelle des **this** jeweils der Namen des Widget oder Window Objekt stehen.

Code

```
#include <QPainter>
#include <QPaintEvent>
```

```
drawRect(xStartPos, yStartPos, width, height);
```

```
drawArc(xStartPos, yStartPos,
        xRadius, yRadius,
        radiantStart, radiantStop);
```

```
drawLine(), drawPoint(), drawText(), drawPixmap(),
....usw
```

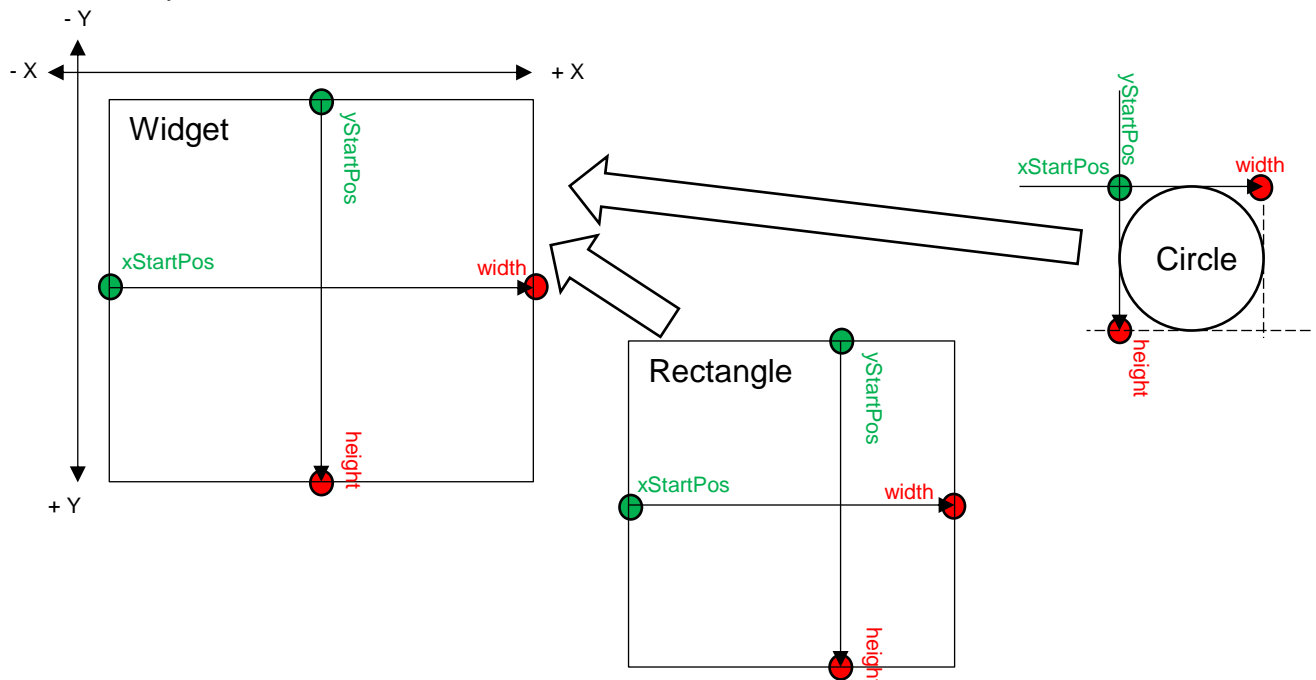
Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qpainter.html>
<http://doc.qt.io/qt-5/qpaintevent.html>

Es gibt soviel verschiedene Arten etwas zu zeichnen das man am besten die Klassen mit dem Link oben durchsucht. Zum Beispiel: Rechteck mit Ecken, Rechteck mit runden Ecken, ... usw.

Koordinatensystem beim Zeichnen:



Ein gutes Beispiel ist die analoge Uhr die in Qt projiziert werden kann. Eine genaue Beschreibung und das Projekt findet man hier: <http://doc.qt.io/qt-5/qtwidgets-widgets-analogclock-example.html>

44. Zeichnen mit MouseEvent

Folgendes Beispiel soll zeigen wie man mit der Maus zeichnet indem man das MouseEvent und das PaintEvent verwendet.

Beim drücken der Maustaste links werden die Start Koordinaten x & y gesetzt, beim loslassen der Maustaste werden die Stopp Koordinaten x & y gesetzt. Ausserdem wird beim loslassen mit emit das Signal drawRect() aufgerufen wodurch der System Slot „repaint“ aufgerufen wird und das paintEvent ausgelöst wird.

Code :

main.cpp

```
#include <QApplication>

#include "widget.h"

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

    Widget *myWidget = new Widget();
    myWidget->show();

    return app.exec();
}
```

header.cpp

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>

class QPaintEvent;
class QMouseEvent;

class Widget : public QWidget
{
    Q_OBJECT

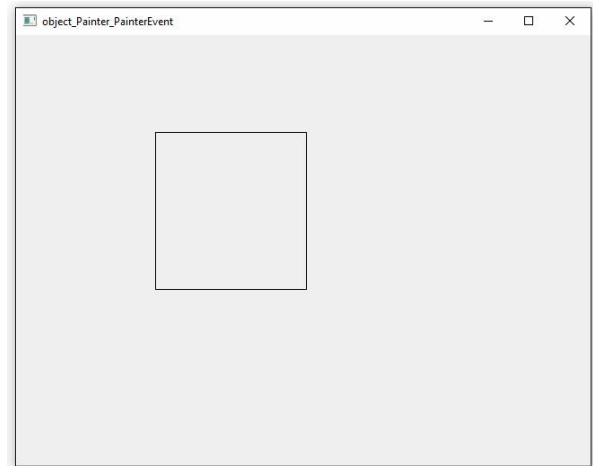
private:
    int xStart, yStart, xStop, yStop;

public:
    Widget();
    void paintEvent(QPaintEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);

signals:
    void drawRect(); // Keine Implementierung notwendig!
};

#endif
```

Ausgabe:



widget.cpp

```
#include "widget.h"

#include <QPainter>
#include <QPaintEvent>
#include <QPoint>
#include <QDebug>

#include <iostream>
#include <cstdlib>
using namespace std;

Widget::Widget() {
    setMouseTracking(true);

    QObject::connect(this, SIGNAL(drawRect()), this, SLOT(repaint()));
}

void Widget::paintEvent(QPaintEvent *event) {
    QPainter p(this);
    p.drawRect(xStart, yStart, (xStop - xStart), (yStop - yStart));
}

void Widget::mouseMoveEvent(QMouseEvent *event) {
    qDebug() << event -> x() << ", " << event -> y();
}

void Widget::mousePressEvent(QMouseEvent *event) {
    qDebug() << "PRESS:" << event -> x() << ", " << event -> y();
    xStart = event -> x();
    yStart = event -> y();
}

void Widget::mouseReleaseEvent(QMouseEvent *event) {
    qDebug() << "RELEASE:" << event -> x() << ", " << event -> y();
    xStop = event -> x();
    yStop = event -> y();
    emit drawRect(); // mit emit kann Signal ausgelöst werden!
}
```

Code

```
#include <QPainter>  
#include <QPaintEvent>
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qpainter.html>
<http://doc.qt.io/qt-5/qpaintevent.html>

Signal:

update()

löst nicht sofort aus wie <repaint> sondern plant das paintEvent so, dass es erst ausgelöst wird wenn Qt in den main loop zurückkehrt.

repaint()

<repaint> löst sofort aus was unter Umständen zu Rekursion führen kann. <repaint> sollte nur verwendet werden wenn absolut notwendig ist, z.Bsp in Verbindung mit einem Thread oder wenn. Im Normalfall reicht <update> aus.

45. Dialog

Als Dialog ist eine „Konversation“ zwischen Benutzer und Software gemeint! Dialoge haben im Gegensatz zu Widgets und MainWindow zusätzliche Kanäle (OK-, Ja, Nein PushButton, usw.) und sind immer vom Element abgeleitet welches den Dialog erstellt.

MessageBoxes sind auch Dialoge. Mit Dialogen kann ein Widget von einem anderem Widget zum Beispiel über einen PushButton geöffnet / erstellt werden.

In diesem Beispiel wird der aufgezeigt wie man vorgehen muss, damit es funktioniert. Es wird ein MainWindow gebaut in welches das Widget eingefügt ist. Von diesem kann man dann über ienen Button das nächste Widget „Dialog“ öffnen.

- Wichtig:
1. Die Klassendeklaration (rot) mit den Parents muss stimmen
 2. Die Bibliotheken (grün) müssen an der richtigen Stelle eingefügt werden

Code :

main.cpp

```
#include "mainwindow.h"

#include <QApplication>

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

    MainWindow w;
    w.show();

    return a.exec();
}
```

Header mainwindow.h

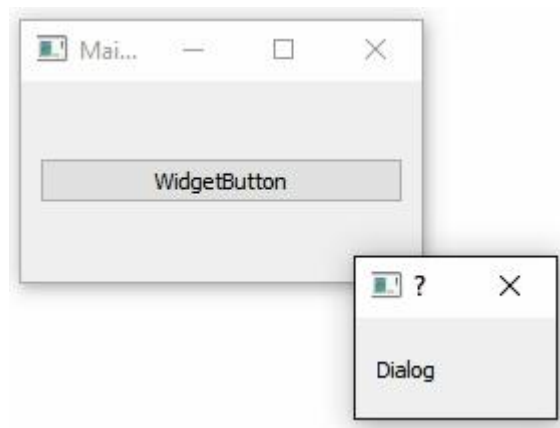
```
#include "widget.h"

#include <QMainWindow>
#include <QObject>

class QMenuBar;
class QMenu;
class QToolBar;

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = 0);
};
```

Ausgabe:



Implementierung mainwindow.cpp

```
#include "mainwindow.h"

#include <QWidget>

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent)
{
    Widget *myWidget = new Widget();
    setCentralWidget(myWidget);

    setWindowTitle("MainWindow");
}
```

Header widget.h

```
#include "dialog.h"

#include <QWidget>
#include <QObject>

class QGridLayout;
class QLineEdit;
class QPushButton;

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);

private:
    QGridLayout *myGridLayout;
    QPushButton *myPushButton;

public slots:
    void slotButton(bool check);
};
```

Header dialog.h

```
#include <QDialog>

class QLabel;
class QVBoxLayout;

class Dialog : public QDialog
{
    Q_OBJECT
public:
    Dialog(QWidget *parent = 0);

private:
    QVBoxLayout *myLayout;
    QLabel *myLabel;
};
```

Implementierung widget.cpp

```
#include "widget.h"

#include <QDialog>
#include <QPushButton>
#include <QGridLayout>

Widget::Widget(QWidget *parent) : QWidget(parent)
{
    myGridLayout = new QGridLayout();
    myPushButton = new QPushButton("WidgetButton");
    myGridLayout->addWidget(myPushButton, 0, 0);

    setLayout(myGridLayout);

    connect(myPushButton, SIGNAL(clicked(bool)),
            this, SLOT(slotButton(bool)));
}

void Widget::slotButton(bool check)
{
    Dialog *dialog = new Dialog();
    dialog->setFixedSize(100, 50);
}
```

Implementierung dialog.cpp

```
#include "dialog.h"

#include <QLabel>
#include <QVBoxLayout>

Dialog::Dialog(QWidget*parent) : QDialog(parent)
{
    myLayout = new QVBoxLayout();
    myLabel = new QLabel("Dialog");
    myLayout->addWidget(myLabel);
    setLayout(myLayout);
    show();
}
```

46. ChildWidget (Vererbung)

Code :

Ausgabe:

Code

#include Q

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz:

Signal:

Slot:

47. MySQL

Code :

Ausgabe:

Code

```
#include Q
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz:

Signal:

Slot:

48. TcpServer

Code :

Ausgabe:

Code

#include Q

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz:

Signal:

Slot:

49. TcpSocket

Bevor TCPSocket Anwendungen in Qt erstellt werden können muss die folgende Qt Bibliothek in der qmake Datei myProjectName.pro eingefügt werden:

QT += network

Siehe auch Kapitel „Qmake Projekt Datei“ in diesem Dokument.

Code :

Ausgabe:

Code

#include Q

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz:

Signal:

Slot:

50. Lesen Dateien

Code :

Ausgabe:

Code

```
#include Q
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz:

Signal:

Slot:

51. Schreiben Dateien

Code :

Ausgabe:

Code

#include Q

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz:

Signal:

Slot:

52. Mutex

Code :

Ausgabe:

Code

#include Q

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz:

Signal:

Slot:

53. Thread

In der Informatik bezeichnet Thread („Faden“, „Strang“) auch Aktivitätsträger oder leichtgewichtiger Prozess genannt. Ein Thread ist ein Ausführungsstrang oder eine Ausführungsreihenfolge in der Abarbeitung eines Programms. Ein Thread ist immer Teil eines Prozesses.

Man unterscheidet zwei Arten von Threads:

1. Threads im engeren Sinne, die sogenannten Kernel-Threads, laufen ab unter Steuerung durch das Betriebssystem.
2. Im Gegensatz dazu stehen die sogenannten User-Threads, die das Computerprogramm des Anwenders komplett selbst verwalten muss.

Als User-Thread, gelegentlich auch Userlevel-Thread und z. B. unter Windows Fiber genannt, bezeichnet man in der Informatik eine bestimmte Art, Programme bzw. Programmteile verzahnt ablaufen zu lassen. Die Funktionalität ist dabei nicht direkt im Kernel implementiert sondern in einer separaten Programmbibliothek, die im Userspace liegt. Dadurch ist ein Kontextwechsel (Taskswitching) zwischen den Userthreads ohne aufwendige Systemaufrufe möglich.

Anwendung:

Header myTest.h

```
#include <amyLibarys>

class QPaintEvent ;
class QMouseEvent;
class Thread;

class myTest : public QWidget
{
    private:
        int x, y;
        Thread *thread;

    public:
        Canvas();
        void paintEvent(QPaintEvent *event);
        void mousePressEvent(QMouseEvent
                             *event);
};
```

Implementierung myTest.cpp

```
#include "myTest.h"
#include "thread.h"

#include <QPaintEvent>
#include <QPainter>
#include <QMouseEvent>
#include <QDebug>

myTest::myTest()
{
    x = width()/2; //Regel für Thread, Var die ändert
    y = height()/2; //Regel für Thread, Var die ändert

    thread = new Thread(x, y);

    QObject::connect(thread, SIGNAL(updated()),
                     this, SLOT(update()));
}

void Canvas::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);

    x = thread->getCurrentX();
    y = thread->getCurrentY();
}

void Canvas::mousePressEvent(QMouseEvent
                             *event)
{
    qDebug() << event->x() << " " << event->y();
    thread->setFutureX(event->x());
    thread->setFutureY(event->y());
}
```

Header myThread.h

```
#include <QThread>

class Thread : public QThread
{
    Q_OBJECT

private:
    int futureX, futureY, currentX, currentY;

public:
    Thread(int x, int y);
    void run();
    int getFutureX() const;
    void setFutureX(int value);

    int getFutureY() const;
    void setFutureY(int value);

    int getCurrentX() const;
    void setCurrentX(int value);

    int getCurrentY() const;
    void setCurrentY(int value);

signals:
    void updated();
};
```

Implementierung myThread.cpp

```
#include "thread.h"

Thread::Thread(int x, int y) :
    currentX(x), futureX(x), currentY(y), futureY(y)
{
}

void Thread::run()
{
    while (true)
    {
        int dx, dy;
        emit updated();
        msleep(100);
    }
}

int Thread::getFutureX() const
{
    return futureX;
}

void Thread::setFutureX(int value)
{
    futureX = value;
}

int Thread::getFutureY() const
{
    return futureY;
}

void Thread::setFutureY(int value)
{
    futureY = value;
}

int Thread::getCurrentX() const
{
    return currentX;
}

void Thread::setCurrentX(int value)
{
    currentX = value;
}

int Thread::getCurrentY() const
{
    return currentY;
}

void Thread::setCurrentY(int value)
{
    currentY = value;
}
```

Code:

Ausgabe:

Code

```
#include <QThread>
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qthread.html>

Signal:

Slot:

54. KeyboardHandler

Code :

Ausgabe:

Code

```
#include <KeyboardHandler>
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qml-qt3d-input-keyboardhandler.html>

Signal:

Slot:

55. KeyEvent

Code :

Ausgabe:

Code

```
#include <QKeyEvent>
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qkeyevent.html>

Signal:

Slot:

56. Font

Code :

Ausgabe:

Code

```
#include <QFont>
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qfont.html>

Signal:

Slot:

57. Translator

Code :

Ausgabe:

Code

```
#include <QTranslator>
```

Erklärung

Bibliothek in Projekt aufnehmen

Link im Netz: <http://doc.qt.io/qt-5/qtranslator.html>

Beispiel: <http://doc.qt.io/qt-5/qtlinguist-hellotr-example.html>

Signal:

Slot:

58. Conan Package Manager installieren

Conan gibt beim Programmieren die Möglichkeit fremde Bibliotheken (librarys) einzubinden, ohne alle Abhängigkeiten (Dependencies) dieser gewünschten Bibliothek von Hand einzeln builden zu müssen.

Beispiel: libssh2 library (<https://www.libssh2.org>)

Um eine SSH Verbindung zu erstellen, braucht es zusätzlich die folgenden Bibliotheken:

- OpenSSL für die Verschlüsselung
- zlib für Datenkomprimierung

Ohne Conan müsste man zum Beispiel für cmake als Build-Management-Tool ein Skript schreiben, welches zuerst OpenSSL und zlib buildet und dann mit diesen zwei zusammen noch die libssh2 buildet.

WICHTIG: Der Rechner „muss“ x64 sein, Conan unterstützt mit der aktuellsten Version x86 nicht mehr!

Installation

Da die Installation von conan.io zusammen mit Qt abhängig vom Betriebssystem (OS) ist und deshalb unterschiedlich aussieht, wird folgend für die drei wichtigsten OS Windows, Linux und Mac die Installation beschrieben.

Download Anleitung aus dem folgenden Git Repository:

<https://github.com/aufdiewelle/set-up-conan.io-for-libssh2-with-Qt.git>

mit dem folgenden git Befehl im Commando Line Interpreter (CLI):

git clone <https://github.com/aufdiewelle/set-up-conan.io-for-libssh2-with-Qt.git>

```
C:\Users\misimi1\Desktop>git clone https://github.com/aufdiewelle/set-up-conan.io-for-libssh2-with-Qt.git
Cloning into 'set-up-conan.io-for-libssh2-with-Qt'...
remote: Counting objects: 9, done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 9 (delta 1), reused 9 (delta 1), pack-reused 0
Unpacking objects: 100% (9/9), done.

C:\Users\misimi1\Desktop\Test>
```

WICHTIG: Der letzte Schritt in der Anleitung „- Download lib source & build dependencies with conan“ muss ausgelassen werden. Um dies zu machen muss zuerst „Bibliothek in das Qt-Projekt einbinden“ ausgeführt werden.

Schritte um Bibliothek in Qt-Projekt einbinden

1. Mit der Datei „conanfile.txt“ wird definiert, welches conan Dependencies Pakete geladen werden soll, mit was es gebuildet werden soll usw., sprich wie conan konfiguriert wird. Unter dem Link <https://www.conan.io/search> kann nach den gewünschten Paketen gesucht werden.

Beispiel Datei „conanfile.txt“

```
[requires]
libssh2/1.8.0@Wi3ard/stable
OpenSSL/1.0.2k@lasote/stable

[generators]
```

qmake

[options]

libssh2:shared=True

OpenSSL:shared=False

zlib:shared=False

[imports]

include, libssh2.h -> ./include

bin, *.dll -> ./lib

OS	Dynamische Bibliotheken	Statische Bibliotheken
Windows	.dll	.lib
Linux	.so (.so.1)	.a
Mac	.dylib or .so	.a

Erklärung der Datei „conanfile.txt“:

[requires] Welche Dependencies Pakete sollen gebuildet werden.

Unter <https://www.conan.io/search> kann nach den Paketen gesucht werden.

WICHTIG: Wie oben beschrieben ist zum Beispiel für die libssh2 auch die OpenSSL für die Verschlüsselung notwendig. Wenn jetzt zum Beispiel eine neue Version der OpenSSL mit einem Bug fix kommt so macht es Sinn die aktuellste Version zu verwenden. Um dies zu erreichen kann eine aktuellere Dependencie verwendet werden um eine veraltete in einem Projekt zu überschreiben. Dazu muss man lediglich die neue unter dem Dependencie welche die alte enthält, wie folgt eingesetzt werden:

[requires]

libssh2/1.8.0@Wi3ard/stable

OpenSSL/1.0.2k@lasote/stable

Erklärung:

„libssh2/1.8.0@Wi3ard/stable“ enthält OpenSSI Version 1.0.2i aber mit“ OpenSSL/1.0.2k@lasote/stable“ wird die aktuellste OpenSSL Version 1.0.2k verwendet.

[generators] Definieren welches Build-Management-Tool verwendet werden soll. Am häufigsten sind:

- cmake
- qmake

Weitere können und dem folgenden Link gefunden werden:

https://en.wikipedia.org/wiki/List_of_build_automation_software

[options] Definieren ob statische oder dynamische Bibliotheken beim Build erstellt werden, heisst es wird eine Bibliothek Datei entsprechend dem System wie in diesem Dokument im Kapitel „Linken mit Bibliotheken“ erstellt.

OS	Dynamische Bilbiotheken	Statische Bilbiotheken
Windows	.dll	.lib
Linux	.so (.so.1)	.a
Mac	.dylib or .so	.a

Wenn nichts angegeben wird ist default = false und bedeutet „statisch“
linken. Dynamisches linken wird mit der folgenden Syntax gemacht:
<libraryName>:shared=True

Beispiel 1: Bauen der dynamischen libssh2 als .dll, .so oder .dylib je nach OS. Die libssh2 braucht für die OpenSSL-Bibliothek für die Verschlüsselung und die zlib-Bibliothek für Datenkomprimierung. Da nur die libssh2 dyn. sein muss kann OpenSSL und zlib statisch gebaut werden. Beim linken mit dem Compiler wird dann die OpenSSL und die zlib in die libssh2 integriert (siehe auch Kapitel „Linken mit Bibliotheken“).

```
[requires]
libssh2/1.8.0@Wi3ard/stable
OpenSSL/1.0.2k@lasote/stable
```

```
[options]
libssh2:shared=True
```

Es werden bei Windows die Bibliothek Datei wie folgt erstellt:

```
libssh2:    libssh2.dll
OpenSSL:    libeay32MD.lib
            ssleay32MD.lib
zlib:       zlibstatic.lib
```

..oder unter Linux:

```
libssh2:    libssh2.so
            libssh2.so.1
OpenSSL:    libcrypto.a
            libssl.a
zlib:       libz.a
```

Wie ersichtlich muss die Bibliothek Datei nicht denselben Namen haben wie das Packet und es kann auch sein, dass mehrere Bibliothek Dateien pro Packet erstellt werden.

Die Dateien werden im lokalen cache „~/conan/data“ von conan generiert. Unter Windows ist das der Pfad „C:\Users\<myLogin>\conan\data“. Einfach in diesem pfad Abhängig vom OS nach *.dll, *.so, *.so.1 oder *.dylib suchen.

Info: Im Fall von libssh2/1.8.0@Wi3ard/stable ist anstelle der libssh2.dll Datei eine unsichtbare

Datei „conan_link“ im Verzeichnis gespeichert, welche den Pfad des /bin Ordner enthält in dem die libssh2.dll gespeichert ist.

Beispiel 2: Bauen der dynamischen libssh, OpenSSL und zlib als .dll, .so oder .dylib je nach OS. Die libssh2 braucht für die OpenSSL-Bibliothek für die Verschlüsselung und die zlib-Bibliothek für Datenkomprimierung. Diese können also gleichzeitig auch dynamisch gebaut werden (siehe auch Kapitel „Linken mit Bibliotheken“).

```
[requires]
libssh2/1.8.0@Wi3ard/stable
OpenSSL/1.0.2k@lasote/stable
```

```
[options]
libssh2:shared=True
OpenSSL:shared=True
zlib:shared=True
```

Es werden bei Windows die Bibliothek Datei wie folgt erstellt:

```
libssh2:      libssh2.dll
OpenSSL:      libeay32MD.dll
              ssleay32MD.dll
zlib:         zlib.dll
```

..oder unter Linux:

```
libssh2:      libssh2.so
              libssh2.so.1
OpenSSL:      libssl.so
              libcrypto.so
zlib:         libz.so
              libz.so.1
Electric-fence libefence.so
```

WICHTIG : Beim deployen von App mit dyn. Bibliotheken muss normalerweise das OS selbst wissen wo die Bibliotheken sind. Der Dateityp und der Ort ist abhängig vom Betriebssystem und deshalb wird folgend für die wichtigsten OS Windows, Linux und Mac kurz aufgezeigt:

OS	Dateityp	Pfad
Windows	.dll	mit der ausführbaren Datei (.exe) zusammen
Linux	.so (.so.1)	System Bibliotheken Pfad, meisten: /usr/lib siehe auch unter folgenden Link: http://doc.qt.io/qt-5/linux-deployment.html
Mac OSX	.dylib	mit der ausführbaren Datei zusammen, siehe auch Link unten!

Mit Mac OSx den „Tip“ am Ende der Seite vom folgenden Link anschauen:
http://docs.conan.io/en/latest/manage_deps/conanfile_txt.html#

[imports] Hier können Regeln angegeben werden, um Dateien automatisch an den gewünschten Ort zu kopieren. Somit können beim dynamischen linken mit [option] zum Beispiel die Bibliothek Dateien automatisch dem Projekt hinzugefügt werden.

Dabei gilt es folgende Konvention für Windows *.dll Dateien zu beachten: *.dll sind im „./bin“ Ordner vom conan cache (). Dies ist nur eine Konvention, heisst der Ersteller des Dependencies muss sich nicht daran halten.

```
bin, *.dll -> ./bin      #for Windows (bin is convention for Windows)
lib, *.so -> ./lib       #for Linux 1
lib, *.so.1 -> ./lib     #for Linux 2
lib, *.dylib -> ./lib    #for MacOS
```

Für mehr Informationen und Details den folgenden Link verwenden:

http://docs.conan.io/en/latest/manage_deps/conanfile_txt.html#

2. Mit conan werden die Bibliotheken manuell oder automatisch erstellt. Die Bibliothek-Dateien die conan generiert werden je nach OS in das folgende Verzeichnis abgelegt:

Windows	C:\Users\[Benutzer]\.conan C:\.conan
Linux & Mac OSx	/home/[Benutzer]/.conan/

Manuell: Dazu muss der CLI (Command Line Interpreter) geöffnet werden und in das Projektverzeichnis gewechselt werden in dem die Datei „conaninfo.txt“ liegt. Dort gibt man in den CLI den folgenden Befehl ein:

CLI~\$: install conan .

Info: Dieser Befehl funktioniert bei Linux - & Mac OS nicht und führt zu einem Fehler!

...wenn es nicht funktioniert kann man es mit dem folgenden Befehl versuchen und conan dazu zwingen vom binary zu builden. Dazu müssen aber zuerst alle conan Daten im Verzeichnis...

@Windows	C:\Users\[Benutzer]\.conan\data C:\.conan
@Linux & Mac OSx	/home/[Benutzer]/.conan/data/

...zuerst wieder gelöscht werden bevor man den folgenden Befehl eingibt::

CLI~\$: conan install --build libssh2

Info: Dieser Befehl funktioniert bei Windows nicht und führt zu einem Fehler!

Automatisch: Um mit Qt automatisch die Bibliotheken zu builden und zu integrieren muss diese vor dem build-Prozess bei Qt über die Projektdatei *.pro gebuildet werden.

Dazu müssen die Befehle für das jeweilige OS in der Projektdatei *.pro integriert werden.

```
win32: system($$cmd.exe conan install .)
unix: system($$xterm conan install --build libssh2)
```

Damit wird auf dem CLI vom OS der jeweilige conan Befehl vor dem Projekt-Build ausgeführt ABER nur wenn noch kein build gemacht wurde. Aus diesem Grund müssen die folgenden zwei Ordner zuerst gelöscht werden falls diese vorhanden sind:

```
build-<myProjectName>_Qt_5_n_n_MinGW_32bit-Debug
build-<myProjectName>_Qt_5_n_n_MinGW_32bit-Release
```

3. Bibliothek dem Qt Projekt hinzufügen

Hinzufügen einer externen Bibliothek zu einem Qt Projekt

Indirekt über conan: Es muss dem Qt Projekt mitgeteilt werden, dass conan.io verwendet wird und die Konfigurationsdatei von conan muss eingebunden werden. Dies wird über die „meinProjektName.pro“ Datei welche im Toplevel Ordner vom Projekt gespeichert ist gemacht. Um dies zu erreichen müssen nur folgende zwei Zeilen in die „meinProjektName.pro“ Datei eingefügt werden:

```
CONFIG += conan_basic_setup
include(conanbuildinfo.pri)
```

Siehe auch Link:

<http://conanio.readthedocs.io/en/latest/integrations/qmake.html#qmake>

Funktioniert nicht bei @ Windows?

Direkt über Qt: Um eine Bibliothek direkt zu einem Projekt hinzufügen muss wie folgt die Code-Zeilen in der Qt Projektdatei *.pro wie folgt hinzugefügt werden:

Statisch:

TODO: muss noch getestet werden!

Dynamisch:

```
INCLUDEPATH += $$PWD/
DEPENDPATH += $$PWD/
win32: LIBS += -L$$PWD"bin" -llibssh2
unix: LIBS += -L$$PWD/lib/ -lssh2
```


59. App deployment Windows mit InnoSetup

????????????????

60. App deployment bei Qt

????????????????