1 „Hello World“

Thema: Erstellen eines ersten Programms mit Fenstertechnik

1. Projekt erstellen
   a) Qt Creator öffnen
   b) Alle alten Projekte im Qt Creator schließen
   c) Leeres Qt Projekt erstellen (Hinweis: Qt Creator erstellt ein Unterverzeichnis mit dem Projektnamen).

2. „main.cpp“ - Datei anlegen und dem Projekt hinzufügen
   a) Projekt in der Projektübersicht auswählen
   b) Neue „C++ Quelldatei“ anlegen
   c) Name „main.cpp“ eingeben
   d) Pfadangabe kontrollieren
   e) Zum Projekt hinzufügen

3. Programm schreiben und starten
   a) Quellcode in „main.cpp“ eingeben:

```
1  #include <QApplication>
2  #include <QLabel>
3
4  int main(int argc, char **argv)
5  {
6      QApplication a(argc, argv);
7      QLabel l("Hello World");
8      l.show();
9      return a.exec();
10 }
```

   b) Anwendung starten entweder über den grünen Startpfeil auf der linken Seite oder Tastaturkombination: CTRL+R

Beachten Sie, dass sich das Anwendungsfenster bereits in Größe und Form verändern lässt. Der im Fenster enthaltene Text ist immer links ausgerichtet und vertikal zentriert.

Aufgabe: Ausrichten von Text im QLabel

Der Text im Fenster soll in alle Richtungen zentriert ausgerichtet werden. Um herauszufinden, wie dies zu realisieren ist, kann das Hilfesystem zu **QLabel** genutzt werden.

1. Setzen Sie den Cursor auf das Wort **QLabel** im Quelltext.
2. Drücken Sie die Taste F1. Dadurch gelangen Sie zur Hilfe zu **QLabel**. Alternativ können Sie auch über das Menü auf der linken Seite zur Hilfe gelangen und nach **QLabel** im Index suchen.

Alle Qt Klassen sind nach einer spezifischen Struktur dokumentiert.

- Minimalerklärung zur Klasse mit einem Link (*more...*) zu einer detailierteren Beschreibung
- Einzubindende Bibliothek (\#include)
- Vaterklasse (Stichwort: Vererbung)
- Attribute (engl. properties) der Klasse
- Methode der Klasse
  - Öffentliche Methoden (**public**)
  - Öffentliche Slots (**public slots**)
  - Signale (**signals**)
  - Statische öffentliche Methoden (**static public**)
  - ...
  - Detailierte Beschreibung (oft mit Beispielen)

Wenn Sie nun die detailierte Beschreibung von **QLabel** lesen, erkennen Sie, dass der Text im Label per Default immer linksbündig und vertikal zentriert ausgerichtet wird. Die Position kann aber mittels setAlignment() verändert werden. Schauen Sie in der Beschreibung der setAlignment() Methode und der Konstanten Qt-Variable Qt ::Alignment nach, wie die Methode anzuwenden ist, um den Text im **QLabel** in alle Richtungen zu zentrieren. Erweitern Sie Ihren Quellcode entsprechend.

Aufgabe: QPushButton hinzufügen

Nun soll noch ein Button zum Schließen der Anwendung hinzugefügt werden.

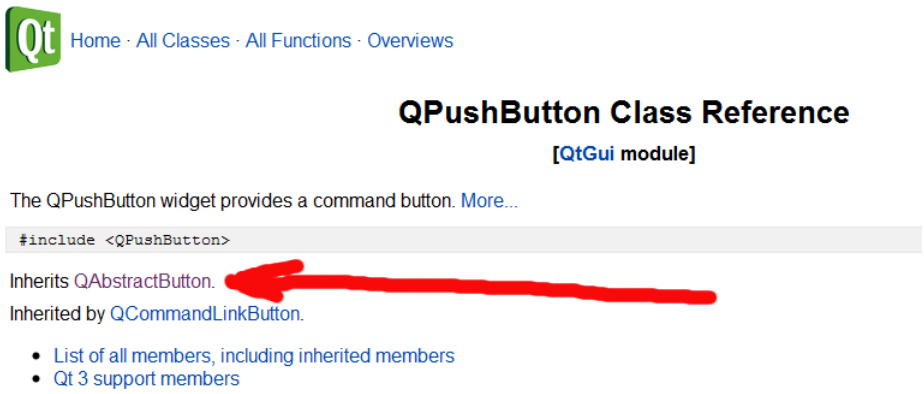
1. Fügen Sie folgenden Quellcode vor dem Aufruf der Methode show des **QLabel** ein:

```
1 QPushButton b("Schließen");
2 b.show();
```

2. Binden Sie die für den **QPushButton** nötige Header-Datei im main ein.

3. Schauen Sie in die Hilfe zum **QPushButton**, um herauszufinden, wie der Button mit Funktionalität gefüllt werden kann.

   a) Rufen Sie zunächst die Hilfe zum **QPushButton** auf. Da dort keinerlei Signale dokumentiert sind, müssen Sie in der Dokumentation der Vater-Klasse von **QPushButton** wechseln. Der Programmierer sagt: „Die Signale der Vater-Klasse werden an die Kind-Klasse vererbt“. Die Vater-Klasse, QAbstractButton, ist ganz oben angegeben:



   Über den Link können Sie zur Dokumentation von QAbstractButton wechseln.

   b) Im Signale-Bereich der Dokumentation zu QAbstractButton finden Sie die Methode clicked.

   c) Das clicked Signal (Quelle bzw. Source) muss nun noch mit der **QApplication** (Ziel bzw. Destination) „verbunden“ werden. Die wird mit dem Makro QObject::**connect**(src, signal, dest, slot) erreicht. Füge den folgenden Befehl direkt vor dem **return** ... ein:

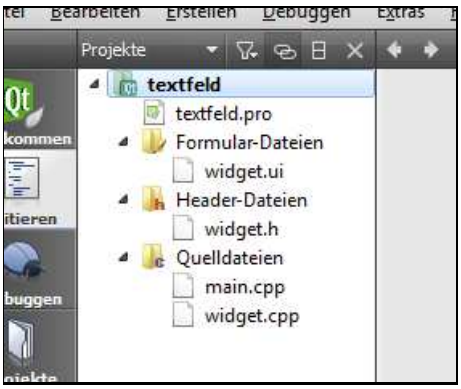
```
1  QObject::connect(&b, SIGNAL(clicked()), &a, SLOT(slot()));
```

d) Starte und testen Sie das Programm, insbesondere den neuen Button. Sie werden **keine** Fehlermeldung erhalten. Dennoch sollte der Button (noch) nicht funktionieren. Wenn Sie aber im *QT Creator* Fenster unten in den Frame *Ausgabe der Anwendung* schauen (ALT+3), so sollten Sie eine Warnung sehen:
Object::connect: No such slot QApplication::slot() in main.cpp:12
Die Warnung besagt, dass das „Ziel“ keinen Slot `slot()` besitzt. Schauen Sie in die Dokumentation des „Ziels“ (**QApplication**) bzw. der Vater-Klasse, um einen geeigneten Slot zu finden.

2 „Texteingabe im Fenster“

Thema: Erstellen eines ersten Programms mit dem Designer

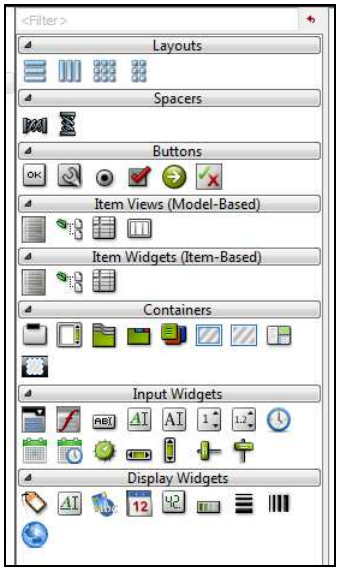
1. Neues Qt4 GUI Projekt erstellen
   a) Qt Creator öffnen
   b) Alle alten Projekte im Qt Creator schließen
   c) Qt-GUI-Anwendung erstellen (Hinweis: Qt Creator erstellt ein Unterverzeichnis mit dem Projektnamen).
   d) **keine** weiteren Module einbinden
   e) Als Basisklasse *QWidget* auswählen
   f) Haken machen in der Checkbox für „Form-Datei generieren“
2. Beachten Sie in der Projektübersicht die Projektstruktur:
   - Projektdatei: „textfeld.pro“ mit allen Informationen zum Projekt
   - Formular-Dateien, in denen das Fensterlayout abgelegt ist
   - Header-Dateien
   - Quelldateien mit der „main.cpp“



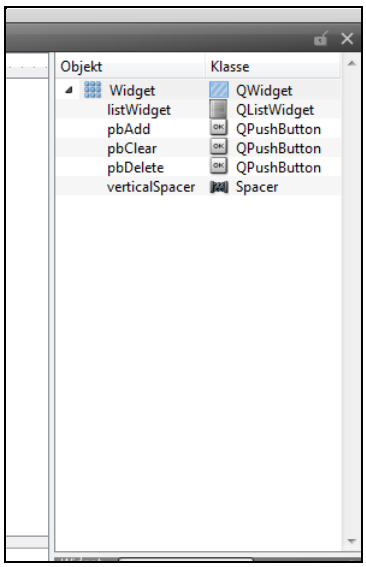
3. Öffnen Sie den Designermodus, indem Sie doppelt auf die Formular-Datei linksklicken

   - In der Toolbar finden Sie Buttons für die vier Bearbeitungsmodi:
     - „Widgets bearbeiten“ F3
     - „Signale und Slots bearbeiten“ F4
     - „Buddies bearbeiten“ (wird erst später benötigt)
     - „Tabulatorreihenfolge bearbeiten“ (wird erst später benötigt)

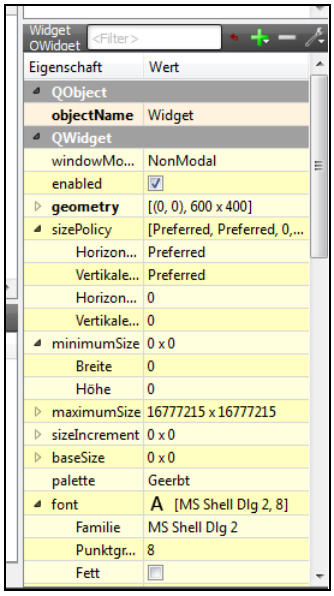
     und Buttons für verschiedene Layouts, u.a.:
     - „Objekte waagrecht anordnen“ CTRL+H
     - „Objekte senkrecht anordnen“ CTRL+L
     - „Objekte tabellarisch anordnen“ CTRL+G
     - „Layout auflösen“

- 

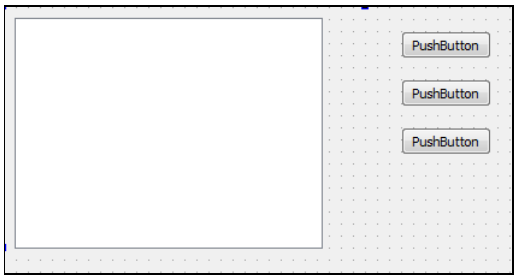
  Links befindet sich die Anzeige aller nutzbaren Widgets. Durch einen Rechtsklick können Sie die Anzeige in den Icon-Modus versetzen (s. Bild)

- 

  Im Objektbaum (rechts oben) wird die Hierarchie der im Fenster verwendeten Objekte angezeigt (im Bild ist das Fenster bereits durch Buttons und ein List-Widget erweitert worden)

- 

  Unten rechts werden die Eigenschaften des aktuell ausgewählten Widgets angezeigt. Dabei wird die Vererbungshierarchie des Widgets herausgestellt.

4. 

   Fügen Sie drei „QPushButton“ und ein „QListWidget“, wie im Bild zu sehen, dem Fenster hinzu.

5. Ändern Sie Objektnamen und Beschriftung der Buttons wie folgt:

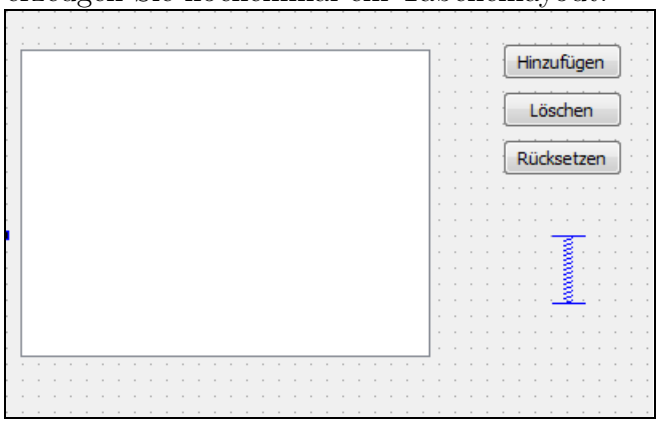
Button	**Objektname**	**Beschriftung**	Funktionalität
oberster Button	pbAdd	Hinzufügen	Textzeile erstellen
mittlerer Button	pbDelete	Entfernen	markierte Zeile entfernen
unterster Button	pbClear	Reset	Textfeld komplett leeren

   Die Beschriftung kann am einfachsten über einen Doppel-Linksklick auf den Button verändert werden. Den Objektnamen können Sie über die Eigenschaften ändern.

6. Schauen Sie sich die Vorschau Ihres Fensters an: *Extras→Formulareditor→Vorschau*. Beachten Sie, dass sich das Fenster nicht korrekt in seiner Größe verändern lässt (die Objekte wandern nicht mit). Um ein sich korrekt verhaltendes Fenster zu erzeugen, muss dem Fenster **immer** ein Layout zu Grunde liegen. Erst

dann „weiß“ das Programm, wie sich das Fenster bei Größenveränderungen zu verhalten hat.

7. Erzeugen Sie für das Fenster ein Tabellenlayout:
   a) Schließen Sie das Vorschaufenster
   b) Klicken Sie so in das Fenster, dass das Fenster selbst und nicht ein einzelnes Objekt des Fensters ausgewählt ist
   c) Klicken Sie dann auf den Button „Objekte tabellarisch anordnen“ in der Toolbar
   d) Sie sehen, wie die vier Elemente (drei Buttons und ein ListWidget) in einer Tabellenform angeordnet werden. Allerdings sieht das Ergebnis noch nicht so gut aus, da die Summe der Höhen der drei Knöpfe kleiner als die Fensterhöhe ist. Zwar kann sich das ListWidget in der Größe anpassen, nicht jedoch die Buttons. Daher werden wir noch eine Modifikation vornehmen:
   e) Zerstören Sie das Fensterlayout über den Button „Layout auflösen“ in der Toolbar.
   f) Fügen Sie einen „Vertical Space“ unter die drei PushButtons hinzu und erzeugen Sie nocheinmal ein Tabellenlayout:



   g) Schauen Sie sich das Fenster in der Vorschau an.

8. Den „Rücksetzen“ Button mit Funktionalität füllen:
   a) Wechseln Sie in den Bearbeitungsmodus: „Signale und Slots bearbeiten“ F4

b) Verbinden Sie den „Rücksetzen“ Button über das Signal *clicked()* mit dem Slot *clear()* vom ListWidget

c) Die Verbindung sollte in der „Signale und Slots“ Übersicht angezeigt werden. Evtl. müssen Sie noch den Karteireiter „Signale und Slots“ ganz unten am Bildschirmrand anwählen

Sender	Signal	Empfänger	Slot
pbClear	clicked()	listWidget	clear()

d) Wechseln Sie zurück zum Bearbeitungsmodus „Widgets bearbeiten“

9. Nun soll der „Hinzufügen“ Button mit Funktionalität gefüllt werden. Da die Verknüpfung dieses Buttons nicht mit einer einfachen Verbindung zwischen Button und ListWidget realisiert werden kann, müssen Sie den Quellcode bearbeiten:

   a) Rechtsklicken Sie auf den „Hinzufügen“ Button und wählen Sie „Slot anzeigen“ aus dem Kontextmenü

   b) Wählen Sie in der Slotliste „clicked()“ aus. Sie werden automatisch im Quellcode an die Stelle mit der verantwortlichen Methode für das ausgewählte „clicked()“ Signal versetzt. Da die Methode bisher nicht existierte, wird Sie auch gleich für Sie angelegt. Der Methodenname wurde vom Creator speziell gewählt und darf nicht verändert werden. Durch den besonderen Namen wird die Methode automatisch mit dem Button verknüpft. Ein `connect(pbAdd, SIGNAL(clicked()), this, SLOT(on_pbAdd_clicked()));` im Konstruktor ist nicht mehr nötig.

   c) Ergänzen Sie den Quellcode für den Slot:

```
28  void Widget::on_addButton_clicked()
29  {
30    QString newText = QInputDialog::getText(this, "Enter text", "Text:");
31    if( !newText.isEmpty() )
32    ui->listWidget->addItem(newText);
33  }
```

   d) Binden Sie noch die Bibliothek „QInputDialog“ ein.

10. Starten Sie die Anwendung CTRL+R. Bis auf den „Löschen“ Buttons sollte die Anwendung einwandfrei funktionieren

11. Implementieren Sie den „Löschen“ Button.
    a) Gehen Sie dazu zunächst genau so vor, wie bei der Implementierung des „Hinzufügen“ Buttons
    b) Schauen Sie dann in die Dokumentation zum ListWidget („QListWidget“). Suchen Sie nach einer Methode, die die ausgewählten Einträge zurück liefert. Achten Sie auf den Datentyp, der zurück geliefert wird!
    c) Bauen Sie eine Schleife innerhalb Ihrer Methode, die jedes zurückgelieferte Item löscht: `delete ...`
    d) Starten und testen Sie die Anwendung.
    e) Verfeinern Sie die Anwendung, indem Sie eine Möglichkeit finden, mehrere Einträge im ListWidget zu markieren. Schauen Sie dazu in die Dokumentation vom ListWidget.

12. Der „Löschen“ Button ist immer nutzbar, auch wenn das ListWidget leer ist oder kein Element ausgewählt wurde. Dies soll nicht so sein.
    a) Gehen Sie zur Header-Datei und schreiben Sie einen *private slot* mit Namen: `updateDeleteEnabled`
    b) Wechseln Sie wieder zur Quelldatei und schreiben Sie das Methodengrundgerüst:

```
42  void Widget::updateDeleteEnabled(void)
43  {
44  }
```

    c) Versuchen Sie die Methode zu implementieren. Schauen Sie dazu in die Dokumentation von *QListWidget*, um heraus zu finden, wie Sie heraus bekommen könne, ob (bzw. wie viele) Items im ListWidget ausgewählt sind, und *QPushButton*, um heraus zu finden, wie Sie den Button aktivieren (engl. enable) bzw. deaktivieren (engl. disable) können.
    d) Anschließend muss der Slot noch verbunden (*connect*) werden. Fügen Sie dazu im Konstruktor nach `ui->setupUi(this);` folgende Zeile ein:

```
connect(ui->listWidget,
        SIGNAL(itemSelectionChanged()),
        this,
        SLOT(updateDeleteEnabled()));
```

Hinweis: Der Quellcode umfasst nur eine Zeile und ist hier aufgrund der Übersichtlichkeit auf mehrere Zeilen aufgeteilt.
Versuchen Sie die Zeile mit Ihrem nachbarn nachzuvollziehen. Schlagen Sie auch die verwendeten Signale in der Dokumentation nach!

e) Starten und testen Sie das Programm. Beachten Sie, dass der „Löschen“ Button nach dem Programmstart solange nutzbar ist, wie noch kein Eintrag zur Liste hinzugefügt wurde. Überlegen Sie, woran das liegt und wie das geändert werden kann (es ist nur eine Programmzeile nötig).

3 „Exit Button 1“

Thema: Vater- und Kind-Fenster; Größe eines *QWidgets* bzw. eines *QPushButtons*; Schriftart; *connect*

1. Leeres Qt4 Projekt mit Namen „exitbutton1“ erstellen (Siehe Beschreibung vom Arbeitsblatt „Hello World“

2. „main.cpp“ - Datei anlegen und dem Projekt hinzufügen (Siehe Beschreibung vom Arbeitsblatt „Hello World“

3. Programm schreiben und starten

   a) Quellcode in „main.cpp“ eingeben:

```
1  #include <QApplication>
2  #include <QWidget>
3
4  int main(int argc, char **argv)
5  {
6      QApplication a(argc, argv);
7      QWidget w;
8      w.show();
9      return a.exec();
10 }
```

   b) Anwendung starten entweder über den grünen Startpfeil auf der linken Seite oder Tastaturkombination: CTRL+R

Erklärungen

Schauen wir uns die Zeilen aus *main* mal genauer an:

```
QApplication a(argc, argv);
```

In dieser Zeile wird ein Objekt der Klasse *QApplication* erstellt. In jeder Fensteranwendung **muss** genau ein Objekt dieser Klasse erzeugt werden.

```
QWidget w;
```

Hier wird ein Objekt der Klasse *QWidget* angelegt. Ein *QWidget* ist ein einfaches Anwendungsfenster (ohne Menüleiste).

```
w.show();
```

Das gerade erstellte Widget-Objekt wird in der nächsten Zeile angezeigt. Ein Objekt ist standardmäßig niemals sichtbar. Es muss immer erst die *show()* Methode zum Objekt aufgerufen werden.

```
return a.exec();
```

Zuletzt wird das Objekt von *QApplication* ausgeführt. Dabei wird die Programmkontrolle von *main* an Qt übergeben. Erst wenn die *QApplication* beendet wird, geht die Programmkontrolle an *main* zurück. In den meisten Fällen, wie auch in diesem Fall, wird im Anschluss auch *main* beendet.

Aufgabe

Ändern Sie *main* und testen Sie:

```
1  #include <QtGui/QApplication>
2  #include <QPushButton>
3  #include <QWidget>
4
5  int main(int argc, char *argv[])
6  {
7      QApplication a(argc, argv);
8      QWidget w;
9      w.show();
10     QPushButton pbExit("Beenden");
11     pbExit.show();
12     return a.exec();
13 }
```

Ändern Sie *main* erneut (Zeile 10) und testen Sie:

```
1  #include <QtGui/QApplication>
2  #include <QPushButton>
3  #include <QWidget>
4
5  int main(int argc, char *argv[])
6  {
```

```
7       QApplication a(argc, argv);
8       QWidget w;
9       w.show();
10      QPushButton pbExit("Beenden",&w);
11      pbExit.show();
12      return a.exec();
13  }
```

Schlagen Sie in der Dokumentation die Klasse *QWidget* nach und achten Sie auf die verschiedenen Konstruktoren. Erklären Sie das unterschiedliche Fenster-Verhalten der beiden Varianten.
Wieso ist der Button noch ohne Funktion?

Aufgabe

Ändern Sie den Titel des Fensters in „Exit Button Übung“ und setzen Sie die Größe des Widgets unveränderbar (engl. fixed) auf $300 \cdot 120$. Schauen Sie dazu in die Dokumentation von *QWidget*.

Aufgabe

Modifizieren Sie das Programm wie folgt:

- Fügen Sie einen *QPushButton* mit der Aufschrift „Beenden“ hinzu.
- Der *QPushButton* soll mit Hilfe der *setGeometry* mit der Höhe von 40 Pixeln und der Breite von 200 Pixeln mittig im Fenster angezeigt werden.
- Die Schrift auf dem Button soll in der Schriftart „Times“ mit 18 Punkten Schriftgröße in Fettdruck erscheinen.
- Ein Klick auf den Button soll über das *connect* Makro mit dem *close()* Slot des Fensters verbunden werden. Damit sollte sich das Fenster auch über den „Beenden“ Button schließen lassen.

Beachten Sie, dass ein seperates Aufrufen der Methode *show()* für den Button nicht nötig ist. Da das Fenster über den *parent* Parameter zu einem Teil des Widgets gemacht worden ist.

4 „Exit Button 2“

Thema: Vater- und Kind-Fenster, selbst erstellte Widgets, Signale & Slots

1. Leeres Qt4 Projekt mit Namen „exitbutton2“ erstellen (Siehe Beschreibung vom Arbeitsblatt „Hello World“

2. „main.cpp“ - Datei anlegen und dem Projekt hinzufügen (Siehe Beschreibung vom Arbeitsblatt „Hello World“

3. Programm schreiben und starten

```
1  #include <QApplication>
2  #include <QFont>
3  #include <QPushButton>
4  #include <QWidget>
5
6  class MyWidget : public QWidget
7  {
8  public:
9   MyWidget(QWidget *parent = 0);
10 };
11
12 MyWidget::MyWidget(QWidget *parent)
13   : QWidget(parent)
14 {
15 }
16
17 int main(int argc, char *argv[])
18 {
19   QApplication app(argc, argv);
20   MyWidget w;
21   w.show();
22   return app.exec();
23 }
```

Erklärungen

Die benutzerdefinierte Klasse *MyWidget* ist von der Qt-Klasse *QWidget* vererbt.

```
class MyWidget : public QWidget
```

Das Schlüsselwort *public* in der Vererbungsbeziehung gibt an, dass alle *public* und *protected* deklarierten Elemente der Vaterklasse (*QWidget*) auch aus der benutzerdefinierten Kindklasse (*MyWidget*) erreichbar sind. Lediglich *private* deklarierte Elemente der Vaterklasse werden nicht mitvererbt. Somit können alle öffentlichen (*public*) und geschützten (*protected*) Elemente aus QWidget genutzt werden, die privaten Elemente (*private*) jedoch nicht.

Aufgabe

Ändern Sie durch entsprechende Zeilen im Konstruktor von *MyWidget* den Titel des Fensters in „Mein eigenes Widget“ und die (feste/unveränderliche) Größe des Widgets auf $300 \cdot 120$.

Erklärungen

Schauen wir uns mal die Klasse etwas genauer an:

```
MyWidget(QWidget *parent = 0);
```

Hier wird der Konstruktor deklariert. Der Parameter für den Konstruktor wird, sofern bei einem Aufuf nicht angegeben, mit dem Standardwert *0* gefüllt. Dies ist der Fall, wenn es keinen Vater für das Widget geben soll. Wie in *main* zu sehen ist, wird das Widget ohne Parameter, also auch ohne Vater (*parent*) initialisiert.
ACHTUNG: Die Angaben eines *parents* als Parameter hat nichts mit Klassenvererbung zu tun. Es gibt lediglich die Fensterzugehörigkeiten wieder. Ein Kind-Widget eines Fensters wird innerhalb des *parent* angezeigt. Hat ein Widget keinen *parent*, wird es als eigenständiges Fenster angezeigt (s. Arbeitsblatt „Exit Button 1“).
Die Definition des Konstruktors sieht ebenfalls etwas ungewohnt aus:

```
MyWidget::MyWidget(QWidget *parent) :
    QWidget(parent)
```

Der Quelltext nach dem einzelnen Doppelpunkt ist eine Erweiterung des Konstruktors für eine vererbte Klasse. *MyWidget* ist von der Qt-Klasse *QWidget* vererbt.

Dem Standardkonstruktor von *QWidget* kann ein Parameter *parent* übergeben werden.

```
QWidget ( QWidget *parent = 0, Qt::WindowFlags f = 0 )
```

Wenn nun eine Objekt *MyWidget* erzeugt wird, wird der übergebene *parent* Parameter an die Vaterklasse *QWidget* weiter gereicht.

Aufgabe

Modifizieren Sie das Programm auch diesmal wie unten angegeben. ABER modifizieren Sie Ihren Quellcode NICHT in *main* wie beim Arbeitsblatt „Exit Button 1“ sondern in der Klasse *MyWidget*.
Der Quellcode von *main* soll NICHT verändert werden!

- Fügen Sie einen *QPushButton* mit der Aufschrift „Beenden“ hinzu.
- Der *QPushButton* soll mit Hilfe der *setGeometry* mit der Höhe von 40 Pixeln und der Breite von 200 Pixeln mittig im Fenster angezeigt werden.
- Die Schrift auf dem Button soll in der Schriftart „Times“ mit 18 Punkten Schriftgröße in Fettdruck erscheinen.
- Ein Klick auf den Button soll über das *connect* Makro mit dem *close()* Slot des Fensters verbunden werden. Damit sollte sich das Fenster auch über den „Beenden“ Button schließen lassen.

5 „Digitale Anzeigeelemente“

Thema: QLCDNumber und QSlider, Fenster-Layouts

1. Leeres Qt4 Projekt mit Namen „anzeigeelemente1“ erstellen (Siehe Beschreibung vom Arbeitsblatt „Hello World“

2. „main.cpp“ - Datei anlegen und dem Projekt hinzufügen (Siehe Beschreibung vom Arbeitsblatt „Hello World“

3. Programm schreiben und starten

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

class MyWidget : public QWidget
{
public:
  MyWidget(QWidget *parent = 0);
};

MyWidget::MyWidget(QWidget *parent)
: QWidget(parent)
{
  QPushButton *pbExit = new QPushButton(tr("Beenden"), this);
  pbExit->setFont(QFont("Times", 18, QFont::Bold));

  // Signale und Slots verbinden
  connect(pbExit, SIGNAL(clicked()), qApp, SLOT(quit()));
}

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  MyWidget w;
  w.show();
  return app.exec();
}
```

Aufgabe

Fügen Sie folgende Elemente (im Konstruktor) zum Widget *MyWidget* hinzu:

- *QLCDNumber* als Zeiger mit Namen „lcd“
  - Setzen Sie den Segment-Stil auf *Filled*
  - Setzen Sie die Anzahl an Ziffern auf zwei fest
- *QSlider* als Zeiger mit Namen „slider“ in horizontaler Richtung
  - Setzen Sie den Zahlenbereich auf 0 bis 99 fest
  - Setzen Sie den aktuellen Wert auf 0

Verbinden Sie dann noch mit einem *connect* den Slider mit der LCD-Number. ACHTUNG: Das Programm ist noch nicht funktionsfähig. Lesen Sie bitte vor einer Ausführung erst weiter.

Erklärungen und Aufgabe

Ein Fenster mit mehreren Elementen muss immer ein Layout bekommen. Andernfalls ist nicht festgelegt, wie die einzelnen Fensterelemente (beim Verändern der Fensterabmessungen) angeordnet werden sollen. Die beiden letzten Arbeitsblätter (*Exit Button1/2*) haben das Problem dadurch umgangen, dass das einzige Fensterelement mit absoluten Geometriewerten in einem unveränderlichen Fenster angelegt wurde. Verwenden Sie folgende Programmzeilen für dieses Projekt:

```
QVBoxLayout *layout = new QVBoxLayout;    // Ein neues Layout wird erstellt
layout->addWidget(pbExit);    // Der PushButton wird zum Layout hinzugefügt
layout->addWidget(lcd);    // Die LCD Number wird zum Layout hinzugefügt
layout->addWidget(slider);    // Der Slider wird zum Layout hinzugefügt
setLayout(layout);    // Das Layout wird auf unser Fenster angewendet
```

Hier wird das *QVBoxLayout* verwendet, um die Fensterelemente vertikal untereinander anzuordnen. Es gibt drei wichtige Layouts in Qt:

- *QVBoxLayout*, für eine vertikale Ausrichtung
- *QHBoxLayout*, für eine horizontale Ausrichtung
- *QGridLayout*, für eine tabellarische Ausrichtung

Durch den Aufruf von `setLayout(layout)` wird das *QVBoxLayout* zu einem Teil von *MyWidget*.

Compilieren, starten und testen Sie das Programm!

6 „Digitale Anzeigeelemente en masse“

Thema: Widgets kapseln

1. Leeres Qt4 Projekt mit Namen „anzeigeelemente2“ erstellen
2. „main.cpp“ - Datei anlegen und dem Projekt hinzufügen

3. Programm schreiben und starten

```
1  #include <QApplication>
2  #include <QtCore> // Diese Bibliothek bindet die wichtigsten, grundlegenden
       Qt-Bibliotheken ein
3  #include <QtGui>     // Diese bibliothek bindet die wichtigsten grafischen
       Qt-Bibliotheken ein
4
5  class LCDRange : public QWidget
6  {
7  public:
8      LCDRange(QWidget *parent = 0);
9  };
10
11 LCDRange::LCDRange(QWidget *parent)
12     : QWidget(parent)
13 {
14     QLCDNumber *lcd = new QLCDNumber(2);
15     lcd->setSegmentStyle(QLCDNumber::Filled);
16
17     QSlider *slider = new QSlider(Qt::Horizontal);
18     slider->setRange(0, 99);
19     slider->setValue(0);
20     connect(slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)));
21
22     QVBoxLayout *layout = new QVBoxLayout;
23     layout->addWidget(lcd);
24     layout->addWidget(slider);
25     setLayout(layout);
26 }
27
28 class MyWidget : public QWidget
29 {
30 public:
31     MyWidget(QWidget *parent = 0);
32 };
33
34 MyWidget::MyWidget(QWidget *parent)
35     : QWidget(parent)
36 {
37     setWindowTitle("Anzeigeelemente 2");
38
39     QPushButton *pbExit = new QPushButton(tr("Beenden"));
40     pbExit->setFont(QFont("Times", 18, QFont::Bold));
41
42     connect(pbExit, SIGNAL(clicked()), qApp, SLOT(quit()));
43
44   // Hier müssen neun LCDRange - Objekte erzeugt und in ein GridLayout
       gepackt werden!
45
46     QVBoxLayout *layout = new QVBoxLayout;
47     layout->addWidget(pbExit);
```

```
48      //layout->addLayout(grid);
49      setLayout(layout);
50  }
51
52  int main(int argc, char *argv[])
53  {
54      QApplication app(argc, argv);
55      MyWidget widget;
56      widget.show();
57      return app.exec();
58  }
```

Erklärungen

Werden viele verschiedene Qt-Bibliotheken in einem Programm genutzt, ist es meist einfacher, die beiden umfassenden Bibliotheken *QtCore* für Basisfunktionalitäten und *QtGui* für Fensterelemente einzubinden (s. Quellcode).
Die benutzerdefinierte Klasse *LCDRange* ist von der Qt-Klasse *QWidget* vererbt und erstellt eine *QLCDNumber* samt einem *QSlider*. Die beiden Elemente sind über ein *connect* miteinander verbunden und über *QVBoxLayout* übereinander angeordnet.

Aufgabe

Es soll nun ein Fenster mit neun digitalen Anzeigeelementen samt Slidern erstellt werden.

Die Klasse für die Slider samt LCD-Anzeige ist bereits fertig implementiert. Es müssen lediglich neun Instanzen der klasse *LCDRange* erstellt und dann in ein

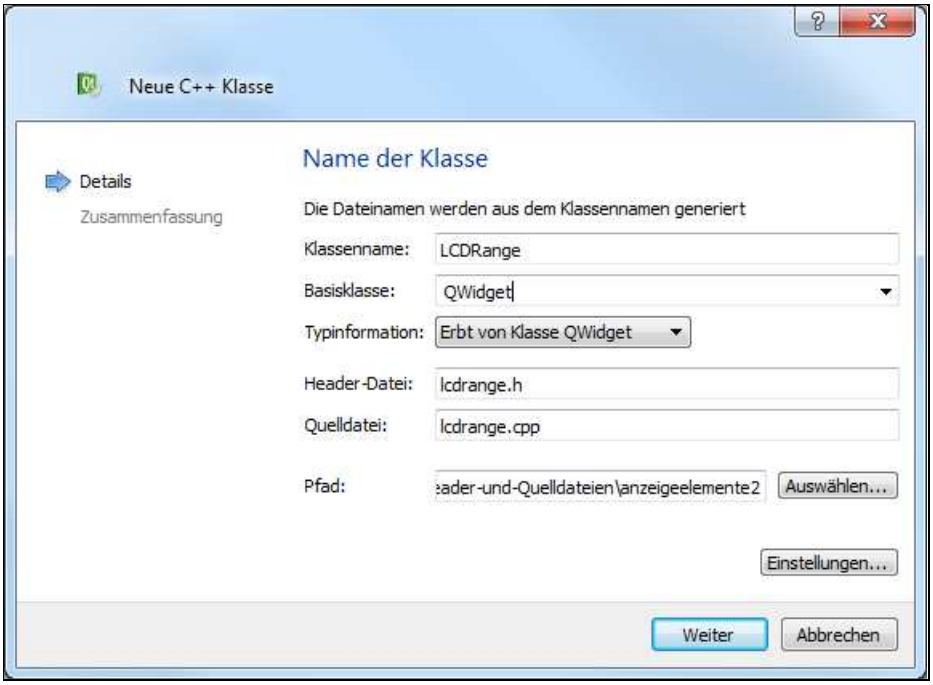
QGridLayout mit Namen *grid* gesteckt werden. Nutzen Sie dafür eine Schleife. Sobald die Zeile 48 `layout->addLayout(grid)` einkommentiert wurde, werden die Elemente angezeigt. Compilieren und testen Sie das Programm.

Aufgabe

Es soll nun mit Header- und Quelldateien gearbeitet werden. Lagern Sie dazu die Klasse `class LCDRange : public QWidget` in die Dateien „lcdrange.h“ bzw. „lcdrange.cpp“ aus. Gehen Sie dazu wie folgt vor:

1. Öffnen Sie das Menü:
   Datei→Neu...→C++ Klasse

2. Wählen Sie folgende Einstellungen für die neue Klasse:
   - Klassenname: *LCDRange*
   - Basisklasse: *QWidget*
   - Header-Datei: *lcdrange.h*
   - Quelldatei: *lcdrange.cpp*



3. Modifizieren Sie Ihren Quellcode

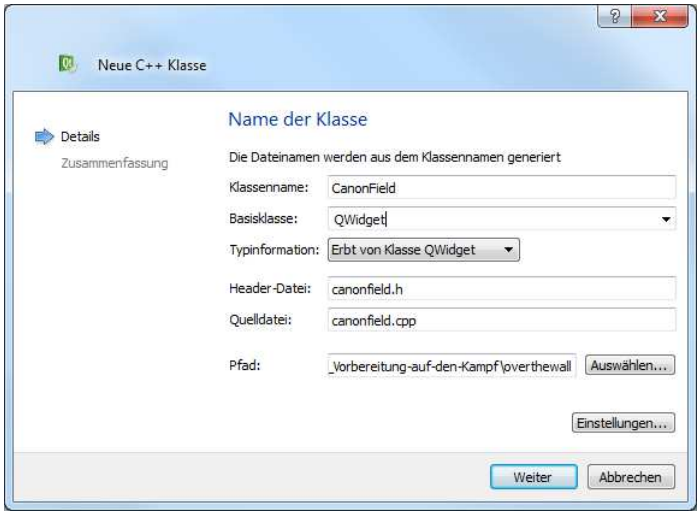
- Klasse aus *main.cpp* entfernen und in die beiden neuen Dateien „lcdrange.h“ bzw. „lcdrange.cpp“ kopieren
- Anpassen der `#include` Anweisungen

4. Compilieren und testen

7 „Vorbereitung auf den Kampf“

Thema: Zeichnen mit *QPaintEvent*

1. Leeres Qt4 Projekt mit Namen „overthewall“ erstellen
2. Fügen Sie, wie im letzten Projekt beschrieben, die C++ Klasse `class CannonField : public QWidget` mit den beiden Dateien *canonfield.h* und *canonfield.cpp* zum Projekt hinzu:



3. Da der im letzten Projekt erstellte Quellcode in diesem Projekt sinnvoll nutzbar ist, kopieren Sie bitte die folgenden Dateien in den neuen Projektordner „overthewall“
   - „main.cpp“
   - „lcdrange.h“
   - „lcdrange.cpp“

4. Wie Sie im QtCreator erkennen können, gehören die kopierten Dateien aber noch nicht zum neuen Projekt. Um die neuen Dateien dem Projekt hinzuzufügen, müssen Sie die Projektdatei „overthewall.pro“ im QtCreator öffnen und wie folgt modifizieren:

```
1 SOURCES += main.cpp \
2         lcdrange.cpp \
3     canonfield.cpp
4
5 HEADERS += lcdrange.h \
6     canonfield.h
```

Erklärungen

Die Projektdatei (immer mit der Endung „.pro“ bei Qt) beinhaltet alle Informationen zum Projekt, wie notwendige Module (mySQL, Netwerk-Modul, etc.) und alle Quellcode- bzw. Headerdateien. Damit handelt es sich bei der Projektdatei um den zentralen Punkt eines Softwareprojektes. In Ihrer Projektdatei können Sie zwei Abschnitte erkennen:

- Unter dem Punkt „SOURCES“ in der Projektdatei werden alle Quellcodedateien aufgelistet. Dabei steht in jeder Zeile ein Dateiname. Alle Zeilen, bis auf die letzte Zeile dieses Abschnittes, enden mit einem Backslash.
- Unter dem Punkt „HEADERS“ in der Projektdatei werden alle Headerdateien aufgelistet. Auch hier steht in jeder Zeile ein Dateiname und jede Zeile, bis auf die letzte, werden mit einem Backslash abgeschlossen.

Das „+ =“ in den beiden Abschnitten bedeutet, dass die folgenden Dateien hinzugefügt werden. Dadurch ist es möglich, auch an anderen Stellen weitere Dateien hinzuzufügen ohne die ursprüngliche Liste von Dateien zu überschreiben, wie es bei einem einfachen „=“ sein würde.

Aufgabe

Modifizieren Sie den Quellcode der Datei „main.cpp“ wie folgt:

```
1 #include <QApplication>
2 #include <QtCore>
3 #include <QtGui>
```

```
4  #include "lcdrange.h"
5  #include "canonfield.h"
6
7  class MyWidget : public QWidget
8  {
9  public:
10   MyWidget(QWidget *parent = 0);
11 };
12
13 MyWidget::MyWidget(QWidget *parent)
14     : QWidget(parent)
15 {
16   setWindowTitle("Over the wall");
17
18   QPushButton *pbExit = new QPushButton(tr("Beenden"));
19   pbExit->setFont(QFont("Times", 18, QFont::Bold));
20
21   // Hier müssen noch *lcdRange und *canonField erstellt werden
22
23   QGridLayout *gridLayout = new QGridLayout;
24   gridLayout->addWidget(pbExit, 0, 0);
25   // Hier müssen noch lcdrange und canonField in das QGridLayout gepackt
         werden
26   gridLayout->setColumnStretch(1, 10);
27   setLayout(gridLayout);
28
29   // angle->setValue(60); // Startwert setzen (wird später implementiert)
30   // angle->setFocus();   // Tastaturfokus auf die Klasse LCDRange setzen
31
32   connect(pbExit, SIGNAL(clicked()), qApp, SLOT(quit()));
33 }
34
35 int main(int argc, char *argv[])
36 {
37   QApplication app(argc, argv);
38   MyWidget widget;
39   // Fenster an der Position (100,100) mit der Größe 500 x 355 anzeigen
40   widget.setGeometry(100, 100, 500, 355);
41   widget.show();
42   return app.exec();
43 }
```

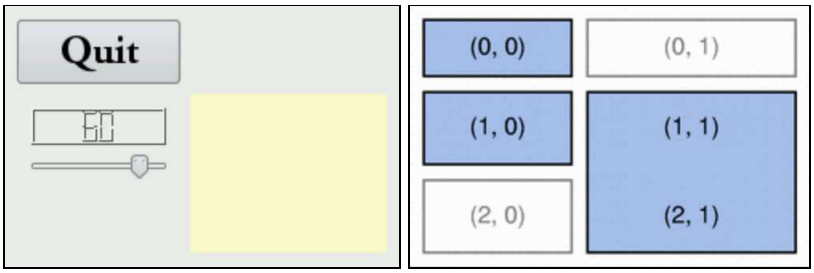
Compilieren und testen Sie das Programm. Ihr Programmfenster sollte wie folgt aussehen:

Erklärung

Das im Quellcode verwendete *QGridLayout* wird verwendet, um Layouts in Tabellenform zu erstellen. Dieses Layout ist wesentlich mächtiger als die beiden Layouts *QVBoxLayout* und *QHBoxLayout*. Momentan haben wir nur einen *QPushButton*, so dass das Tabellenlayout (noch) wenig Sinn macht. Es werden aber weitere Elemente folgen.
Beim *QGridLayout* müssen keine Abmessungen, sondern nur Tabellenzellen angegeben werden. Die Breiten bzw. Höhen der einzelnen Spalten (engl. column) bzw. Reihen (engl. row) der Tabelle werden entsprechend der Inhalte ausgerichtet.
Für das *QGridLayout* erwartet der Befehl `gridLayout->addWidget(quit, 0, 0)` drei Argumente:

1. Das Widget, welches in eine Tabellenzelle gepackt werden soll
2. Die x-Koordinate der Tabellenzelle, in die das Widget soll
3. Die y-Koordinate der Tabellenzelle, in die das Widget soll

In der folgenden Grafik sehen Sie auf der linken Seite das angestrebte Layout für das Programmfenster. Auf der rechten Seite ist das Tabellenlayout mit Koordinatenbeschriftung gezeigt. So hat das linke obere Widget, der *QPushButton*, im Programm die Koordinaten (0,0). Das Widget *LCDRange* soll an der Koordinate (1,0) und das Widget *CanonField* über die Koordinaten (1,1) und (2,1) angezeigt werden.



Die Programmzeile

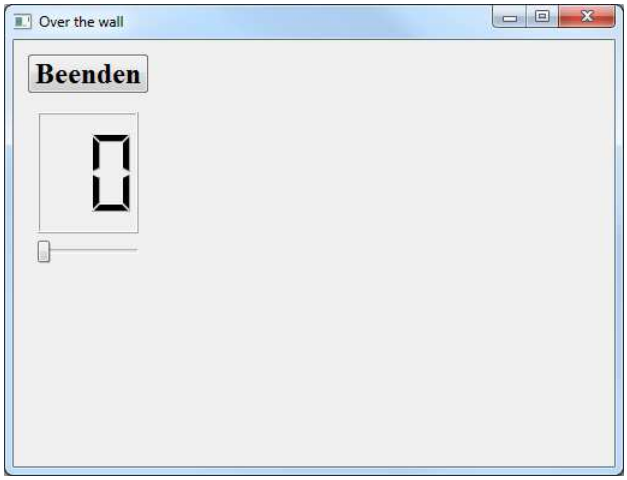
```
26      gridLayout->setColumnStretch(1, 10);
```

steuert das Verhalten des Fensters beim Verändern der Größe. Normalerweise versucht Qt eine Tabellenspalte immer mit der gleichen Breite darzustellen. Mit der

Methode *setColumnStretch* kann dieses Verhalten geändert werden. Der erste Parameter der Methode gibt die entsprechende Spalte an. Der zweite Parameter gibt an, mit welchem Faktor diese Spalte bei einer Größenveränderung des Fensters mitwächst/-schrumpft. Genauso wie das Verhalten der Breiten aller Spalten über den Stretchfaktor angegeben werden kann, kann auch das Verhalten der Höhen festgelegt werden. Die entsprechende Methode lautet: *setRowStretch*.

Aufgabe

Ergänzen Sie den Quellcode so, dass im Programmfenster jeweilse ein Objekt der Klassen/Widgets *LCDRange* und *CanonField* dargestellt werden. Achtung: Das Widget *CanonField* hat noch keinen Inhalt. Daher ist dieses Widget auch noch nicht sichtbar im Programmfenster. Fügen Sie es aber denoch schon zum Layout hinzu.



Aufgabe

Nun soll es möglich werden, auf die Digitalanzeige *lcdrange* von *MyWidget* aus mehr Einfluss zu nehmen. Stellen Sie zunächst sicher, dass der Quellcode der Klasse wie folgt aussieht.

Datei „lcdrange.h“:

```
1 #ifndef LCDRANGE_H
2 #define LCDRANGE_H
3
```

```
4  #include <QtCore>
5  #include <QtGui>
6
7  class LCDRange : public QWidget
8  {
9    Q_OBJECT
10
11 public:
12   LCDRange(QWidget *parent = 0);
13
14 public slots:
15
16 signals:
17
18 protected:
19
20 private:
21
22 };
23
24 #endif // LCDRANGE_H
```

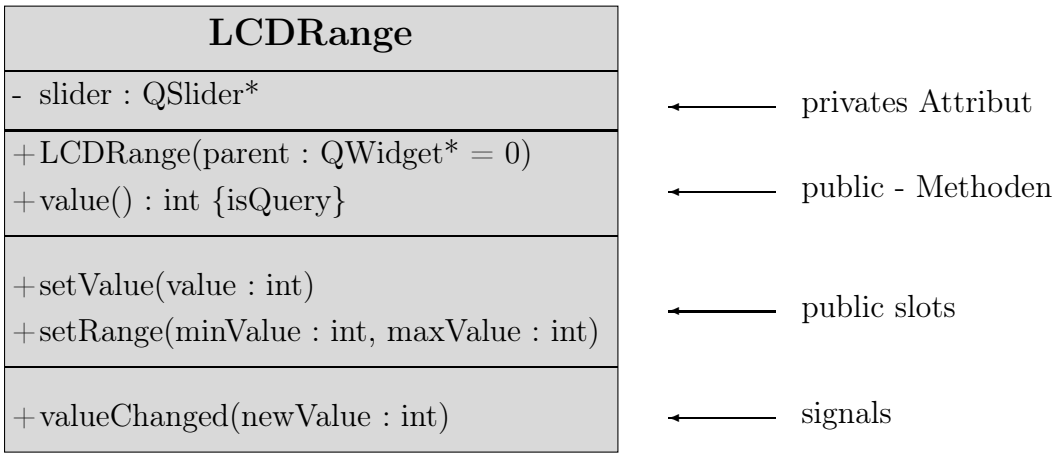
Beachten Sie, dass die Bereiche für Slots und Signale bereits vorbereitet sind. Das Schlüsselwort *protected* stellt eine dritte Möglichkeit zu *private* und *public* dar, welche für Vererbungsbeziehungen zwischen Vater-Kind-Klassen wichtig wird.
Das Makro Q_OBJECT muss im privaten Bereich der jeder Klassedeklaration stehen, welche Signale oder/und Slots nutzt.

Datei „lcdrange.cpp“:

```
1  #include "lcdrange.h"
2
3  LCDRange::LCDRange(QWidget *parent)
4      : QWidget(parent)
5  {
6    QLCDNumber *lcd = new QLCDNumber(2);
7    lcd->setSegmentStyle(QLCDNumber::Filled);
8
9    // ACHTUNG: Das Attribut "slider" ist jetzt fest in der Headerdatei
         deklariert!!!
10   // Dies ist notwendig, da in verschiedenen Methoden dieser Klasse der
         Wert
11   // vom Slider abgefragt oder gesetzt werden muss!
12   slider = new QSlider(Qt::Horizontal);
13   slider->setRange(0, 99);
14   slider->setValue(0);
15
16   connect(slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)));
```

```
17    connect(slider, SIGNAL(valueChanged(int)), this, SIGNAL(valueChanged(int)
          ));
18
19    QVBoxLayout *layout = new QVBoxLayout;
20    layout->addWidget(lcd);
21    layout->addWidget(slider);
22    setLayout(layout);
23
24    setFocusProxy(slider); // Den Tastaturfokus von dieser Klasse auf Slider
          setzen
25  }
26
27  void LCDRange::setRange(int minValue, int maxValue)
28  {
29    // Hier fehlt noch Code: Wenn der Bereich nicht zulässig ist, dann ...
30    qWarning("LCDRange::setRange(%d, %d) \n"
31          "\tBereich muss zwischen 0..99 liegen \n"
32          "\tund minValue darf nicht größer als maxValue sein!",
33          minValue, maxValue);
34    // Hier fehlt evtl. noch Code: Ansonsten setze den neuen Bereich ...
35  }
```

Beachten Sie die hinzugekommene Zeile 14. Diese Zeile bewirkt, dass das Klasseneigene (*this*) Signal *valueChanged(int)* gesendet wird sobald sich der Wert des Sliders (*slider*) ändert (*valueChanged(int)*). Dieses Signal muss aber noch zu der Klasse QLCDRange hinzugefügt werden. Dabei bekommt die Methode *valueChanged* keinen Quellcode, sondern muss nur als Methode in der Headerdatei deklariert werden.
Modifizieren bzw. ergänzen Sie die Klasse *LCDRange* wie im Klassendiagramm angegeben:



HINWEISE: Die Methoden *value()*, *setValue(int value)* sollten selbsterklärend

sein. Die Methoden *setRange(int minValue, int maxValue)* setzt den Zahlenbereich des Sliders. *setValue(int value)* und *setRange(int minValue, int maxValue)* werden als *public slots* deklariert. Damit können diese beiden Methoden auch in einem *connect* Befehl als Slot verwendent werden. Wir werden das später ausnutzen. Die Methode *void valueChanged(int newValue)* wird als Signal deklariert. Das bedeutet, dass Sie dafür keine Methodendefinition in die Datei „lcdrange.cpp“ programmieren müssen. Die Methode wird nur in der Headerdatei im entsprechenden Bereich deklariert. Wir werden dieses Signal ebenfalls später verwenden.

Aufgabe

Zuletzt fehlt noch eine Ausgabe in unserem Widget *canonField*. Stellen Sie sicher, dass der Quellcode der Klasse dem Folgenden entspricht:

Datei „canonfield.h“:

```
1  #ifndef CANONFIELD_H
2  #define CANONFIELD_H
3
4  #include <QtGui>
5
6  class CanonField : public QWidget
7  {
8    Q_OBJECT
9
10 public:
11   CanonField(QWidget *parent = 0);
12   int angle() const { return currentAngle; }
13
14 public slots:
15   void setAngle(int angle);
16
17 protected:
18   void paintEvent(QPaintEvent *event);
19
20 private:
21   int currentAngle;
22 };
23
24 #endif
```

Erklärung

Die Methode `int angle() const { return currentAngle; }` ist eine sogenannte Inline-Methode. Dies bedeutet, dass sie in der Headerdatei komplett definiert (Quellcode ist komplett) und nicht nur deklariert (nur der Methodenkopf) ist. Die Methode soll den aktuell eingestellten Winkel, also den Wert des Sliders bzw. der LCDNumber, zurück liefern.

`void setAngle(int angle)` aus Zeile 15 ist ein Slot und wird später dazu genutzt den aktuellen Winkel einer Kanone im Widget einzustellen. Zunächst soll erstmal nur der eingestellte Winkel im Widget durch einen Zahlenwert dargestellt werden.

Das PaintEvent aus Zeile 21 ist eigentlich eine von *QWidget* geerbte Methode. Es handelt sich um einen sogenannten EventHandler, also um eine Methode, die sich um die Abarbeitung bestimmter Ereignisse kümmert. Die *QWidget* Methode wird automatisch immer dann aufgerufen, wenn das Fenster bzw. der Fensterbereich neu gezeichnet werden muss. Dies kann zum Beispiel dadurch passieren, dass sich die Größe des Fenster verändert hat oder das Fenster durch überdeckende andere Programme wieder sichtbar wird. Zudem kann ein PaintEvent auch manuel durch aufruf von `update()` angestoßen werden. Da es die PaintEvent Methode bereits in der Vaterklasse *QWidget* gibt, bräuchten wir uns eigentlich nicht darum zu kümmern. Allerdings soll in diesem Widget ja immer der aktuell eingestellte Abschusswinkel unserer Kanone als Zahlenwert sichtbar sein. Dies kann der Standard-PaintEvent nicht leisten. Daher müssen wir den PaintEvent überschreiben. Dies bedeutet, dass wir eine eigene Methode `void paintEvent(QPaintEvent *event)` programmieren müssen. Anstatt der QWidget-Methode wird dann in Zukunft immer unsere CanonField-Methode aufgerufen.

Schauen wir uns mal den Quellcode der Quelldatei *canonfield.cpp* an.

Datei „canonfield.cpp“:

```
1  #include <QtGui>
2  #include "canonfield.h"
3
4  CanonField::CanonField(QWidget *parent)
5      : QWidget(parent)
6  {
7      currentAngle = 45;
8      // Neue Farbpalette für das Widget setzen
9      setPalette(QPalette(QColor(250, 250, 200)));
10     // Hintergrund immer automatisch füllen
```

```
11    setAutoFillBackground(true);
12  }
13
14  void CanonField::setAngle(int angle)
15  {
16    if (angle < 5)   angle = 5;
17    if (angle > 70)  angle = 70;
18    if (currentAngle == angle)  return;
19    currentAngle = angle;
20    update(); // Hiermit wird ein PaintEvent angestossen
21  }
22
23  // Eigentlich sollte im Methodenkopf noch der Übergabeparameter stehen.
24  // Da dieser in der Methode aber nicht genutzt wird, wird er weg gelassen,
25  // um keine Warnung beim Compilieren auszulösen
26  void CanonField::paintEvent(QPaintEvent*)
27  {
28    // Hier wird ein "Zeichenstift" erstellt
29    QPainter painter(this);
30    // An die Position (200,200) einen Text schreiben
31    painter.drawText( /*HIER FEHLT NOCH ETWAS*/ );
32  }
```

Aufgabe

Vervollständigen Sie den Aufruf `painter.drawText(...)`. Es soll der aktuell eingestellte Winkel angezeigt werden.

Aufgabe

Zudem fehlt noch eine Verbindung, damit sich die Anzeige im Widget *canonField* tatsächlich aktualisieren kann. Fügen Sie die fehlende *connect* Verbindung zum Quellcode hinzu. Schauen Sie sich dazu die bestehenden Signale und Slots der verschiedenen Klassen an. TIP: Die Verbindung kann im Konstruktor der Klasse *MyWidget* hergestellt werden.

Aufgabe

Überprüfen Sie, ob der Slider mit den Pfeiltasten, Bild auf bzw. ab Tasten und Pos1 bzw. Ende Taste steuerbar ist.

Aufgabe

Ist der Slider auch noch über Tastatur steuerbar, wenn Sie die Zeile angle->setFocus() in *MyWidget* auskommentieren?

Aufgabe

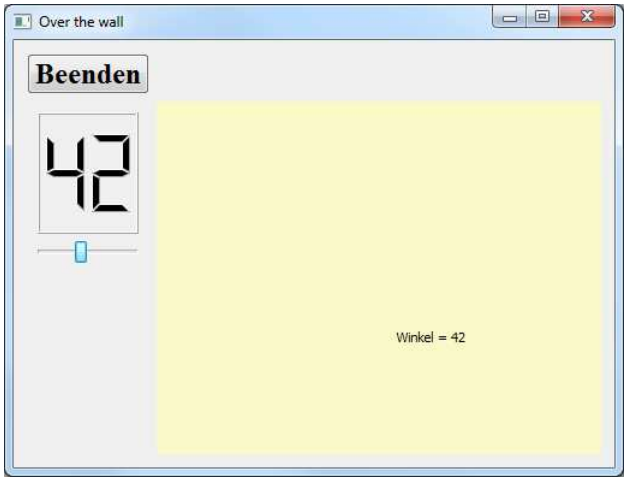
Testen Sie, was passiert, wenn Sie der linken Spalte im GridLayout einen Stretchfaktor ungleich Null geben?

Aufgabe

Was passiert, wenn Sie die Beschriftung des *Beenden* Buttons auf *Be&enden* ändern?

Aufgabe

Zentrieren Sie den Text im Widget *canonField*



8 Jetzt wird gezeichnet

Thema: Zeichnen mit *QPaintEvent* Nun soll endlich eine kleine, blaue Kanone gezeichnet werden. Dazu ist es notwendig das *paintEvent* zu modifizieren. Der text mit dem eingestellten Winkel verschwindet und statt dessen wird eine Kanone

gezeichnet. Schauen wir uns den Quellcode des *paintEvent* genauer an. zunächst erstellen Sie einen *QPainter*, welcher auf dem Widget zeichnen kann:

```
1  void CanonField::paintEvent(QPaintEvent * /* event */)
2  {
3    QPainter painter(this);
```

Die Kanten, die der *painter* beim Zeichnen von geometrischen Strukturen erzeugt, werden durch den Stift (engl. Pen) definiert. Da wir keine sichtbaren Kanten (Umrandungen) haben wollen, „entfernen“ wir den Stift:

```
4    painter.setPen(Qt::NoPen);
```

Dann legen wir die Füllfarbe (blau) geometrischer Strukturen fest. Es soll eine blaue, vollständig gefüllte Fläche entstehen. Es wäre hier auch möglich, andere Farben oder Füllmuster festzulegen.

```
5    painter.setBrush(Qt::blue);
```

Der Koordinatenursprung (0,9) befindet sich im Normalfall immer links oben. Die x-Achse zeigt nach rechts und die y-Achse nach unten. Diese „Voreinstellung“ kann durch die Funktion *QPainter::translate()* geändert werden. Wir setzen den Ursprung auf die linke, untere Ecke des Widgets. Dabei müssen Sie aber daran denken, dass die y-Achse noch immer nach unten zeigt. Alle sichtbaren Punkte haben nach der Verschiebung daher negative y-Werte. Die Parameter des folgenden Befehls geben die Verschiebung an. In x-Richtung wird nicht verschoben. In y-Richtung wird um die Widgethöhe (*rect().height()*) verschoben. Die Funktion *rect()* gibt die Widgetabmessungen zurück. Die Funktion nimmt dann davon nur die Höhe:

```
6    painter.translate(0,rect().height());
```

Jetzt wird der Bauch der Kanone gezeichnet. Dafür verwenden wir ein Kuchenstück. Die Funktion *QPainter::drawPie()* wird dafür verwendet. Der Funktion wird ein Rechteck, in der der Kuchen gezeichnet wird und ein Start- bzw. ein Endwert für den Winkel des Kuchenausschnitts übergeben. **Achtung:** Winkelmaße werden in Qt in sechzehntel eines Grades angegeben! Der Winkel 0 ist bei 3:00 Uhr. Die Zählrichtung ist mathematisch positiv, also entgegengesetzt des Uhrzeigersinnes. Es soll ein viertel Stück (90°) des Kuchens mit dem Radius 35 in der linken, unteren Ecke des Widgets gezeichnet werden:

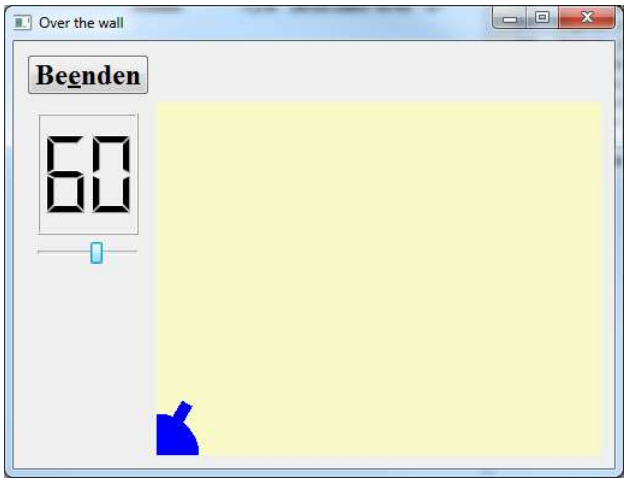
```
7    painter.drawPie( QRect( -35, -35, 70, 70), 0, 90*16 );
```

Nun soll noch ein Kanonenrohr in Form einen Rechtecks an den Kanonenbauch angehängt werden. Das Komplizierte daran ist, dass das Rohr in Abhängigkeit des eingestellten Winkels an der Kanone befinden muss. Daher rotieren wir nun das Koordinatensystem im Uhrzeigersinn so weit, wie der eingestellte Winkel es angibt. Beachte, dass der Rotationswinkel als Fließkommazahl und **nicht** in sechzehntel angegeben werden muss:

```
8     painter.rotate( -currentAngle );
```

Dann muss nur noch das Rechteck als Kanonenrohr gezeichnet werden:

```
9     painter.drawRect( QRect( 30, -5, 20, 10 ) );
```



Aufgaben

1. Compilieren Sie das Programm und testen Sie die Funktion der Winkelverstellung
2. Verwenden Sie einen anderen Pen zum Zeichnen
3. Verwenden Sie ein Füllmuster zum Zeichnen

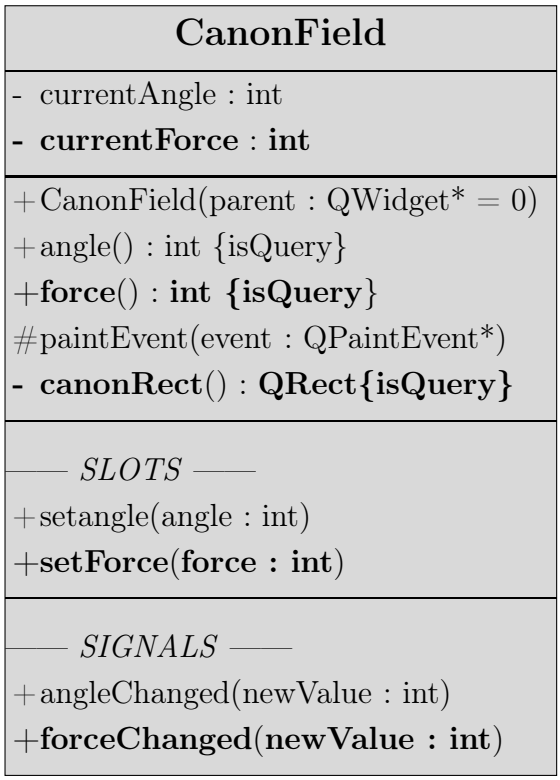
9 „Schusskraft“

Thema: Zeichnen mit *QPaintEvent*

Das Programm soll nun durch eine Möglichkeit, die Schusskraft zu verändern, erweitert werden. Dazu ist folgendes zu programmieren:

- Dem grafischen Benutzerinterface (UI) wird ein weiteres Objekt *LCDNumber* hinzugefügt. Dieses zusätzliche Widget ist für das Einstellen der Schusskraft zuständig. Die Modifikation geschieht in der datei *main.cpp*.

- In der Klasse *CanonField* werden einige Methoden und ein Attribut hinzugefügt:



Hinweise: Die Methode *QRect canonRect()* wird in einem späteren Kapitel benötigt und gibt den Zeichenbereich der Kanone zurück:

```
1  QRect CanonField::canonRect() const
2  {
3    // Rechteck mit der Größe des Zeichenbereichs für die Kanone (inkl.
       Kanonenrohr) erzeugen
4      QRect result(0,0,70,70);
5    // Rechteckkoordinaten auf den linken unteren Punkt des Zeichenbereichs
       legen
6      result.moveBottomLeft(rect().bottomLeft());
7    // Rechteck zurück geben
8      return result;
9  }
```

Die Methode ist privat deklariert, da sie nur für klasseninterne Zwecke gedacht ist.

10 „Der erste Schuss“

Thema: Nutzung eines Timer (*QTimer*) für Animationen

1. Leeres Qt4 Projekt mit Namen „overthewall“ erstellen
2. Fügen Sie, wie im letzten Projekt beschrieben, die C++ Klasse **class** CannonField : **public QWidget** mit den beiden Dateien *canonfield.h* und *canonfield.cpp* zum Projekt hinzu:
3. Da der im letzten Projekt erstellte Quellcode in diesem Projekt sinnvoll nutzbar ist, kopieren Sie bitte die folgenden Dateien in den neuen Projektordner „overthewall“
   - „main.cpp“
   - „lcdrange.h“
   - „lcdrange.cpp“
4. Wie Sie im QtCreator erkennen können, gehören die kopierten Dateien aber noch nicht zum neuen Projekt. Um die neuen Dateien dem Projekt hinzuzufügen, müssen Sie die Projektdatei „overthewall.pro“ im QtCreator öffnen und wie folgt modifizieren:

```
1 SOURCES += main.cpp \
2       lcdrange.cpp \
3    canonfield.cpp
4
5 HEADERS += lcdrange.h \
6    canonfield.h
```

Erklärungen

Die Projektdatei (immer mit der Endung „.pro“ bei Qt) beinhaltet alle Informationen zum Projekt, wie notwendige Module (mySQL, Netwerk-Modul, etc.) und

alle Quellcode- bzw. Headerdateien. Damit handelt es sich bei der Projektdatei um den zentralen Punkt eines Softwareprojektes. In Ihrer Projektdatei können Sie zwei Abschnitte erkennen:

- Unter dem Punkt „SOURCES“ in der Projektdatei werden alle Quellcodedateien aufgelistet. Dabei steht in jeder Zeile ein Dateiname. Alle Zeilen, bis auf die letzte Zeile dieses Abschnittes, enden mit einem Backslash.
- Unter dem Punkt „HEADERS“ in der Projektdatei werden alle Headerdateien aufgelistet. Auch hier steht in jeder Zeile ein Dateiname und jede Zeile, bis auf die letzte, werden mit einem Backslash abgeschlossen.

Das „+ =“ in den beiden Abschnitten bedeutet, dass die folgenden Dateien hinzugefügt werden. Dadurch ist es möglich, auch an anderen Stellen weitere Dateien hinzuzufügen ohne die ursprüngliche Liste von Dateien zu überschreiben, wie es bei einem einfachen „=“ sein würde.

Aufgabe

Modifizieren Sie den Quellcode der Datei „main.cpp“ wie folgt:

```
1  #include <QApplication>
2  #include <QtCore>
3  #include <QtGui>
4  #include "lcdrange.h"
5  #include "canonfield.h"
6
7  class MyWidget : public QWidget
8  {
9  public:
10   MyWidget(QWidget *parent = 0);
11 };
12
13 MyWidget::MyWidget(QWidget *parent)
14     : QWidget(parent)
15 {
16   setWindowTitle("Over the wall");
17
18   QPushButton *pbExit = new QPushButton(tr("Beenden"));
19   pbExit->setFont(QFont("Times", 18, QFont::Bold));
20
21   // Hier müssen noch *lcdRange und *canonField erstellt werden
22
23   QGridLayout *gridLayout = new QGridLayout;
24   gridLayout->addWidget(pbExit, 0, 0);
```

```
25    // Hier müssen noch lcdrange und canonField in das QGridLayout gepackt
          werden
26    gridLayout->setColumnStretch(1, 10);
27    setLayout(gridLayout);
28
29    // angle->setValue(60); // Startwert setzen (wird später implementiert)
30    // angle->setFocus();    // Tastaturfokus auf die Klasse LCDRange setzen
31
32    connect(pbExit, SIGNAL(clicked()), qApp, SLOT(quit()));
33  }
34
35  int main(int argc, char *argv[])
36  {
37    QApplication app(argc, argv);
38    MyWidget widget;
39    // Fenster an der Position (100,100) mit der Größe 500 x 355 anzeigen
40    widget.setGeometry(100, 100, 500, 355);
41    widget.show();
42    return app.exec();
43  }
```

Compilieren und testen Sie das Programm. Ihr Programmfenster sollte wie folgt aussehen:

Erklärung

Das im Quellcode verwendete *QGridLayout* wird verwendet, um Layouts in Tabellenform zu erstellen. Dieses Layout ist wesentlich mächtiger als die beiden Layouts *QVBoxLayout* und *QHBoxLayout*. Momentan haben wir nur einen *QPushButton*, so dass das Tabellenlayout (noch) wenig Sinn macht. Es werden aber weitere Elemente folgen.
Beim *QGridLayout* müssen keine Abmessungen, sondern nur Tabellenzellen angegeben werden. Die Breiten bzw. Höhen der einzelnen Spalten (engl. column) bzw. Reihen (engl. row) der Tabelle werden entsprechend der Inhalte ausgerichtet.
Für das *QGridLayout* erwartet der Befehl `gridLayout->addWidget(quit, 0, 0)` drei Argumente:

1. Das Widget, welches in eine Tabellenzelle gepackt werden soll
2. Die x-Koordinate der Tabellenzelle, in die das Widget soll
3. Die y-Koordinate der Tabellenzelle, in die das Widget soll

In der folgenden Grafik sehen Sie auf der linken Seite das angestrebte Layout für das Programmfenster. Auf der rechten Seite ist das Tabellenlayout mit Koordinatenbeschriftung gezeigt. So hat das linke obere Widget, der *QPushButton*, im Programm die Koordinaten (0,0). Das Widget *LCDRange* soll an der Koordinate (1,0) und das Widget *CanonField* über die Koordinaten (1,1) und (2,1) angezeigt werden.

Die Programmzeile

```
26      gridLayout->setColumnStretch(1, 10);
```

steuert das Verhalten des Fensters beim Verändern der Größe. Normalerweise versucht Qt eine Tabellenspalte immer mit der gleichen Breite darzustellen. Mit der Methode *setColumnStretch* kann dieses Verhalten geändert werden. Der erste Parameter der Methode gibt die entsprechende Spalte an. Der zweite Parameter gibt an, mit welchem Faktor diese Spalte bei einer Größenveränderung des Fensters mitwächst/-schrumpft. Genauso wie das Verhalten der Breiten aller Spalten über den Stretchfaktor angegeben werden kann, kann auch das Verhalten der Höhen festgelegt werden. Die entsprechende Methode lautet: *setRowStretch*.

Aufgabe

Ergänzen Sie den Quellcode so, dass im Programmfenster jeweilse ein Objekt der Klassen/Widgets *LCDRange* und *CanonField* dargestellt werden. Achtung: Das Widget *CanonField* hat noch keinen Inhalt. Daher ist dieses Widget auch noch nicht sichtbar im Programmfenster. Fügen Sie es aber denoch schon zum Layout hinzu.

Aufgabe

Nun soll es möglich werden, auf die Digitalanzeige *lcdrange* von *MyWidget* aus mehr Einfluss zu nehmen. Stellen Sie zunächst sicher, dass der Quellcode der Klasse wie folgt aussieht.

Datei „lcdrange.h“:

```
1  #ifndef LCDRANGE_H
2  #define LCDRANGE_H
3
4  #include <QtCore>
5  #include <QtGui>
6
7  class LCDRange : public QWidget
8  {
9    Q_OBJECT
10
11 public:
12   LCDRange(QWidget *parent = 0);
13
14 public slots:
15
16 signals:
17
18 protected:
19
20 private:
21
22 };
23
24 #endif // LCDRANGE_H
```

Beachten Sie, dass die Bereiche für Slots und Signale bereits vorbereitet sind. Das Schlüsselwort *protected* stellt eine dritte Möglichkeit zu *private* und *public* dar, welche für Vererbungsbeziehungen zwischen Vater-Kind-Klassen wichtig wird.
Das Makro Q_OBJECT muss im privaten Bereich der jeder Klassedeklaration stehen, welche Signale oder/und Slots nutzt.

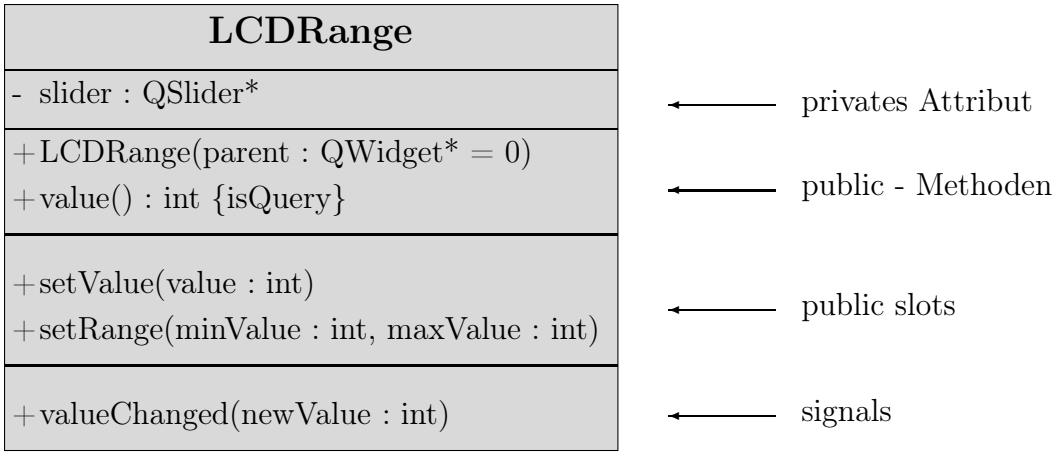
Datei „lcdrange.cpp“:

```
1  #include "lcdrange.h"
2
3  LCDRange::LCDRange(QWidget *parent)
4      : QWidget(parent)
5  {
6    QLCDNumber *lcd = new QLCDNumber(2);
7    lcd->setSegmentStyle(QLCDNumber::Filled);
8
9    // ACHTUNG: Das Attribut "slider" ist jetzt fest in der Headerdatei
         deklariert!!!
10   // Dies ist notwendig, da in verschiedenen Methoden dieser Klasse der
         Wert
11   // vom Slider abgefragt oder gesetzt werden muss!
```

```
12    slider = new QSlider(Qt::Horizontal);
13    slider->setRange(0, 99);
14    slider->setValue(0);
15
16    connect(slider, SIGNAL(valueChanged(int)), lcd, SLOT(display(int)));
17    connect(slider, SIGNAL(valueChanged(int)), this, SIGNAL(valueChanged(int)
          ));
18
19    QVBoxLayout *layout = new QVBoxLayout;
20    layout->addWidget(lcd);
21    layout->addWidget(slider);
22    setLayout(layout);
23
24    setFocusProxy(slider); // Den Tastaturfokus von dieser Klasse auf Slider
          setzen
25  }
26
27  void LCDRange::setRange(int minValue, int maxValue)
28  {
29    // Hier fehlt noch Code: Wenn der Bereich nicht zulässig ist, dann ...
30    qWarning("LCDRange::setRange(%d, %d) \n"
31         "\tBereich muss zwischen 0..99 liegen \n"
32         "\tund minValue darf nicht größer als maxValue sein!",
33         minValue, maxValue);
34    // Hier fehlt evtl. noch Code: Ansonsten setze den neuen Bereich ...
35  }
```

Beachten Sie die hinzugekommene Zeile 14. Diese Zeile bewirkt, dass das Klasseneigene (*this*) Signal *valueChanged(int)* gesendet wird sobald sich der Wert des Sliders (*slider*) ändert (*valueChanged(int)*). Dieses Signal muss aber noch zu der Klasse QLCDRange hinzugefügt werden. Dabei bekommt die Methode *valueChanged* keinen Quellcode, sondern muss nur als Methode in der Headerdatei deklariert werden.

Modifizieren bzw. ergänzen Sie die Klasse *LCDRange* wie im Klassendiagramm angegeben:



HINWEISE: Die Methoden *value()*, *setValue(int value)* sollten selbsterklärend sein. Die Methoden *setRange(int minValue, int maxValue)* setzt den Zahlenbereich des Sliders. *setValue(int value)* und *setRange(int minValue, int maxValue)* werden als *public slots* deklariert. Damit können diese beiden Methoden auch in einem *connect* Befehl als Slot verwendent werden. Wir werden das später ausnutzen. Die Methode *void valueChanged(int newValue)* wird als Signal deklariert. Das bedeutet, dass Sie dafür keine Methodendefinition in die Datei „lcdrange.cpp“ programmieren müssen. Die Methode wird nur in der Headerdatei im entsprechenden Bereich deklariert. Wir werden dieses Signal ebenfalls später verwenden.

Aufgabe

Zuletzt fehlt noch eine Ausgabe in unserem Widget *canonField*. Stellen Sie sicher, dass der Quellcode der Klasse dem Folgenden entspricht:

Datei „canonfield.h“:

```
1  #ifndef CANONFIELD_H
2  #define CANONFIELD_H
3
4  #include <QtGui>
5
6  class CanonField : public QWidget
7  {
8    Q_OBJECT
9
10 public:
11   CanonField(QWidget *parent = 0);
12   int angle() const { return currentAngle; }
13
14 public slots:
15   void setAngle(int angle);
16
17 protected:
18   void paintEvent(QPaintEvent *event);
19
20 private:
21   int currentAngle;
22 };
23
24 #endif
```

Erklärung

Die Methode `int angle() const { return currentAngle; }` ist eine sogenannte Inline-Methode. Dies bedeutet, dass sie in der Headerdatei komplett definiert (Quellcode ist komplett) und nicht nur deklariert (nur der Methodenkopf) ist. Die Methode soll den aktuell eingestellten Winkel, also den Wert des Sliders bzw. der LCDNumber, zurück liefern.

`void setAngle(int angle)` aus Zeile 15 ist ein Slot und wird später dazu genutzt den aktuellen Winkel einer Kanone im Widget einzustellen. Zunächst soll erstmal nur der eingestellte Winkel im Widget durch einen Zahlenwert dargestellt werden.

Das PaintEvent aus Zeile 21 ist eigentlich eine von *QWidget* geerbte Methode. Es handelt sich um einen sogenannten EventHandler, also um eine Methode, die sich um die Abarbeitung bestimmter Ereignisse kümmert. Die *QWidget* Methode wird automatisch immer dann aufgerufen, wenn das Fenster bzw. der Fensterbereich neu gezeichnet werden muss. Dies kann zum Beispiel dadurch passieren, dass sich die Größe des Fenster verändert hat oder das Fenster durch überdeckende andere Programme wieder sichtbar wird. Zudem kann ein PaintEvent auch manuel durch aufruf von `update()` angestoßen werden. Da es die PaintEvent Methode bereits in der Vaterklasse *QWidget* gibt, bräuchten wir uns eigentlich nicht darum zu kümmern. Allerdings soll in diesem Widget ja immer der aktuell eingestellte Abschusswinkel unserer Kanone als Zahlenwert sichtbar sein. Dies kann der Standard-PaintEvent nicht leisten. Daher müssen wir den PaintEvent überschreiben. Dies bedeutet, dass wir eine eigene Methode `void paintEvent(QPaintEvent *event)` programmieren müssen. Anstatt der QWidget-Methode wird dann in Zukunft immer unsere CanonField-Methode aufgerufen.

Schauen wir uns mal den Quellcode der Quelldatei *canonfield.cpp* an.

Datei „canonfield.cpp“:

```
1  #include <QtGui>
2  #include "canonfield.h"
3
4  CanonField::CanonField(QWidget *parent)
5      : QWidget(parent)
6  {
7      currentAngle = 45;
8      // Neue Farbpalette für das Widget setzen
9      setPalette(QPalette(QColor(250, 250, 200)));
10     // Hintergrund immer automatisch füllen
```

```
11    setAutoFillBackground(true);
12  }
13
14  void CanonField::setAngle(int angle)
15  {
16    if (angle < 5)   angle = 5;
17    if (angle > 70)  angle = 70;
18    if (currentAngle == angle)  return;
19    currentAngle = angle;
20    update(); // Hiermit wird ein PaintEvent angestossen
21  }
22
23  // Eigentlich sollte im Methodenkopf noch der Übergabeparameter stehen.
24  // Da dieser in der Methode aber nicht genutzt wird, wird er weg gelassen,
25  // um keine Warnung beim Compilieren auszulösen
26  void CanonField::paintEvent(QPaintEvent*)
27  {
28    // Hier wird ein "Zeichenstift" erstellt
29    QPainter painter(this);
30    // An die Position (200,200) einen Text schreiben
31    painter.drawText( /*HIER FEHLT NOCH ETWAS*/ );
32  }
```

Aufgabe

Vervollständigen Sie den Aufruf `painter.drawText(...)`. Es soll der aktuell eingestellte Winkel angezeigt werden.

Aufgabe

Zudem fehlt noch eine Verbindung, damit sich die Anzeige im Widget *canonField* tatsächlich aktualisieren kann. Fügen Sie die fehlende *connect* Verbindung zum Quellcode hinzu. Schauen Sie sich dazu die bestehenden Signale und Slots der verschiedenen Klassen an. TIP: Die Verbindung kann im Konstruktor der Klasse *MyWidget* hergestellt werden.

Aufgabe

Überprüfen Sie, ob der Slider mit den Pfeiltasten, Bild auf bzw. ab Tasten und Pos1 bzw. Ende Taste steuerbar ist.

Aufgabe

Ist der Slider auch noch über Tastatur steuerbar, wenn Sie die Zeile angle−>setFocus() in *MyWidget* auskommentieren?

Aufgabe

Testen Sie, was passiert, wenn Sie der linken Spalte im GridLayout einen Stretchfaktor ungleich Null geben?

Aufgabe

Was passiert, wenn Sie die Beschriftung des *Beenden* Buttons auf *Be&enden* ändern?

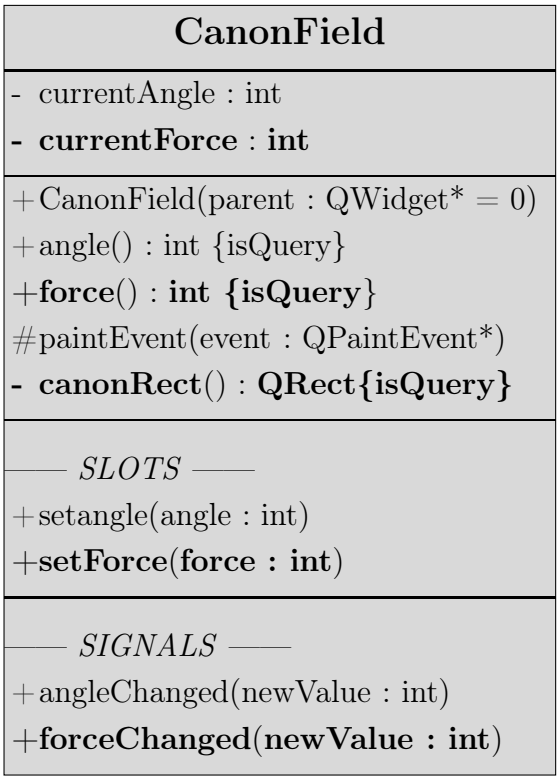
Aufgabe

Zentrieren Sie den Text im Widget *canonField*

11 „Schusskraft“

Thema: Zeichnen mit *QPaintEvent*
Das Programm soll nun durch eine Möglichkeit, die Schusskraft zu verändern, erweitert werden. Dazu ist folgendes zu programmieren:

- Dem grafischen Benutzerinterface (UI) wird ein weiteres Objekt *LCDNumber* hinzugefügt. Dieses zusätzliche Widget ist für das Einstellen der Schusskraft zuständig. Die Modifikation geschieht in der datei *main.cpp*.
- In der Klasse *CanonField* werden einige Methoden und ein Attribut hinzugefügt:



Hinweise: Die Methode *QRect canonRect()* wird in einem späteren Kapitel benötigt und gibt den Zeichenbereich der Kanone zurück:

```
1  QRect CanonField::canonRect() const
2  {
3    // Rechteck mit der Größe des Zeichenbereichs für die Kanone (inkl.
       Kanonenrohr) erzeugen
4    QRect result(0,0,70,70);
5    // Rechteckkoordinaten auf den linken unteren Punkt des Zeichenbereichs
       legen
6    result.moveBottomLeft(rect().bottomLeft());
7    // Rechteck zurück geben
8    return result;
9  }
```

Die Methode ist privat deklariert, da sie nur für klasseninterne Zwecke gedacht ist.