Qt Framework

Stepan Rutz

30.10.2023

Einführung in das Qt Framework

Was ist Qt?

QT ist ein C++ GUI-Framework

- Initial sehr stark mit KDE Desktop verbunden
- Qt vereinfacht die C++ Entwicklung
- Qt hat mächtige Features und Widgets
- Qt ist ein Ökosystem
- Qt ist Multi-Plattform und Cross-Plattform

Warum Qt verwenden?

Das beste Framework mit:

- 3D, 2D, Webengines, Device-Unterstützung, SQL . Multimedia, QML
- Base-Library
- Eigener Technologiezyklus + Guter Geschmack $>$ 20 Jahre Verlässlichkeit
- Momentum

Einführung in die Qt-Entwicklungsumgebung

- Installieren Sie Qt auf Ihrem Computer
- Erste Schritte mit Qt
- Erstellen Sie ein neues Qt-Projekt
- Entwickeln Sie Ihre erste Qt-Anwendung

Installieren Sie Qt auf Ihrem Computer

Opensource version

https://www.qt.io/download-qt-installer-oss

Commercial version

- Tools, Add-ons, especially on embedded devices.
- Statisches Linken
- Kein Opensource Zwang

Einführung in die Qt-Entwicklungsumgebung

`Qt Creator` ist die Entwicklungsumgebung.

Großer Effort von Qt und sehr leistungsfähige IDE.

IDE bedeutet, dass man seinen ganzen Tag in dem Tool verbringt

Wir entwickeln eine Desktop App mit Qt 6 in C++ (Desktop Apps können auch mit Python entwickelt werden)

Which way?

- Python vs. C++
- qmake vs. CMake
- QtGui vs QtQuick/QML
- Qt Creator vs. other Editors/IDEs
- Develop on Linux

Qt und C++

- Qt ist native C++ und nur hier bekommt man das volle Potential
- C++ selber ist erst in letzter Zeit besser geworden:
  C++11: Lambda, Typeinferenz/Auto, std::move, ranges, Tupel, SmartPointers, Initializerlists
  C++14: Lambda++, decltype
  C++17: Filesystem library, std::string_view, parallel algorithms
- Keine Garbage Collection
  Tipp bei Problemen: Valgrind
- Trotz Verbesserungen nur begrenzte Standard-Bibliothek

Qt to the rescue

Qt ersetzt durch sein Ökosystem viele Libraries
Qt ist **wie aus einem Guss**.

Erste Schritte mit Qt

RTM: https://doc.qt.io/

Wir werden nun eine einfache Anwendung programmieren. Das geschieht fast komplett durch den Designer.

Dazu bitte ein “Primer” Projekt anlegen.

Erstellen Sie ein neues Qt-Projekt

- Projekt anlegen
- Menü erzeugen
- Anwendung beenden

Reihenfolge der Include-Anweisungen

Es gibt 3 Arten von Headern.

1. Eigene Header aus dem Projekt (ein normalen " Anführungszeichen),
2. Header von Qt (starten alle mit Q)
3. Header aus der C oder C++ Standardbibliothek und weiteren Bibliotheken.

Reihenfolge der Include-Anweisungen

Folgende Regel kann man grob anwenden:

- Eigene Header zuerst. Das sorgt, dafür dass die eigenen Header nicht noch weitere Header benötigen, die nicht direkt inkludiert werden.
- Danach die QtHeader, auch hier werden Macros etc definiert die man nicht kontrolliert und welche die eigenen Header nicht beeinflussen sollen.
- Danach alle weiteren Header. Es gilt das gleiche wie für die Qt-Header. In einem Qt Projekt kommen die Qt-Header jedoch zuerst.

```
#include "MyCustomWidget.h" // Your own header
#include <QApplication>     // Qt header
#include <vector>           // Standard library header
```

Es gibt aber Ausnahmen und Gründe von dieser Empfehlung in Einzelfällen abzuweichen.

QObject, Teil 1

1. **Definition**:
   - `QObject` ist die Basisklasse für die meisten Qt Klassen
   - Stellt Kernfunktionen zur Verfügung
2. **Main Features**:
   - Ownership, durch QObject entstehen Parent/Child Verbindungen zwischen Objekten
   - Meta-object System (MOC): Signals &Sslots, Qt-RTTI (Runtime type information) and dynamic property management.

QObject macht Qt so leistungsfähig

QObject, Teil 2

1. **Signals**:
   - Mechanismus um Signale zu senden.
   - Deklariert mit dem `signals:` keyword. (kein natives C++)
2. **Slots**:
   - Funktionen die auf Signale reagieren
   - Können reguläre Member-Funktionen sein
   - Deklariert mittels `public slots:`, `protected slots:`, oder `private slots:`
3. **Connections**:
   - Eine Verbindung zwischen signal und slot.
   - Wird mittels `QObject::connect()` hergestellt (oder grafisch im Qt Creator)

Signale und Slots sind die Basis für Qt's Event-basierten Ansatz und Typesafe

QObject, Teil 3

- Slots und Signale können auch Parameter haben und sind hier ebenfalls typsicher
- Man kann Signale auch mit Lambdas (anonyme Codeblöcke) verbinden, ganz ohne Slots

```
QObject::connect(
    globalStatus,
    &GlobalStatus::statusKeyChanged,
    [=] {
        qInfo() << "status key has changed";
    }
);
```

QString, Teil 1

1. **Definition**:
   - QString, hat Unicode Support (QChar = 16 Bit Unicode intern)
   - Zentrale Klasse für Text in Qt
2. **Main Features**:
   - Copy-on-write mit Referenzzähler. Daten werden extrem schnell kopiert und erst bei Modifikation kopiert.
   - Vergleichbar mit std::wstring aber mit viel mehr Funktionen.

QString, Teil 2

1. **Erzeugung von QStrings**:
   - Can be constructed from literals, numbers, or other strings.
   - QString str = “Hello, World!”;
2. **Manipulation**:
   - append(), prepend(), remove(), replace(), etc.
   - Für einzelne Zeichen oder Strings (via C++ Operator-Overloading)
3. **Abfragen**:
   - length(), isEmpty(), at(index), contains(), startsWith(), endsWith(), etc.

QString, Teil 3

Beispiel: Aufteilen eines QString mittels split():

```
QStringList sentences = url.split(".");
```

oder

```
QString q = "one. two three.. four";
auto sentences = q.split(".", Qt::SkipEmptyParts);
for (const auto& sentence : sentences) {
   qInfo() << sentence.trimmed();
}
```

QString, Teil 4: Konvertierungen

1. **QByteArray**:
   - toUtf8(), toLatin1(), fromUtf8(), fromLatin1(), etc.
2. **Zahlen**:
   - setNum(), number(), toInt(), toDouble(), etc.
3. **Interaktion mit der STL**:
   - Hin und zurück von/zu std::string, std::wstring

QString, Teil 5

Zusammensetzen von QStrings

```
int age = 25;
QString name = "John";
QString result = QString("My name is %1 and I am %2 years o
  .arg(name).arg(age);

QString first = "Hello, ";
QString last = "world!";
QString result = first + last;  // "Hello, world!"

int age = 25;
QString result = QString::asprintf(
  "I am %d years old.", age);
```

Qt Misc

- QRegularExpression
- QtConcurrent
- QtNetwork
- QtOpenGL
- QtPng, QtJpeg
- QtSvg
- QtSql
- QtXml
- QSettings

QRegularExpression

```
#include <QRegularExpression>

/* finds URLs in einem Test (ungenau) */
void findUrls(const QString &text) {
    QRegularExpression re("(https?://\\S+)");
    auto it = re.globalMatch(text);
    while (it.hasNext()) {
        QRegularExpressionMatch match = it.next();
        qInfo() << "Found URL:" << match.captured(0);
    }
}
int main() {
    QString text = "Visit https://www.qt.io or "
    "https://www.google.de for more information.";
    findUrls(text);
    return 0;
}
```

QSettings

Mittels QSettings kann man folgende Einstellungsdateien verarbeiten:

1. **QSettings::IniFormat** für .ini Files
2. **QSettings::NativeFormat** für Registry / .pfiles / .txt

Desweiteren kann man noch zwischen 32bit und 64bit Registry Formaten wählen

Container Klassen

1. **Collections** QVector, QSet, QVarLengthArray und QList
2. **QBitArray** Bitset
3. **Maps/Dictionaries** QMap, QHash, QMultiMap, QMultiHash

Pointers

- QPointer (Guarded Pointer, wird intern “null” wenn das QObject gelöscht wird)
- QSharedPointer, QWeakPointer

QWeakPointer hat toStringRef() und ist somit auch ein GuardedPointer

QSharedPointer

```
class TestObject {
public:
    TestObject() {
        qDebug() << "TestObject()";
    }

    ~TestObject() {
        qDebug() << "~TestObject()";
    }

    void sayHello() const {
        qDebug() << "Hello from TestObject";
    }
};

QSharedPointer<TestObject> ptr1(new TestObject());
ptr1->sayHello();
```

Valgrind

QMap

```
QStringList words = { "one", "two", "one",
  "one", "two", "three" };
QMap<QString, int> wordCount;
for (const QString &word : words) {
    QString lowerWord = word.toLower();
    wordCount[lowerWord]++;
}
QMapIterator<QString, int> i(wordCount);
while (i.hasNext()) {
  i.next();
  qInfo() << i.key() << ": " << i.value();
}
```

Entwickeln Sie Ihre erste Qt-Anwendung

- PDF Viewer
- PDFs auswählen und anzeigen

1. Gui Gerüst
2. TextWidget + PDFWidget
3. Splitter + Resizing
4. Dateien laden
5. Widgets finden und ansprechen

QObjects und Signals & Slots

- QObject
- Speicherverwaltung
- Reentrant (keine gemeinsamen Globals
- vs. Threadsafe (kein eigenens Locking notwending)

QObject Properties, Teil 1

- **Eigenschaftsname:** Eindeutiger Name für die Eigenschaft.
- **Lese-Methode:** Methode, um den Wert der Eigenschaft abzurufen.
- **Schreib-Methode:** Methode, um den Wert der Eigenschaft zu setzen.
- **Benachrichtigungs-Signal:** Ein Signal, das ausgelöst wird, wenn sich der Wert der Eigenschaft ändert.

```
myObject->setProperty("volume", QVariant(200));
QVariant value = myObject->property("volume");
```

QObject Properties, Teil 2

- **Automatisches Binding**
- **Benachrichtigungsmechanismus**
- **Integration mit dem Qt Designer** QProperties können direkt im Qt Designer visualisiert und bearbeitet werden, wodurch die Entwicklung von Benutzeroberflächen erleichtert wird.
  Letzteres via Introspection:

```
QMetaProperty = myObject->metaObject()->property(2)
```

IO + Networking

Streams und Files

- QFile
- QTextStream
- QDataStream

Network immer Async

- NetworkAccessManager
- Libraries (siehe spätere Folie)

GUI

GUIs / Interfaces sind die Stärke von QT.

Die zentralen Konzepte sind

- **Widgets (QWidget)**
- **Basic Widgets**
- **Layouts**
- **Stacked-Widget / TabWidget**
- **Tables and Treeviews**

QWidget

Was ist ein QWidget?

- Grundlegende Klasse für alle UI-Objekte in Qt.
- Kann als Fenster oder ein Steuerelement innerhalb eines Fensters verwendet werden.
- ist von QObject abgeleitet

Hauptmerkmale

1. **Ereignisverarbeitung**: Bearbeitet UI-bezogene Ereignisse wie Mausklicks, Tastendrücke usw.
2. **Geometrie-Management**: Festlegen und Abrufen von Größe, Position und anderen geometrischen Eigenschaften.
3. **Layout-System**: Ermöglicht das organisierte Anordnen von UI-Elementen innerhalb eines Widgets.
4. **Owner-Draw**: Möglichkeit selber zu Zeichnen.

Die Eventloop

Was ist die Qt Event Loop?

Die Event Loop ist ein Kernmechanismus des Qt-Frameworks, der für das Abfragen und das Verarbeiten von Ereignissen und das Weiterleiten dieser an die zuständigen Objekte verantwortlich ist.

Event Loop Kernkomponenten

- **Ereignisse (Events):** Aktionen, die wie Mausklicks oder Tastendrücke als Reaktion auf Benutzerinteraktionen auftreten können.
- **Event Queue:** Eine Warteschlange, in der Ereignisse auf ihre Verarbeitung warten.
- **Event Loop:** Ein kontinuierlicher Zyklus, der Ereignisse aus der Queue entnimmt und sie behandelt.

Funktion

```
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    return app.exec(); // Startet die Event Loop
}
```

Eventloop Verwendung

Verwendung

- Benutzerdefinierte Ereignisse: Können erstellt und in die Queue eingefügt werden.
- Timer-Ereignisse: Werden für periodische Aktionen genutzt.
- Überschreiben von Event-Handlern: Ermöglicht benutzerdefinierte Reaktionen auf Ereignisse.

Layouts und Stile

- HBox und VBox
- GridLayout
- FormLayout
- Splitters

Widget Größe

- QWidget Properties: pos und size
- QWidget::sizePolicy() legt Verhalten in Layouts fest

QVBoxLayout

- Eine der häufigsten Layout-Klassen in Qt.
- Stapelt Widgets vertikal.
- Einfach zu verwenden und flexibel.

QVBoxLayout - Anwendung

```
QWidget *container = new QWidget();
QVBoxLayout *layout = new QVBoxLayout();

layout->addWidget(new QPushButton("Oben"));
layout->addWidget(new QPushButton("Mitte"));
layout->addWidget(new QPushButton("Unten"));

container->setLayout(layout);
container->show();
```

QHBoxLayout - Anwendung

```
QWidget *container = new QWidget();
QHBoxLayout *layout = new QHBoxLayout();

layout->addWidget(new QPushButton("Links"));
layout->addWidget(new QPushButton("Mitte"));
layout->addWidget(new QPushButton("Rechts"));

container->setLayout(layout);
container->show();
```

QSplitter

- Ermöglicht das dynamische Anpassen von Widget-Größen.
- Ideal, um UI-Elementen variable Raumverteilung zu ermöglichen.
- Kann sowohl horizontal als auch vertikal verwendet werden.

QSplitter - Anwendung

```
QSplitter *splitter = new QSplitter(Qt::Horizontal);
splitter->addWidget(new QTextEdit());
splitter->addWidget(new QListView());
splitter->show();
```

QFormLayout

- Für formularartige Widgets.
- Pairs aus Labels und Feld-Widgets.
- Automatisches Anordnen von Widgets.

QFormLayout - Anwendung

```
QFormLayout *layout = new QFormLayout();
layout->addRow("Name:", new QLineEdit());
layout->addRow("Alter:", new QSpinBox());

QWidget *container = new QWidget();
container->setLayout(layout);
container->show();
```

QGridLayout

- Widgets werden in einem Raster angeordnet.
- Maximale Flexibilität für die Anordnung.
- Definiert durch Zeilen und Spalten.

QGridLayout - Anwendung

```
QGridLayout *layout = new QGridLayout();

layout->addWidget(new QPushButton("1"), 0, 0);
layout->addWidget(new QPushButton("2"), 0, 1);
layout->addWidget(new QPushButton("3"), 1, 0);
layout->addWidget(new QPushButton("4"), 1, 1);

QWidget *container = new QWidget();
container->setLayout(layout);
container->show();
```

Widgetentwicklung

- Eigene Widgets erstellen
- Eigenes Zeichnen
- In den Creator einbinden

Fortgeschrittene Qt-Themen

- **Verwendung von Datenbanken mit Qt**
- **Verbindungen zu externen APIs**
- **Erweiterung von Qt mit Plugins**
- **Benutzeroberflächen- und Interaktionsdesign**

Verwendung von Datenbanken mit Qt

QSql ist eine vollständige SQL-API.

Datenbanken Teil 1

Synopsis Selects:

```
QString sql =
"select email, name from users where email like ?";
{
    QSqlQuery query;
    query.prepare(sql);
    query.bindValue(0, QVariant("%hans%"));
    if (query.exec()) {
        while (query.next()) {
            auto email = query.value("email").toString();
            auto name = query.value("name").toString();
            // ....
        }
    } else {
        qDebug() << query.lastError().text();
    }
}
```

Datenbanken Teil 2

Synopsis Updates:

```
QString sql =
"insert into users(email,name) values (:email, :name)";
{
    QSqlQuery query;
    query.prepare(sql);
    query.bindValue(":email",
        "willi.millowitsch@cologne.de");
    query.bindValue(":name", "Willi Millowitsch");
    if (query.exec()) {
        qInfo().nospace() << "inserted #"
            << query.numRowsAffected() << " row(s)";
    } else {
        qDebug() << query.lastError().text();
    }
}
```

Datenbanken Teil 3

Anzeige mittles **QSqlQueryModel**

```
// Header
class ResultsView : public QTableView
{
    // ...
    void showQuery(QSqlQuery &&query);
    // ...
};

// Impl
void ResultsView::showQuery(QSqlQuery &&query) {
    QSqlQueryModel *model =
       dynamic_cast<QSqlQueryModel*>(this->model());
    model->setQuery(std::move(query));
}
```

Verbindungen zu externen APIs

Achtung: Immer Async

- Mittels QNetworkManager
- Mittels libcurl
- CPR/C++Requests
- cpp-httplib

Erweiterung von Qt mit Plugins

- QtCreator kann Plugins aufnehmen (nicht nur Widgets)
- QtCreator kann Customwidgets im Designer anzeigen und verwenden
- Man kann seine eigenen Widgets QtCreator-Ready machen
- Fallback: Ist promoteTo, dann gibt ea aber keine Properties und kein Preview

Benutzeroberflächen- und Interaktionsdesign

- Qt unterstützt die Themes des nativen Betriebssystem
- Qt unterstützt Stylesheets auf Widgetbasis
- Qt Creator selber kann ergo auch “Themed” werden

Veröffentlichung Ihrer Qt-Anwendung

Übersicht: Schritte zur Auslieferung

1. **Build Prozess**
   - Release-Konfiguration erstellen
   - Abhängigkeiten verwalten
2. **Erstellung des Installers**
   - Qt Installer Framework nutzen
   - Installationsroutinen definieren
3. **Testen des Installers**
   - Installationsprozess überprüfen
   - Funktionsprüfung nach der Installation
4. **Distribution**
   - Auswahl der Plattform(en)
   - Bereitstellungsmöglichkeiten

Build Prozess

- **Release Konfiguration**: `qmake` mit `CONFIG+=release` verwenden.
- **Statische vs. Dynamische Links**: Statische Builds beinhalten alle notwendigen Bibliotheken.
- **Verwendung von windeployqt**:

```
windeployqt --release --no-translations <Pfad-zum-App-Verze
```

Erstellen Sie ein Installationsprogramm

```
binarycreator --config config.xml --packages <Pfad-zu-Paket
```

Zusammenfassung und Fragen

Um Qt zu lernen muss man folgendes beherrschen:

- C++ Grundlagen
- Die Qt Bibliotheken (insbesondere Core)
- Qt Creator

Zusammenfassung der vermittelten Inhalte

- Qt Grundlagen aus QtCore
- Qt Widgets
- Qt Sql
- Qt QWT

Beantwortung offener Fragen

Any questions?