Qt Framework

Stepan Rutz

01.07.2024

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 Unterstützung für:

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

Opensource version

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

- Der Installer benötigt einen Account bei Qt.
- Das Maintenance Tool ist Teil des Installers und wird für Updates und für das Nachinstallieren von Komponenten benötigt.

Commercial version

- Tools, Add-ons, besonders für Embedded-Devices.
- Statisches Linken
- Kein Opensource-Zwang

Dokumentation und weitere Infos

Canonical

https://doc.qt.io/

Empfehlungen

- C++ GUI Programming with Qt 4 (2nd Edition) - The official C++/Qt book
- https://github.com/mikeroyal/Qt-Guide
- Kurs zu QtWidgets von KDAB

Weitere Bücher zu Qt

- Qt5 C++ GUI Programming Cookbook - Second Edition

Bücher und Links zu C++

- Die C++ Programmiersprache von Bjarne Stroustrup
- Effective Modern C++ by Scott Meyers
- https://cplusplus.com/
- Fluent C++
- Cpp Con Videos (Empfehlung: Alle Back to Basics Videos)

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
- CoPilot muss als Plugin aktiviert werden und verwendet Vim + Node.js Infrastruktur

Man kann auch Vim, Neovim, Emacs, VisualStudio-Code, Visualstudio verwenden.

Which way?

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

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
  C++20: std::range, new literals
- Keine Garbage Collection
  Tipp bei Problemen: Valgrind
- Trotz Verbesserungen nur begrenzte Standard-Bibliothek

Achtung: Qt hat eigene Implementierungen wie QVector, QSharedPointer etc und ist gleichzeitig stark interoperabel mit Standard C++.

Qt to the rescue

Qt ersetzt durch sein Ökosystem viele Libraries Qt ist **wie aus einem Guss**. Qt ist einfach für (Old-School) Java-Entwickler zu erlernen.

Obwohl Standard C++ aufgeholt hat, braucht man Bibliotheken in jedem Projekt. Bei Qt braucht man in der Regel kaum externe Bibliotheken.

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

C++17

Qt verwendet C++17, wobei C++20 möglich sein sollte.
Interessante Konstrukte/Idiome aus C++17 die in Qt genutzt werden sind:

- C++ Attribute [[nodiscard]]
- noexcept Operator (seit C++11)
- auto (C++11)
- constexpr (C++11)
- decltype (C++11)
- Neue APIs: chrono, random, regex, thread, tuple, array, future, chrono
- Initialisierer Ausdrücke:

```
struct P { int x; int y; };
auto p = P{
   .x = 10,
   .y = 30
};
auto a1 = std::array<int,2>{1, 2};
```

Tools

- GDB (Gnu Debugger), unterstützt C++ und auch remote Debugging
- KDiff3, alternative zu Meld und unterstützt 3-Wege Merge und ist ein gutes Beispiel für ein Qt-Programm
- Valgrind, Memory Leak Detection und finden von Speicherzugriffsfehlern

Perf benötigt root und sampled (wenig overhead) Valgrind benötigt kein root und instrumentiert (sehr viel overhead)

```
valgrind --leak-check=full --track-origins=yes --verbose --
```

C++ Exceptions und QtExceptions

In Qt macht es Sinn die QtException zu subclassen oder bestehenden QtExceptions zu verwenden. Exceptions sind nicht leichtgewichtig, noch leicht zu verstehen. Verwendung in Ausnahmefällen. Zum Resource-Cleanup ist anstelle RAII zu verwenden.

Merksätze zu QtExceptions. - Doku-Zitat: “QException subclasses must be thrown by value and caught by reference” - Nicht für fehlende Werte verwenden, sondern std::optional anstelle verwenden.

Guter CppCon-Talk zum Thema: Cpp Exceptions

Reihenfolge der Include-Anweisungen

Es gibt 3 Arten von Headern.

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

Header mit spitzen Klammern werden in den Include-Paths gesucht,
Header in Anführungszeichen relativ zum Arbeitsverzeichnis.
Absolute Header könnten theoretisch auch in “” angegeben werden.

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 durch die Header selber 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 vor den System Headern.

```
#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.

Exkurs . . . QDebug und QElapsedTimer

Debugausgabe mit qDebug (qCritical, qFatal, qWarning, qInfo, qDebug). Diese Fumktionen sind Teil des Qt-Debugging-Frameworks und können auch in Release-Builds verwendet werden. Der Header ist nicht notwendig, wenn man inkludiert.

```
qDebug() << "Hello, World!";
```

qDebug() kann konfiguriert werden. Die wichtigsten Möglichkeiten sind:

```
// log format for qDebug
qSetMessagePattern("%{time yyyy-MM-dd hh:mm:ss.zzz} %{file
```

Formatoptionen:

```
// no-spaces, alignment, field widths
qDebug() << noquote << nospace << 42 << 3.14;
qDebug() << qSetFieldWidth(10) << 42 << 3.14;
qDebug() << qSetFieldWidth(10) << qSetPadChar('0') << 42 <<
qDebug() << qSetFieldWidth(10) << qSetFieldAlignment(QTextS
```

Zeiten messen mit QElapsedTimer

Microbenchmarks sind oft einfach zu schreiben. QElapsedTimer kann für die Zeitmessung verwendet werden. Die Zeitmessung ist nicht für extrem kurze Zeitspannen genau genug.

```
QElapsedTimer timer;
timer.start();
// do something
qDebug() << "Elapsed time:" << timer.elapsed() << "ms";

// pure C++ Alternative mit std::chrono

auto start = std::chrono::high_resolution_clock::now();
// do something
auto end = std::chrono::high_resolution_clock::now();
// not using auto here to show the type
chrono::duration<double, std::milli> duration = end - start
double durationMs = duration.count();
```

QObject als Basisklasse (oder nicht)

Klassen, die von QObject erben QWidget, QApplication, QTimer, QThread, QNetworkAccessManager, QPushButton, QLabel, QMainWindow, QDialog, QFile, QPainter, QLineEdit, QComboBox, QCheckBox, QVBoxLayout, QHBoxLayout, QStackedWidget, QTableWidget, QListWidget, QTextEdit, QScrollArea, QSplitter, QMenu ....

Werteklassen, die nicht von QObject erben QString, QVector, QList, QMap, QPair, QPoint, QSize, QRect, QColor, QRegExp, QVariant, QImage, QPixmap, QPolygon, QFont, QBrush, QPen, QMatrix, QTransform, QMargins, QByteArray, QBitArray, QFlags, QDate, QTime

Wichtige Konzepte: Ownership und Verbot des Kopierens (für QObject). COW (Copy-On-Write) für QString, QByteArray, QVector.

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

- QObject kann *nicht* kopiert werden. (Der Copy-Constructor ist private und/oder mit =delete markiert)
- In C++ findet die Copy-Initialization (=der Aufruf der Copy-Constructors) in folgenden Situation statt
  - Ein Initialisierer steht rechts von einem Gleichheitszeichen bei einer Objekterzeugung (Achtung, das ist *kein* Assignment)
  - Wenn ein Objekt beim Funktionsaufruf (Lamdas ebenso) als Wert (Pass by Value) übergeben wird.
  - Wenn ein Objekt als Wert (Return by Value) aus einer Funktion zurückgegeben wird.
  - Wenn eine Exception mit einem Objekt geworfen wird.
  - Wenn eine Exception mit einem Objekt gefangen wird. (Hier Referenzen verwenden)
  - Wenn ein Lamdba ein Objekt mit dem Wert captured. (via [=] oder expliziter Auflistung)

QObject, Teil 3

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 4

- 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";
    }
);
```

Hier ist es übrigens am besten noch ein “Ownership”-Objekt als 3. Parameter anzugeben.

Eventloop

Die Qt-Eventloop wird in den Qt-Threads gestartet und verarbeitet jeweils den nächsten Event sequentiell. Events können auf die Eventloop gepostet werden mittels

- QTimer::singleShot (Bevorzugt)
- QCoreApplication::postEvent (Low Level API und Custom-Events müssen definiert werden).

Für QTimer::singleShot kann man sehr gut C++ Lambdas verwenden.

```
void myDeleteLater(QObject *o) {
     QTimer::singleShot(0, o, [o]() { delete o; });
}
```

QString, Teil 1

1. **Definition**:
   - QString, hat Unicode Support (QChar = 16 Bit Unicode intern)
   - Zentrale Klasse für Text in Qt
   - Es gibt auch QLatin1String (sehr unterschiedlich zu QString)
2. **Main Features**:
   - Copy-on-write mit Referenzzähler. Daten werden durch Referenzzähler logisch kopiert und erst bei Modifikation physikalisch kopiert.
   - Vergleichbar mit std::wstring aber mit mehr Funktionalität.

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

Wichtig: QString("MY_STRING_LITERAL") ist flawed und bedeutet in der Regel, dass der Programmierer sich keine Gedanken über das Encoding gemacht hat.

Das Makro #define QT_NO_CAST_FROM_ASCII 1 verhindert eine direkte Konvertierung mittels QString s = "abc";

QString, Teil 4: Konvertierungen Datetime

String zu Datetime:

```
QString s = "2016-05-04 12:24:00";
QDateTime dateTime = QDateTime::fromString(s, "yyyy-MM-dd h
```

Datetime zu string:

```
QDateTime dateTime = QDateTime::currentDateTime();
QString s = dateTime.toString("dd/MM/yy hh:mm");
```

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);
```

QString, Teil 6

Man kann auch vor dem Zusammenführen die Kapazität setzen . . . . mittels QString::reserve) Dazu passend gibt es QString::squeeze() und QString::capacity(). . . QString::resize() ist jedoch eine semantisch-sichtbare Option.

`QStringBuilder` ist potentiell performanter bei größeren String-Konkatenationen und erzeugt auch keine temporären Objekte.

Merksatz. . . Wenn man einen QString in einer Schleife mittels += immer wieder erweitert, dann ist QStringBuilder in der Regel besser.

QByteArray

QByteArray ist oft eine bessere Alternative zu char* oder std::vector für Binärdaten (also alle Daten ;-). Moderner Standard wäre std::vector data;

Beispiel: Chunking (Aufteilen in Batches der Größe n)

```
size_t chunkSize = 10;
std::vector<std::vector<char>> chunks;
for (size_t i = 0; i < data.size(); i += chunkSize) {
    chunks.emplace_back(data.begin() + i, data.begin() + st
}
```

(Achtung: Der std::partition Algorithmus ist fürs interne Reordering und nicht fürs "Chunking")

```
auto chunkSize = 10;
QVector<QByteArray> chunks;
for (auto i = 0; i < data.size(); i += chunkSize) {
   chunks.append(data.mid(i, chunkSize));
}
```

QByteArray und Base64

QByteArray ist auch die richtige Klasse um zu Base64 zu konvertieren (toBase64) und von Base64 einzulesen (fromBase64).

Der Base64 Input/Output ist ebenfalls vom Typ QByteArray (Designflaw?) Es gibt Base64 Encoding-Optionen (QByteArray::Base64Encoding, Base64UrlEncoding, QByteArray::OmitTrailingEquals)

QTextStream

Um Textdaten inkl. Konvertierung zu speichern und zu laden.

```
QFile file("example.txt");
if (file.open(QIODevice::WriteOnly | QIODevice::Text)) {
    QTextStream s(&file);
    out << "Hello World" << endl;
}
```

QTextStream nicht für Binärdaten verwenden. Außerdem verwendet QTextStream locale-aware Zahlen- und Datumsformate. Sehr oft will man das nicht. In jedem Fall sollte man das Encoding mittels

```
QTextStream s(&file);
s.setCodec("UTF-8");
```

setzen.

QByteArray für binäres Laden und Speichern

```
QByteArray loadBinary(const QString& filePath) {
    QByteArray data;
    QFile file(filePath);
    if (!file.open(QIODevice::ReadOnly | QIODevice::Binary)
        throw MyFileException(QString(
         "Failed to open file for writing: %1").arg(filePath
    }
    data = file.readAll();
    file.close(); // aufgrund von RAII nicht notwendig
    return data;
}
```

QException und Custom Exceptions

QException ist (natürlich) kein QObject, aber ähnlich zu Java-Exceptions.

```
class MyFileException : public QException {
public:
    void raise() const override { throw *this; }
    FileIOException *clone() const override { return new Fi
    QString message;
    FileIOException(const QString& msg) : message(msg) {}
    QString what() const { return message; }
};
```

QVariant

QVariant erlaubt es Basistypen, Pointer, null und eigene QVariant-Implementierungen in einem Rückgabetyp zu implementieren. Die QVariant ist also ein Container/Wrapper um den eigentlichen Wert. Mittels Methoden wie toInt(), toPoint(), toRect() ....

Ob die Konvertierung möglich ist, kann zur Laufzeit mittels canConvert() abgefragt werden.

```
template <typename T> bool QVariant::canConvert() const
```

Man kann auch eigene Typen als QVariant registrieren

```
struct MyStruct { int x; double z; };
// global
Q_DECLARE_METATYPE(MyStruct)
// in main
qRegisterMetaType<MyStruct>("MyStruct");
```

Für reine Value-Types reicht das default-Verhalten aus (Rule of Zero-Types)

QVariant (Konvertierungsfunktionen)

Globale Definitionen für die Unterstützung von QVariant:

```
struct MyStruct { int x; double z; };
Q_DECLARE_METATYPE(MyStruct)
// serialize
QDataStream &operator<<(QDataStream &out, const MyStruct &m
    out << myStruct.x << myStruct.y;
    return out;
}
// deserialize
QDataStream &operator>>(QDataStream &in, MyStruct &myStruct
    in >> myStruct.x >> myStruct.y;
    return in;
}
```

QSettings

QSettings unterstützt mehrere Anwendungen die auf der gleichen Datei arbeiten nicht. Man muss FIlelocking oder separate Dateien verwenden. QSettings kann optional in einer Datenbank persistiert werden.

QSettings verwendet QVariant::fromValue(nullptr) um nicht existierende Werte zu signalisieren.

QCommandLineParser

Kommandozeile Parsing mit Optionen, Flags etc

```
#include <QCoreApplication>
#include <QCommandLineParser>
#include <QDebug>

int main(int argc, char *argv[]) {
    QCoreApplication app(argc, argv);
    QCommandLineParser parser;
    parser.setApplicationDescription("Example of QCommandLi
    parser.addHelpOption();
    parser.addVersionOption();
    QCommandLineOption showVersionOption(QStringList() << "
    parser.addOption(showVersionOption);
    parser.process(app);
    if (parser.isSet(showVersionOption)) {
        qDebug() << "Version 1.0.0";
        return 0;
    }
```

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();
```

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();
}
```

QList and Qt Algorithms

QList ist keine Linked-List und ist als Array von Pointern implementiert. In der Regel möchte man QVector or std::vector benutzen. QList hat eine weitere Ebene der Indirektion.

ttps://doc.qt.io/qt-6/qtalgorithms.html

Entwickeln Sie Ihre erste Qt-Anwendung

PDF Viewer und der Möglickeit 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

Definition einer Property “age” in QT mittels Makro

```
class MyClass : public QObject {
    Q_OBJECT
    Q_PROPERTY(int age READ getAge WRITE setAge NOTIFY ageC
```

- **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.

Properties können generisch gesetzt werden.

```
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)
```

QObject Properties, Teil 3

Mittels des metaObject() kann man auf die Methoden und Properties eines Objekts zur Laufzeit zugreifen. Properties können generisch via QVariant modifiziert werden. Das funktioniert mittels Property-Name oder Property-Index.

```
QMetaObject *metaObject = myObject->metaObject();
int propertyIndex = metaObject->indexOfProperty("age");
if (propertyIndex != -1) {
    QMetaProperty property = metaObject->property(propertyI
    property.write(myObject, 200);
}
```

QPointer, QScopedPointer, QSharedPointer, QWeakPointer

Diese “Smart”-Pointer vermeiden Memory-Leaks oder Double-Deletes. In der Regel ist die Benutzung von QSharedPointer zu bevorzugen. Dagegen sprechen höchstens Performance-Gründe oder das die Notwendigkeit von Smart-Pointern im konkreten Fall nicht gegeben ist.

QPointer

QPointer ist ein “guarded pointer” und wird automatisch auf nullptr gesetzt, wenn das QObject gelöscht wird.

```
QObject *obj = new QObject();
QPointer<QObject> ptr = obj;
delete obj;
if (ptr.isNull()) {
    qDebug() << "Object was deleted";
}
```

QScopedPointer

QScopedPointer ist ein “unique pointer” und wird automatisch gelöscht, wenn der Scope verlassen wird.

```
{
    QScopedPointer<QObject> ptr(new QObject());
    ptr->setObjectName("MyObject");
} // ptr ruft an dieser Stelle automatisch delete auf das
```

QSharedPointer

QSharedPointer ist ein “shared pointer” und wird automatisch gelöscht, wenn der letzte shared pointer auf das Objekt gelöscht wird.

```
{
    QSharedPointer<QObject> ptr1(new QObject());
    {

        QSharedPointer<QObject> ptr2 = ptr1; //      refcou
        QVector<QSharedPointer<QObject>> vec;
        vec.append(ptr1); // refcount = 3
        vec.append(ptr2); // refcount = 4
    } // refcount = 1
} // refcount = 0, Objekt wird gelöscht
```

QWeakPointer

QWeakPointer ist ein “weak pointer” und wird automatisch auf nullptr gesetzt, wenn das QObject gelöscht wird. Das ist die sichere Variante von QPointer.

```
{
    QSharedPointer<QObject> ptr(new QObject());
    QWeakPointer<QObject> weakPtr = ptr;
    ptr.clear();
    if (weakPtr.isNull()) {
        qDebug() << "Object was deleted";
    }
}
```

IO + Networking

Streams und Files

- QFile
- QTextStream
- QDataStream

QFile

QFile hat RAII und ist ein Wrapper um die C-File-Operationen.
QFile ist auch ein QIODevice und kann somit auch für Netzwerkoperationen verwendet werden.

```
{
    QFile file("example.txt");
    if (file.open(QIODevice::WriteOnly | QIODevice::Text))
        QTextStream out(&file);
        out << "Hello World" << endl;
    }
    /* close() wird automatisch aufgerufen */
}
```

QFile und Encodings

Welches Encoding wird verwendet? QFile verwendet das Betriebssystem Encoding wenn kein implicites Encoding gesetzt ist. Das wird zu Problemen führen.

Also das Encoding bitte setzen. Erraten ist schwierig.

```
QTextStream stream(&file);
stream.setCodec("UTF-8");
```

BOM (Byte Order Mark) wird von Qt automatisch erkannt und verwendet.

QFile und QDataStream

QDataStream ist ein Binär-Stream und kann auch für die Serialisierung von Objekten verwendet werden.

```
QFile file("example.dat");
if (file.open(QIODevice::WriteOnly)) {
    QDataStream out(&file);
    out << QString("Hello, World!");
}
```

QFile und QDataStream, Serialisierung

Serialisierung von Objekten:

```
QFile file("example.dat");
if (file.open(QIODevice::WriteOnly)) {
    QDataStream out(&file);
    out << QString("Hello, World!");
    out << 42;
    out << 3.14;
}
```

QFile und QDataStream, Teil 2

Anschließendes Deserialisieren:

```
QFile file("example.dat");
if (file.open(QIODevice::ReadOnly)) {
    QDataStream in(&file);
    QString str;
    int i;
```

QFile und QByteArray

Bytearrays können auch direkt in Dateien geschrieben werden.

```
QByteArray data = QByteArray::fromHex("deadbeef");
QFile file("example.dat");
if (file.open(QIODevice::WriteOnly)) {
    file.write(data);
}
```

QFile, QByteArray und QJsonDocument

QByteArray kann Json Daten aufnehmen. JsonDocument::Compact oder JsonDocument::Indented kann verwendet werden. Ersteres spart Platz.

```
void writeJsonToFile(const QJsonObject &jsonObj, const QSt
    QFile file(fileName); // keine fehlerbehandlung hier.
    QJsonDocument jsonDoc(jsonObj);
    QByteArray jsonData = jsonDoc.toJson(QJsonDocument::Inc
    file.write(jsonData);
}
```

Serialisierung und Deserialisierung und QJsonDocument

Um eigene C++-Klassen im Json-Format zu speichern oder zu l
MANUELL vorgehen:

```cpp
class MyClass {
public:
```

Network-Accesss ist asynchron

- NetworkAccessManager
- Alternative: Libraries in eigenen QThreads verwenden (libcurl)
- Alternative mit bon-blocking IO: boost::asio

GUI

- GUIs / Interfaces sind die Stärke von QT.
- Qt Designer ist ein WYSIWYG-Editor.
- Qt Quick ist eine Alternative zu Widgets.
- Qt wurde für die Entwicklung von GUIs entworfen.

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.

QWidget und Children

Ein QWidget hat children und ein QWidget hat in der Regel auch einen Parent. Der Parent kann aber via mywidget->setParent(nullptr) entfernt werden. Parent-Widgets

QWidget Features

- findChild() und findChildren()
- childAt(Qpoint pos) findet das Top-Level Kind an der Position pos.
- z-index mittels stackUnder(QWidget *w), raise() und lower() steuerbar.
- Zugriff auf QScreen* screen() und QWindow* window()y
- Scrolling wird direkt von QWidget unterstützt, aber QScrollArea ist die bessere Alternative.
- Enablement, Transparenz, Double-Buffering und Compositing

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 oder um einmalig später auszuführen.
- 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
- QWidget::setFixedSize() und QWidget::setMinimumSize() und QWidget::setMaximumSize()

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 mittels **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
- Browser können in QtWidgets embedded 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?