

Qt Features and Programming with Qt

Qt Training - TAMK, 21.4.2009

Tony Torp (tony.torp@tamk.fi)

Lecturer

- Tony Torp, 33
- DI (ohjelmistotekniikka, tietoliikennetekniikka), TTY 2000
- Historiikki
 - 1998 tuntiopettaja, ohjelmistotekniikka (TTY)
 - 1999-2002 Nokia Symbian/S60
 - 2002-> ohjelmistotekniikan lehtori, TAMK
- Työnkuva: projekteja (50%), kursseja ja päättötöitä(50%) 😊
 - C++, olio-ohjelmointi, ohjelmistotuotanto,
 - Ohjelmistotekniikan projektityö, Ohjelmistotekniikan työkurssi
 - Tietoliikennetekniikan jatkokurssi, Tietoliikennetekniikan laboratoriotyöt,
 - Java 1, Java 2
 - Symbian perus- ja jatkokurssit

Contents

1. Parents, Children and Memory Management
2. Inter-object Communications
3. Meta-object System
4. Event Handling Mechanism
5. Strings
6. Multithreading
7. Containers and Algorithms
8. Qt Module Walkthrough
9. Qt Development Tools

How much C++ needed

- Objects and classes - Declaring a class, inheritance, calling member functions etc
- Polymorphism - that is virtual methods
- Operator overloading
- Templates - for the container classes only.
- No RTTI, no sophisticated templates, no exceptions thrown...

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

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QPushButton hello("Hello world!");
    hello.resize(100, 30);
    hello.show();
    return app.exec();
}
```

Contents

1. Parents, Children and Memory Management
2. Inter-object Communications
3. Meta-object System
4. Event Handling Mechanism
5. Strings
6. Multithreading
7. Containers and Algorithms
8. Qt Module Walkthrough
9. Qt Development Tools

QObject

- Base class for Qt objects providing
 - Signal/slot connections between any QObject
 - Object trees and object ownership (memory management)
 - Event handling
 - Run-time class and object information
 - object name
 - class name
 - inheritance hierarchy etc.
 - Timing services
- Qt meta-object system (and compiler) needed for gaining all these

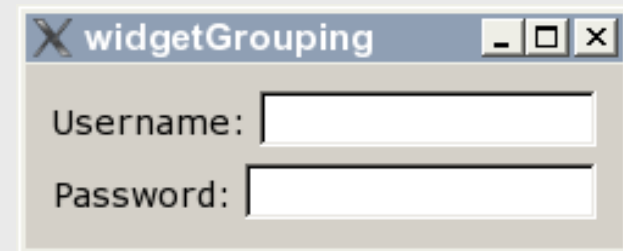
Parent-Child Relationship

- When a **QObject** is created, it is given its parent as an argument

QObject (**QObject** * *parent* = 0)

- The child informs its parent about its existence, upon which the parent adds it to its own list of children
- For **QWidget**s: If *parent* is 0, the new widget becomes a window

```
int main(int argc, char *argv[])
{
    QApplication app( argc, argv );
    QWidget top;
    QLabel *nameLabel = new QLabel("Username:", &top);
    QLineEdit *nameEdit = new QLineEdit(&top);
    QLabel *passLabel = new QLabel("Password:", &top);
    QLineEdit *passEdit = new QLineEdit(&top);
    QVBoxLayout *vlay = new QVBoxLayout(&top);
    QHBoxLayout *nameLayout = new QHBoxLayout;
    QHBoxLayout *passLayout = new QHBoxLayout;
    vlay->addLayout(nameLayout);
    vlay->addLayout(passLayout);
    nameLayout->addWidget(nameLabel);
    nameLayout->addWidget(nameEdit);
    passLayout->addWidget(passLabel);
    passLayout->addWidget(passEdit);
    top.show();
    app.exec();
}
```



Memory management rules

- QObject derived classes are allocated on the heap using `new`
- The parent takes ownership of the object so no explicit delete needed
`QPushButton* button = new QPushButton("Ok",parent);`
- Objects not inheriting QObject are allocated on the stack
`QColor color(100,12,100);`
`QString greeting("Hello Qt");`
- Exceptions
 - QApplication and QFile are allocated on the stack

Parent's responsibilities

- Deletes all children when it is deleted itself
 - But does not set your own possible pointers to those objects to NULL!
 - The parent is informed if a child is deleted manually
- Hides/shows the children when it is hidden/shown itself
- enables/disables the children when it is enabled/disabled itself

Is this ok?

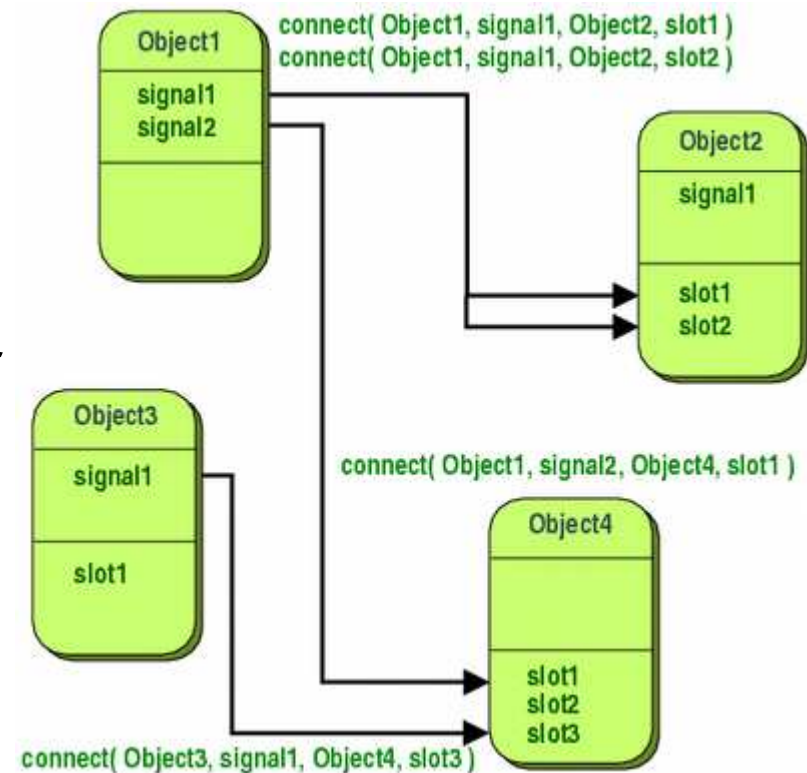
```
int main( int argc, char* argv[] )
{
    QApplication myApp( argc, argv );
    QLabel myLabel("My text");
    QWidget window;
    myLabel.setParent(&window);
    window.show();
    myApp.exec();
}
```

Contents

1. Parents, Children and Memory Management
2. Inter-object Communications
3. Meta-object System
4. Event Handling Mechanism
5. Strings
6. Multithreading
7. Containers and Algorithms
8. Qt Module Walkthrough
9. Qt Development Tools

Signals and Slots

- Type-safe callback between QObjects
- Loose coupling and encapsulation
 - Sender and receiver do not “know” about each other
- A signal is emitted when a particular event occurs.
- A slot is a function that is called in response to a particular signal
- Works across threads



```
class Counter : public QObject
{
    Q_OBJECT

public:
    Counter() { m_value = 0; }

    int value() const { return m_value; }

public slots:
    void setValue(int value);

signals:
    void valueChanged(int newValue);

private:
    int m_value;
};
```

```
void Counter::setValue(int value)
{
    if (value != m_value) {
        m_value = value;
        emit valueChanged(value);
    }
}
```

```
Counter a, b;
QObject::connect(&a, SIGNAL(valueChanged(int)),
                &b, SLOT(setValue(int)));
```

```
a.setValue(12); // a.value() == 12, b.value() == 12
b.setValue(48); // a.value() == 12, b.value() == 48
```

Emitting a signal

- All connected slots to be called in unspecified order
`emit mySignal("info", 100,)`

Real-life snippet

- Problem: How do you reach to a push button being pressed?
- Solution: Implement a slot in your dialog and connect the button to it

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

...more

- One signal can be connected to many slots
 - The slots are executed in arbitrary order
- Several signals can be connect to one slot
- The signal source object can be accessed

```
QObject* QObject::sender() const
```

- A snippet

```
sender()->inherits("QAbstractButton")
sender()->objectName == "okButton"
if (QAbstractButton *button =
    qobject_cast<QAbstractButton *>(sender()))
    button->toggle();
```

...more

- It is not allowed to name parameters in connect:

```
connect( m_slider, SIGNAL( valueChanged(int value)),  
        this, SLOT( setValue( int newValue )))
```

- Signal can be connected to a signal:

```
connect( m_ok, SIGNAL( clicked() ),  
        this, SIGNAL( okSignal() ) )
```

...more

- You can connect a signal with more parameters to a slot with fewer

```
connect( m_slider, SIGNAL( valueChanged(int)),  
        this, SLOT( updateUI()))
```

- It is not allowed to connect a signal with fewer parameters to a slot with more

```
connect( but, SIGNAL( clicked() ) )  
        this, SLOT( setText(const QString&)) )
```

Contents

1. Parents, Children and Memory Management
2. Inter-object Communications
3. Meta-object System
4. Event Handling Mechanism
5. Strings
6. Multithreading
7. Containers and Algorithms
8. Qt Module Walkthrough
9. Qt Development Tools

Meta-Object System

- Meta object system is required in order to use signal/slot mechanism or events
- All objects inheriting from `QObject` can use meta object system features
- `Q_OBJECT` macro must be added into class declaration
- Qt's meta object compiler (MOC) goes through the **headers** to produce standard C++ meta object code for those classes

Meta object system

```
class MyClass: public QObject
{
    Q_OBJECT
public:
    // declare public members here
private:
    // declace private members here
public slots:
    void mySlot();
    // slots can also be called as normal member functions
signals:
    void mySignal(int value);
};
```

Meta objects of QObject

- `QMetaObject` contains meta-information about Qt objects
- `QObject::metaObject()` returns the meta object of the class. Some methods
 - `className()` returns the name of a class.
 - `superClass()` returns the superclass's meta-object.
 - `method()` and `methodCount()` provide information about a class's meta-methods (signals, slots and other member functions).
 - `enumerator()` and `enumeratorCount()` provide information about a class's enumerators.
 - `classInfo()` returns class info, which can be e.g. author and url

Dynamic casting in Qt

```
QObject *obj = new Mywidget;

QWidget *widget = qobject_cast<QWidget*>(obj);
Mywidget *mywidget = qobject_cast<Mywidget*>(obj);
QLabel *label = qobject_cast<QLabel*>(obj); //label is 0

if (QLabel *label = qobject_cast<QLabel*>(obj))
    label->setText("Hello");
else if(QPushButton *button = qobject_cast<QPushButton*>(obj))
    button->setText("World");
```

Contents

1. Parents, Children and Memory Management
2. Inter-object Communications
3. Meta-object System
4. Event Handling Mechanism
5. Strings
6. Multithreading
7. Containers and Algorithms
8. Qt Module Walkthrough
9. Qt Development Tools

Event types

- Sent by window system (GUI)
 - mouse movements, clicks, key presses
- System
 - Timer events
 - Socket notifier
- Application itself
 - Application specific events (you can send events across your application)
- Event subclasses: `QResizeEvent`, `QPaintEvent`, `QMouseEvent`, `QKeyEvent`, `QTimerEvent`

Event loops

- The application main thread event loop is started by:
`QApplication::exec()`
- Application can now receive events
- The event loop is exited by
`QApplication::exit(int returnCode = 0)` or
`QApplication::quit() // equivalent with exit(0)`

Delivering events

- An event is an object that inherits QEvent
- An event causes Qt to create an event object which is delivered to the particular handler object:

```
QObject::event(QEvent& e) //virtual
```

Widgets and event categories

- QWidget reimplements the `event()` method to call the specialized virtual handler methods such as
 - `void QWidget::mousePressEvent(QMouseEvent*)`
 - `void QWidget::keyPressEvent(QKeyEvent*)`
 - `void QWidget::focusInEvent(QFocusEvent*)`
 - `void QWidget::paintEvent(QPaintEvent*)...`
- When implementing widgets, just overwrite these empty implementations in your subclass to receive and respond to those events

Handling events

```
void MyCheckBox::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        // handle left mouse button here
    }
    else {
        // pass on other buttons to base class
        QCheckBox::mousePressEvent(event);
    }
}
```

Event filters

- A mechanism for any QObject to listen to events of another QObject
- An event filter gets to process events **before** the target object does and **decides** if the event is propagated further
- Procedure:
 - Register your QObject as an event filter with `installEventFilter()`
 - Handle received events with `eventFilter()`

Example: event filter getting all app events

```
MyAppEventListener::MyAppEventListener()
{
    qApp->installEventFilter( this );
}
bool MyAppEventListener::eventFilter( QObject obj, QEvent *event )
{
    if( event->type == 8000 ) // user events between 1024 and 65535
    {
        QMyEvent *myEvent = static_cast<QMyEvent*>(event);
        handleMyEvent( myEvent->MyEventData() );
        return true;
    }
    else if( event->type == 9000 )
    {
    }
    return false;
}
```

Synthetic Events

- You can make a widget believe that it gets events from the underlying system by posting or sending events to it
- Events **posted** to a widget are put into the event queue and handled during the next pass through the event loop
- Events **sent** to a widget are handled right away
- You post an event using `QCoreApplication::postEvent()`, and send one using `QCoreApplication::sendEvent()`
- Both methods take the widget to send the event to as the first argument and the event as the second.

Synthetic events

qApp->postEvent()

- Adds the event *event*, with the object *receiver* as the receiver of the event, to an event queue and returns immediately.
- The event must be allocated on the heap since the post event queue will take ownership of the event and delete it once it has been posted.

qApp->sendEvent()

- processes the event immediately, the event is handled when this returns. `isAccepted()` is implemented in many events to tell if the event was accepted
- User defined event custom types between `QEvent::User` and `QEvent::MaxUser`
- The framework handles custom event types by invoking virtual `QObject::customEvent()` method

Do not confuse events with signals/slots

- Signals are just emitted without knowing the receiver. Events go to a specific widget
- Signals can have any number of receivers. Events go to exactly one receiver
- Events pass through event filters, signals don't

Do not confuse events with signals/slots

- Signals occur between two objects **inside the application**
- Events that come from **outside** of the application
 - User events like mouse or key press
 - Low level events from Qt modules like networking etc.
- Events that come from **inside** of the application
 - Child object added, removed etc.
 - User defined events
- Signals are used when **using** widgets, events are used when **implementing** widgets

Timer with events (QObject)

```
class MyObject : public QObject
{
    Q_OBJECT
public:
    MyObject(QObject *parent = 0);
protected:
    void timerEvent(QTimerEvent *event);
};

MyObject::MyObject(QObject *parent) : QObject(parent) {
    startTimer(50); // 50-millisecond timer
    startTimer(1000); // 1-second timer
    startTimer(60000); // 1-minute timer
}

void MyObject::timerEvent(QTimerEvent *event)
{
    qDebug() << "Timer ID:" << event->timerId();
}
```

Timer with signal-slot (QTimer)

```
QTimer* timer = new QTimer( this );  
connect( timer, SIGNAL(timeout()),  
        this, SLOT(handleTimeout() ) );  
timer->start( 50 );
```

```
void MyObject::handleTimeout()  
{  
    qDebug() << "Timer event...";  
}
```

Contents

1. Parents, Children and Memory Management
2. Inter-object Communications
3. Meta-object System
4. Event Handling Mechanism
5. Strings
6. Multithreading
7. Containers and Algorithms
8. Qt Module Walkthrough
9. Qt Development Tools

QString

- Strings can be created in a number of ways
- Copy constructor and assignment operators

```
QString str("abc");  
str = "def";
```
- From a number using the static method `number()`
- From a char pointer using the static methods `fromLatin1()`, `fromUtf8()`, `fromLocal8Bit()`...

QString

- Strings can be created from other strings by assigning and using `operator+` and `operator+=`

```
QString str = str1 + str2;  
fileName += ".txt";
```

- A version with duplicate whitespace removed using `simplified()`
- A part of a string using one of `left()`, `mid()`, and `right()`
- Padded version using `leftJustified()` and `rightJustified()`

```
QString s = "apple";  
QString t = s.leftJustified(8, '.');  
// t == "apple..."
```

QString

- Data can be extracted from strings using
 - Numbers: `toInt()`, `toFloat()`
 - String: `toLatin1()`, `toUtf8()`, `toLocal8Bit()`
- The later methods return a `QByteArray`, from which you can get a `char*` using `QByteArray::data()` or `QByteArray::constData()`

QString

- Strings can be tested
 - `length()` returns the length of the string.
 - `endsWith()` and `startsWith()` test whether the string starts or ends with an other string.
 - `contains()` returns whether the string matches a given expression, `count()` tells you how many times.
 - `indexOf()` and `lastIndexOf()` search for the next matching expressions, and return its index. The expression can be a sequence of characters, strings, or regular expressions.

QPointer

- Safe auto pointer
- The pointer is NULL:ed automatically when the object it refers to is deleted
- QPointers objects can be used as normal pointers

```
QPointer<QPushButton> pButton = new QPushButton("OK");  
QPointer<QPushButton> pButton2 = pButton;  
pButton->setText( "Cancel" );  
delete pButton; // pButton2 is also set to NULL
```

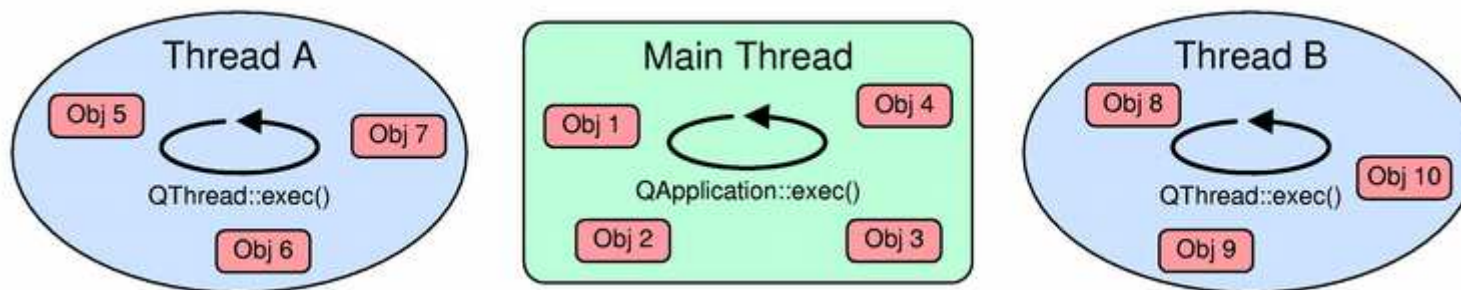
Contents

1. Parents, Children and Memory Management
2. Inter-object Communications
3. Meta-object System
4. Event Handling Mechanism
5. Strings
6. Multithreading
7. Containers and Algorithms
8. Qt Module Walkthrough
9. Qt Development Tools

Multithreading

```
class MyThread : public QThread
{
    Q_OBJECT
protected:
    void run();
};

void MyThread::run() { ... }
```



Multithreading

- Classes for
 - Threads (`QThread`)
 - Mutexes (`QMutex`)
 - Semaphores (`QSemaphore`)
 - Thread-global storage (`QThreadStorage`)
 - Classes supporting various locking mechanisms
- More on threads
<http://doc.trolltech.com/4.5/threads.html>

```
class MyThread : public QThread
{
protected:
    void run();
};
void MyThread::run()
{
    QTcpSocket socket;
    // connect signals somewhere meaningful ...
    socket.connectToHost(hostName, portNumber);
    exec();
}
...
MyThread* myThread = new MyThread(...);
myThread->start();
...
myThread->exit();
```

Signals and slots across threads

- Direct connections
 - the slot gets called immediately when the signal is emitted. The slot is executed in the thread that emitted the signal (which is not necessarily the thread where the receiver object lives).
- Queued connections
 - the slot is invoked when control returns to the event loop of the thread to which the object belongs. The slot is executed in the thread where the receiver object lives.
- Auto connection (the default)
 - the behavior is the same as with direct connections if the signal is emitted in the thread where the receiver lives; otherwise, the behavior is that of a queued connection

Contents

1. Parents, Children and Memory Management
2. Inter-object Communications
3. Meta-object System
4. Event Handling Mechanism
5. Strings
6. Multithreading
7. Containers and Algorithms
8. Qt Module Walkthrough
9. Qt Development Tools

Containers

- The Qt library provides a set of general purpose template-based container classes.
- You may want to use these classes if you are unfamiliar with the STL, or prefer to do things "the Qt way".
- If you are already familiar with the STL, feel free to continue using it.
- Methods exist that convert between Qt and STL containers, e.g. if you need to pass the contents of a `std::list` to a Qt method.

Containers

- `QLinkedList<T>` A doubly linked list. Optimized for inserting items in the middle (cf. `std::list`).
- `QVector<T>` A list optimized for random access (cf. `std::vector`).
- `QList<T>` An array list; good default container (cf. `std::deque`).
- `QStack<T>` This is a convenience subclass of `QVector` that provides "last in, first out" (LIFO) semantics (cf. `std::stack`).
- `QQueue<T>` This is a convenience subclass of `QList` that provides "first in, first out" (FIFO) semantics (cf. `std::queue`).

Containers

- **QSet<T>** A single-valued mathematical set with fast lookups (cf. `std::set`).
- **QMap<K,T>** A dictionary (associative array) that maps keys of type K to values of type T (cf. `std::map`).
- **QMultiMap<K,T>** A variant of QMap that supports maps where one key can be associated with multiple values (cf. `std::multimap`).
- **QHash<K,T>** A variant of QMap using hash tables to perform lookup and insertion in constant time (cf. `std::tr1::unordered_map`).
- **QMultiHash<K,T>** A variant of QHash that supports hashes where one key can be associated with multiple values (cf. `std::tr1::unordered_multimap`).

An example: iterating a map

```
QMap<QString, QString> map;
map.insert("Paris", "France");
map.insert("Guatemala City", "Guatemala");
map.insert("Mexico City", "Mexico");
map.insert("Moscow", "Russia");
QMutableMapIterator<QString, QString> i(map);
while (i.hasNext()) {
    if (i.next().key().endsWith("City"))
        i.remove();
}
}
```

Qt's foreach "keyword"

```
QLinkedList<QString> list;
...
foreach (QString str, list) {
    if (str.isEmpty())
        break;
    qDebug() << str;
}
```

Algorithms

- `qSort` sorts items in a range.
- `qStableSort` similar to `sort`, but guarantees that the order between two equal items stays unchanged.
- `qFind` searches for a value.
- `qEqual` checks if two ranges are the same.
- `qCopy` copies items from one range to another.
- `qCopyBackward` copies items backwards.
- `qCount` counts the number of items matching search criteria.
- Qt also comes with a (small) set of parallel (ie. multi-threaded) algorithms in the `QtConcurrent` namespace

Algorithm example

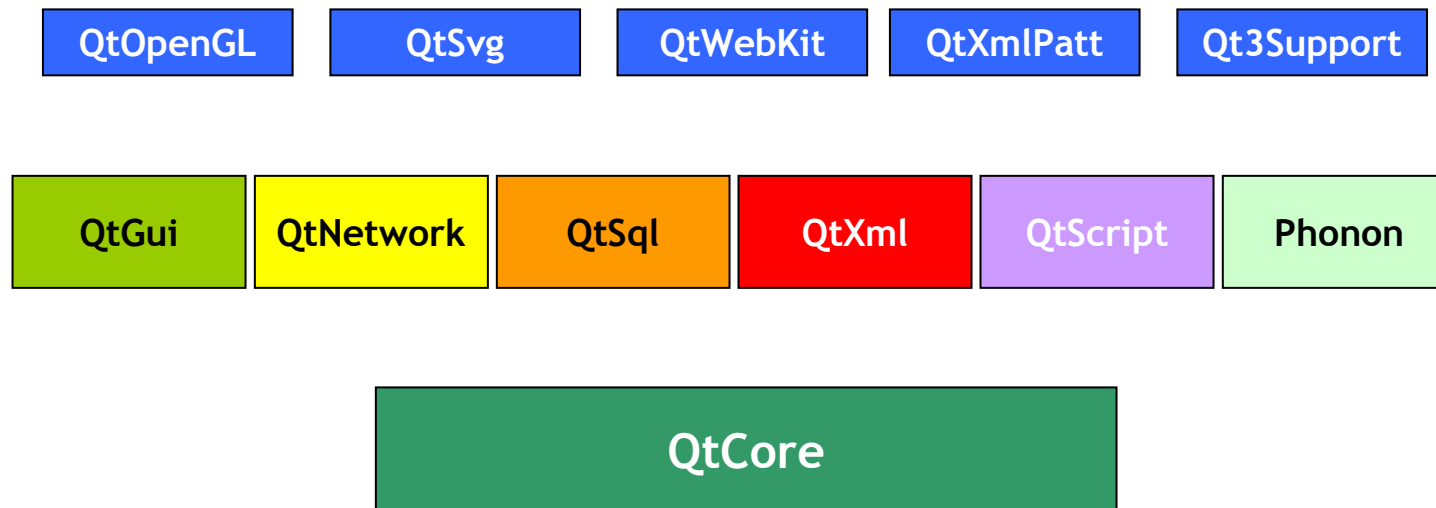
```
QList<QString> list;  
list << "one" << "two" << "three";  
QVector<QString> vect1(3);  
qCopy(list.begin(), list.end(), vect1.begin());  
// vect: [ "one", "two", "three" ]
```

Contents

1. Parents, Children and Memory Management
2. Inter-object Communications
3. Meta-object System
4. Event Handling Mechanism
5. Strings
6. Multithreading
7. Containers and Algorithms
8. Qt Modules Walkthrough
9. Qt Development Tools

Qt as a whole package

- Cross-platform Qt consist of ~700 classes
- Build tools (qmake, moc, uic)
- Development tools (Qt Designer, Qt Assistant, QtLinguist)



I/O

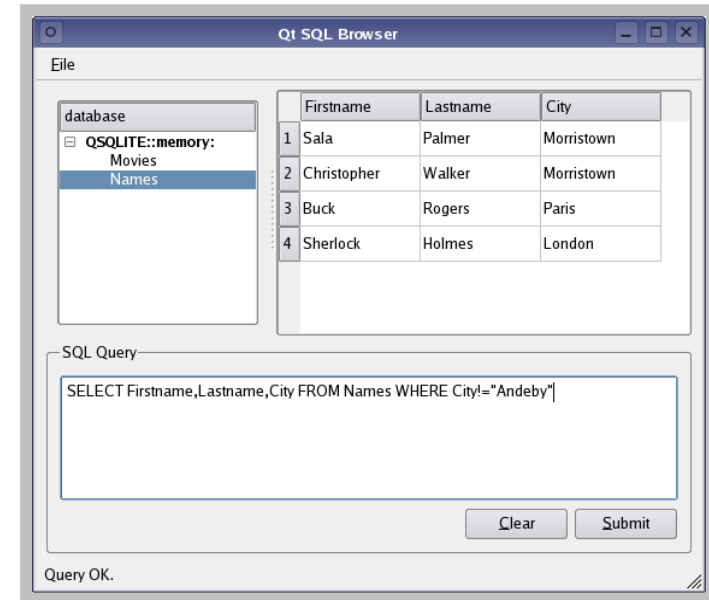
- Reading from or writing to files or other devices
- **QIODevice** as an abstraction
- Concrete classes
 - **QFile**
 - **QTemporaryFile**
 - **QBuffer**
 - **QProcess**
 - **QTcpSocket**
 - **QUdpSocket**
 - **QSslSocket**

Qt Networking

- Provides TCP/IP networking functionality
- TCP sockets for clients and for servers
- Encapsulates TCP and UDP APIs
- SSL support
- HTTP 1.1 compliant asynchronous API
- FTP, DNS implementation

Qt Database Classes

- Provide platform and database-independent access functionality
- Driver Layer
 - Low-level bridge between specific databases and the SQL API layer
- SQL API Layer
 - Provide access to databases
- User Interface Layer
 - Link data from a database to data-aware widgets
- Supports most major database drivers
 - DB2, IBASE, MySQL, OCI, ODBC, PSQL, SQLITE, TDS



Qt OpenGL Classes

- Allows you to build your user interface in Qt, display and manipulate 3D model in OpenGL®
- Integrates OpenGL canvas with Qt
- Provides frame buffer and pixel buffer abstraction
- Supports accelerating 2D painting with OpenGL
- Mix 2D painting and 3D scenes

Qt XML Classes

- Core Module
 - Simple XML stream reader and writer
- XML Module
 - A well-formed XML parser using the SAX2 (Simple API for XML) interface
 - Implementation of the DOM Level 2 (Document Object Model)
- XmlPatterns module
 - An implementation of the XQuery standard
 - Enable users to query XML files similar to SQL
 - Semantics for value assignment, filtering, and simple operations
 - Fully controllable output formatting

Qt WebKit Integration

- An open source HTML rendering component integrated with Qt
- Web standards compliant
 - support for HTML, XHTML, XML, stylesheets, JavaScript, HTML editing, HTML canvas, AJAX, XSLT, XPath, some SVG.
- Deployable wherever Qt is: cross-platform/cross-version/cross-device
- Interact with Web environment, expose native objects



Phonon Multimedia Framework

- Single, easy to use API ("Phonon") for playback
- Provides possibility to play/synchronize multiple sound/video streams
- Will use native back-end format support
 - DirectShow on Windows
 - GStreamer on X11
 - QuickTime on Mac
- Plan to add more video and authoring support in the future

Contents

1. Parents, Children and Memory Management
2. Inter-object Communications
3. Meta-object System
4. Event Handling Mechanism
5. Strings
6. Multithreading
7. Containers and Algorithms
8. Qt Modules Walkthrough
9. Qt Development Tools

qmake

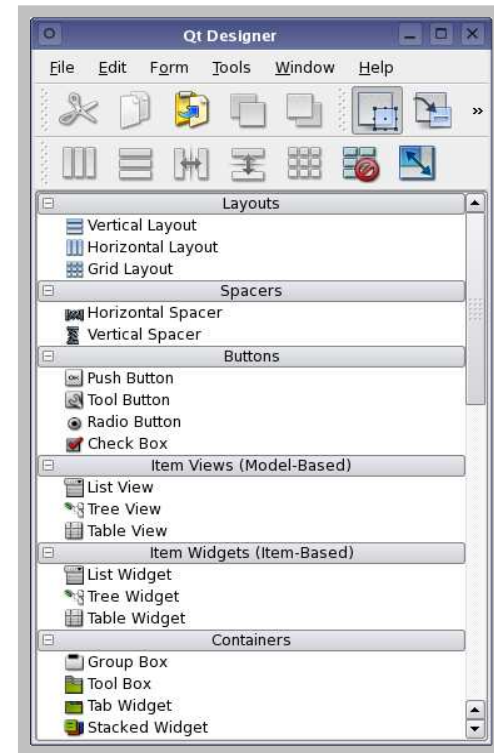
- A cross-platform application build tool
- Features
 - Reads project source, generates dependency tree, generates platform specific project and makefiles
- Benefits
 - Takes the pain out of cross-platform builds
 - Eliminates the need for makefile generation

Meta-Object Compiler (moc)

- Handles Qt's C++ extensions
- Reads C++ header files and finds class declarations that contain `Q_OBJECT` macro
- Required for
 - signals-slots
 - run-time type information
 - dynamic properties

Qt Designer

- Qt Designer is a powerful, drag-and-drop GUI layout and forms builder
- Features
 - Supports forms and dialog creation with instant preview
 - Integrates with Qt layout system
 - Extensive collection of standard widgets
 - Support for custom widgets and dialogs
 - Seamless integration with Microsoft® Visual Studio .NET and Eclipse
- Benefits
 - Greatly speeds the interface design process
 - Enables native look and feel across all supported platforms
 - Developers work within the environment of their choice, leveraging existing skills

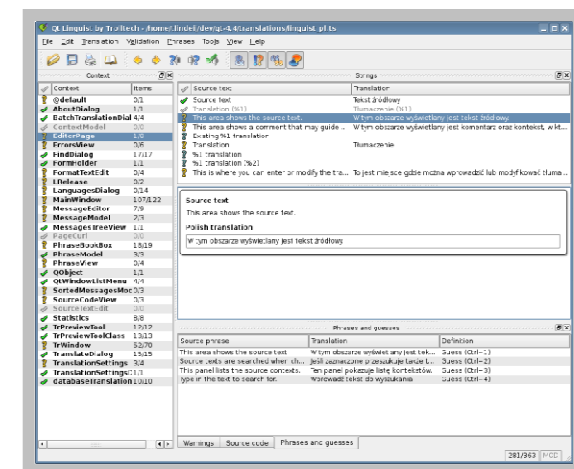


User Interface Compiler (uic)

- Reads an XML format user interface definition (.ui) file created by Qt designer and creates a corresponding header file.

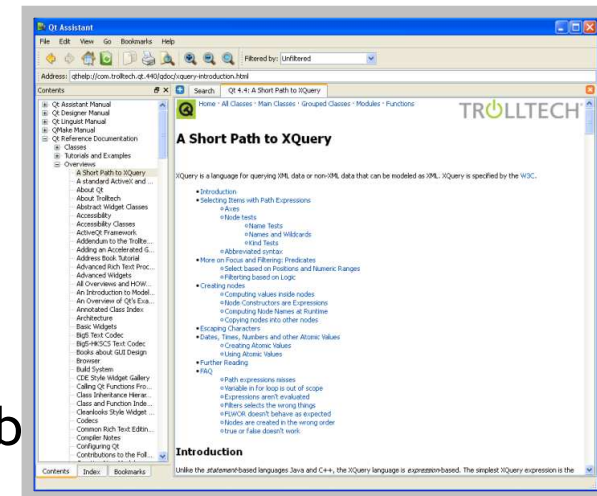
Qt Linguist

- Qt Linguist is a set of tools that smooth the internationalization workflow
- Features
 - Collects all UI text and presents it to a human translator in a simple window
 - Support for all languages, including CJK
 - Simultaneous support for multiple languages and writing systems from within a single application binary
- Benefits
 - Greatly speeds the translation/localization process
 - Works with Qt's language-aware layout engine for clean, consistent interface no matter the language
 - Easily target international markets



Qt Assistant

- Qt Assistant is a fully customizable, redistributable help file/documentation browser
- Features
 - Simple, web-browser-like navigation, bookmarking and linking of documentation files
 - Support for rich text and HTML
 - Full text and keyword lookup
 - Can be customized and shipped with Qt applications
- Benefits
 - No longer have to build a help system from scratch
 - Leverage existing HTML skills
 - Deliver documentation in an easily searchable and navigable format to your end users



Some IDEs for QT development

- QT Creator (Windows, Ubuntu, MAC OS)
 - C++ code editor
 - GUI designer
 - Project and build management tools
 - Integrated help system
 - Visual debugger
 - Code navigation tools
- Carbide.c++ (Windows) for S60

Thank you!