Quick Introduction to Qt Programming

CMPS160
The enclosed Qt Educational Training Materials are provided under the Creative Commons Attribution-Non-Commercial-Share Alike 2.5 License Agreement.

The full license text is available here: http://creativecommons.org/licenses/by-nc-sa/2.5/legalcode.

Nokia, Qt and the Nokia and Qt logos are the registered trademarks of Nokia Corporation in Finland and other countries worldwide.
What is Qt?

"Qt is a cross platform development framework written in C++ ."

- C++ framework – bindings for other languages
  - Python, Ruby, C#, etcetera

- Originally for user interfaces – now for everything
  Databases, XML, WebKit, multimedia, networking, OpenGL, scripting, non-GUI...
What is Qt?

- Qt is built from modules
  - All modules have a common scheme and are built from the same API design ideas
What is Qt?

- Qt extends C++ with macros and introspection

```cpp
foreach (int value, intList) { ... }

QObject *o = new QPustButton;
o->metaObject()->className(); // returns "QPushButton"

connect(button, SIGNAL(clicked()), window, SLOT(close()));
```

- All code is still plain C++
The Purpose of Qt

- Cross platform applications built from one source
- Builds native applications with native look and feel
- Easy to (re)use API, high developer productivity, openness, fun to use
Desktop target platforms

- Windows
- Mac OS X
- Linux/Unix X11
Embedded target platforms

- Windows CE
- Symbian
- Maemo
- Embedded Linux
  - Direct framebuffer access
Hello World
```c++
#include <QApplication>
#include <QLabel>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```
Hello World

```c++
#include <QApplication>
#include <QLabel>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```
Hello World

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

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```
```cpp
#include <QApplication>
#include <QLabel>

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```
Hello World

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

int main( int argc, char **argv )
{
    QApplication app( argc, argv );
    QLabel l( "Hello World!" );
    l.show();
    return app.exec();
}
```
The QObject

- **QObject** is the base class of almost all Qt classes and all widgets

- It contains many of the mechanisms that make up Qt
  - events
  - signals and slots
  - properties
  - memory management
The QObject

- **QObject** is the base class to most Qt classes. Examples of exceptions are:
  - Classes that need to be lightweight such as graphical primitives
  - Data containers (QString, QList, QChar, etc)
  - Classes that needs to be copyable, as QObject cannot be copied
Meta data

- Qt implements introspection in C++
- Every \texttt{QObject} has a \textit{meta object}
- The meta object knows about
  - class name (\texttt{QObject::className})
  - inheritance (\texttt{QObject::inherits})
  - properties
  - signals and slots
  - general information (\texttt{QObject::classInfo})
Meta data

- The meta data is gathered at compile time by the meta object compiler, moc.

Ordinary C++ Build Process

sources *.cpp

compiles

object files *.o

links

executables

headers *.
h

includes
The meta data is gathered at compile time by the meta object compiler, *moc*.

The moc harvests data from your headers.

**Qt C++ Build Process**

- **Sources**: `*.cpp`
- **Includes**: `*.h`
- **Generated**: `moc_*.cpp`
- **Object files**: `*.o`
- **Executables**: 

**Steps:**
- Compiles `*.cpp`
- Compiles `moc_*.cpp`
- Links `*.o` and `moc_*.cpp` to produce executables
### What does moc look for?

```cpp
class MyClass : public QObject
{
    Q_OBJECT
    Q_CLASSINFO("author", "John Doe")

public:
    MyClass(const Foo &foo, QObject *parent=0);

    Foo foo() const;

public slots:
    void setFoo( const Foo &foo );

signals:
    void fooChanged( Foo );

private:
    Foo m_foo;
};
```

- **The Q_OBJECT macro**, usually first.
- **Qt keywords**
- **Make sure that you inherit QObject first (could be indirect)**
- **General info about the class**
Signals and Slots

- Dynamically and loosely tie together events and state changes with reactions

- What makes Qt tick
Signals and Slots vs Callbacks

- A callback is a pointer to a function that is called when an event occurs, any function can be assigned to a callback
  - No type-safety
  - Always works as a direct call

- Signals and Slots are more dynamic
  - A more generic mechanism
  - Easier to interconnect two existing classes
  - Less knowledge shared between involved classes
What is a slot?

- A slot is defined in one of the slots sections
  ```cpp
  public slots:
      void aPublicSlot();
  protected slots:
      void aProtectedSlot();
  private slots:
      void aPrivateSlot();
  ```

- A slot can return values, but not through connections
- Any number of signals can be connected to a slot
  ```cpp
  connect(src, SIGNAL(sig()), dest, SLOT(slt()));
  ```
- It is implemented as an ordinary method
- It can be called as an ordinary method
What is a signal?

- A signal is defined in the signals section

```cpp
signals:
  void aSignal();
```

- A signal always returns void
- A signal must not be implemented
  - The moc provides an implementation
- A signal can be connected to any number of slots
- Usually results in a direct call, but can be passed as events between threads, or even over sockets (using 3rd party classes)
- The slots are activated in arbitrary order
- A signal is emitted using the emit keyword

```cpp
emit aSignal();
```
Making the connection

QObject::connect( src, SIGNAL( signature ), dest, SLOT( signature ) );

A signature consists of the function name and argument types. No variable names, nor values are allowed.

setTitle(QString text)
setValue(42)
setItem(ItemClass)

clicked()
toggled(bool)
setText(QString)
textChanged(QString)
rangeChanged(int,int)

Custom types reduces reusability.
Making the connection

- Qt can ignore arguments, but not create values from nothing

### Signals
- `rangeChanged(int,int)`
- `rangeChanged(int,int)`
- `rangeChanged(int,int)`

### Slots
- `setRange(int,int)`
- `setValue(int)`
- `updateDialog()`

### Connections

<table>
<thead>
<tr>
<th>Signals</th>
<th>Slots</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>rangeChanged(int,int)</code></td>
<td><code>setRange(int,int)</code></td>
</tr>
<tr>
<td><code>rangeChanged(int,int)</code></td>
<td><code>setValue(int)</code></td>
</tr>
<tr>
<td><code>rangeChanged(int,int)</code></td>
<td><code>updateDialog()</code></td>
</tr>
<tr>
<td><code>valueChanged(int)</code></td>
<td><code>setRange(int,int)</code></td>
</tr>
<tr>
<td><code>valueChanged(int)</code></td>
<td><code>setValue(int)</code></td>
</tr>
<tr>
<td><code>valueChanged(int)</code></td>
<td><code>updateDialog()</code></td>
</tr>
<tr>
<td><code>textChanged(QString)</code></td>
<td><code>setValue(int)</code></td>
</tr>
<tr>
<td><code>clicked()</code></td>
<td><code>setValue(int)</code></td>
</tr>
<tr>
<td><code>clicked()</code></td>
<td><code>updateDialog()</code></td>
</tr>
</tbody>
</table>
Automatic Connections

- When using Designer it is convenient to have automatic connections between the interface and your code

  ```
  on_ object name _ signal name ( signal parameters )
  ```

  ```
  on_addButton_clicked();
  on_deleteButton_clicked();
  on_listWidget_currentItemChanged(QListWidgetItem*,QListWidgetItem*)
  ```

- Triggered by calling `QMetaObject::connectSlotsByName`

- Think about reuse when naming
  - Compare `on_widget_signal` to `updatePageMargins`

  *updatePageMargins can be connected to a number of signals or called directly.*
Synchronizing Values

- Connect both ways

```cpp
connect(dial1, SIGNAL(valueChanged(int)), dial2, SLOT(setValue(int)));
connect(dial2, SIGNAL(valueChanged(int)), dial1, SLOT(setValue(int)));
```

- An infinite loop must be stopped – no signal is emitted unless an actual change takes place

```cpp
void QDial::setValue(int v) {
    if(v==m_value) return;
    ...
}
```

This is the responsibility of all code that can emit signals – do not forget it in your own classes.
class AngleObject : public QObject
{
    Q_OBJECT
    Q_PROPERTY(qreal angle READ angle WRITE setAngle NOTIFY angleChanged)

public:
    AngleObject(qreal angle, QObject *parent = 0);
    qreal angle() const;

public slots:
    void setAngle(qreal);

signals:
    void angleChanged(qreal);

private:
    qreal m_angle;
};
void AngleObject::setAngle(qreal angle)
{
    if(m_angle == angle)
        return;

    m_angle = angle;
    emit angleChanged(m_angle);
}
Lab 1 Notes

- Use QImage or QPixmap for rendering the image
- Override QMainWindow::keyPressEvent for keyboard event handling
- Keep track of old mouse position to determine paint stroke direction
- For 2D image patch transformations, use QImage::copy and QImage::transformed