All elemental steps that will get you started for your new life as a computer science programmer.
Week E – GUIs with Qt

What is Qt?

- Pros and Cons

What does Qt do with my code?

- Qt Designer, .ui and .moc files
- Signals and slots

Examples

- MOC and UIC
- Make, CMake, QMake

Week E – GUIs with Qt

This week we’ll talk a little bit about Qt and its capabilities to create nice GUIs. As you might know, Qt comes with a lot of extra feature and is far more than just another GUI library. A slight drawback is that Qt needs to influence your whole build process because it introduces non C++-conform syntax in your code. Before the actual compiler starts parsing, Qt itself parses your files and automatically creates C++ conform code.
What is Qt?

“Qt is a cross-platform application and UI framework” - [http://qt-project.org/](http://qt-project.org/)

Comes with a ton of extras

- Has its own build system (qmake)
- Now also packed in a single IDE (Qt Creator)

Qt is very complex, but has a great documentation

- Make use of it!
- Online (use google) or offline (Qt Assistant)

It is good to know that Qt provides everything for your programming needs: Its own build system (qmake), its own IDE (Qt Creator) and its own UI Designer Program (Qt Designer). It is even better to know that you do not have to use them if you don’t like to. You can use Cmake and your own favorite IDE/Editor if you prefer.

As there are a lot of resources to learn Qt, it may be difficult to pick one good tutorial to get started. For the exercise material, I combined several tutorials that can be useful for you, too:

Learn Qt - [http://thelins.se/learnqt/tutorial/](http://thelins.se/learnqt/tutorial/)

Qt Getting started - [http://qt-project.org/doc/qt-5.1/qtdoc/gettingstarted.html](http://qt-project.org/doc/qt-5.1/qtdoc/gettingstarted.html)

(Für Qt 4) Zetcode Qt4 tutorial - [http://zetcode.com/gui/qt4/](http://zetcode.com/gui/qt4/)
Qt – Pros and cons

Qt is ...

- Powerful
- Big
- Filled with useful extras
- Has excellent documentation

But ...

- Extends C++ standard
- You might not need everything
Qt introduces several new concepts to standard C++. Communication inside and between Qt-Classes is achieved with a signal and slot mechanism. For Qt 4 you can find a good introduction here [http://qt-project.org/doc/qt-4.8/signalsandslots.html](http://qt-project.org/doc/qt-4.8/signalsandslots.html).

Careful! The new Qt 5 version changed the syntax of signals and slots ([http://qt-project.org/wiki/New_Signal_Slot_Syntax](http://qt-project.org/wiki/New_Signal_Slot_Syntax)).

---

**Qt – So what’s different?**

**Signals and slots**
- Classes can communicate with SIGNALS and SLOTS, these can be queried/connected at runtime

**Code Generators**
- Qt-specific stuff is translated to standard C++
- UIC: gui-xml to C++ headers
- MOC: Qt-enhanced C++ (usually headers) to C++
- RCC: Resources into binaries
Counter.h

class Counter : public QObject
{
    Q_OBJECT    // Important!!
private:
    int m_value;
public:
    Counter() { m_value = 0; }
    int value() const { return m_value; }
public slots: // Notice the new C++ keyword
    void setValue(int value);
signals:      // Notice the new C++ keyword
    // This function is not implemented in this class!
    void valueChanged(int newValue); // not implem
};

Counter.cpp

void Counter::setValue(int value)
{
    if (value != m_value) {
        m_value = value;
        emit valueChanged(value); // Emit a signal ...
    }
}

Basic Makefile to get started with Qt

```makefile
INCPATH = -I/usr/include/qt4/QtGui -I/usr/include/qt4
LIBPATH = -L/usr/lib/

OPTIONS = -lQtCore -lQtGui
CCFLAGS = -Wall -g
EXE=qt_app

$(EXE): main.o
    g++ $(LIBPATH) $(OPTIONS) $^ -o $@

main.o: main.cpp
    g++ $(INCPATH) $(CCFLAGS) -c $<

clean:
    rm -f *.o *~ $(EXE)
```

In order to compile your first Qt application, you just need to include the Qt headers and link to the needed libraries. Notice that the libraries depend on the classes you use (often times you need QtCore, QtGui and QtOpenGL). Also note that the libraries might change between major Qt version (e.g. for Qt 5 you need to also link against QWidgets).
It’s important to have a QApplication object. After you set up your application in the main method, you call QApplication::exec() to enter the Qt main loop.
If you are implementing your own Qt Class, e.g. if you derive a class from QWidget and use your own slots/signals, you have to include the Q_OBJECT keyword right after class definition. The official Qt website says:

*The Q_OBJECT macro must appear in the private section of a class definition that declares its own signals and slots or that uses other services provided by Qt’s meta-object system.*

Then you can use the extra keywords slots, signals and emit.

Before you call the C++ compiler, you have to tell the Qt Meta Object Compiler (MOC) to convert your code into standard C++ code.

```cpp
class Communicate : public QWidget
{
    Q_OBJECT

public:
    Communicate(QWidget *parent = 0);

public slots:
    void OnPlus();
    void OnMinus();

private:
    QLabel *label_number;
};

Please note that no “special” syntax is used in .cpp file
```
When deriving a class from QObject, you have to run the MOC to get some meta information about the class. MOC parses the header file and create C++ code that lets you query information about the object, i.e., its name, the position and signature of slots. Typically, these files are named

- `moc_originalname.cpp`
- `originalname.moc.cpp`

or

- `originalname.moc.cxx`

They have to be compiled and linked like any other .cpp file.
01_moc – Makefile

$(EXE): main.o moc_communicate.o communicate.o
g++ $(LIBPATH) $(OPTIONS) $^ -o $@

main.o: main.cpp
  g++ $(INCPATH) $(CCFLAGS) -c $<

moc_communicate.o: moc_communicate.cpp
  g++ $(INCPATH) $(CCFLAGS) -c moc_communicate.cpp

moc_communicate.cpp: communicate.h
  moc $(INCPATH) $< -o $@

communicate.o: communicate.cpp
  g++ $(INCPATH) $(CCFLAGS) -c communicate.cpp

clean:
  rm -f *.o *~ $(EXE) moc_*cpp

This is the modified Makefile, the biggest difference is the invocation of MOC to create the file moc_communicate.cpp.

Please note that both moc_communicate.o and communicate.o have to be compiled and linked.
This is the content of a .moc-file. The file is changed every time you modify and moc the source header or .cpp file and therefore, it is a waste of time to edit the .moc-file by hand. As a hint for moc-files: Simply acknowledge their existence, do not forget to link them but never take any closer look at them 😊
The first step in processing a Qt application is usually to start the Qt designer and create your own user-interface layout with a few clicks. If you already have experience with GUI layouts it should be quite self-explanatory. If not, use the documentation. Your interface is stored in special Qt files, the .ui files.
Hand-coded GUI

```cpp
setWindowTitle("Communicate");
int WIDTH = 350;
int HEIGHT = 190;
resize(WIDTH, HEIGHT);
QPushButton *plus = new QPushButton("+", this);
plus->setGeometry(50, 40, 75, 30);
QPushButton *minus = new QPushButton("-", this);
minus->setGeometry(50, 100, 75, 30);

label_number = new QLabel("0", this);
label_number->setGeometry(190, 80, 20, 30);
```

You can create the contents and (very simplistic) layouts of a GUI by hand. But don’t do it! Be more productive by using an IDE!
Create your GUIs using a graphical editor, the Qt Designer. The designer is already integrated in the Qt Creator and both can be used comfortably from a common IDE.
This is the content of a .ui-file. Similar to the .moc files, do not manually change the file, but use the Qt Designer for it.
UIC : .ui to .h

Convert your GUI into a proper C++ class

“uic communicate.ui -o ui_community.h”

Inherit from the newly created class

Use the object names of the. ui file as actual class/function names
To use a UI, you derive from the created UI class and then setup the UI using the setupUI method passing the instance in the constructor. Make sure, that you always call the method before you use the UI components.

Alternatively, you can make your class store a member of the UI-class instead of deriving from it. You still need to setup the member in the constructor.
Of course, the UIC call has to be integrated into the Makefile.

More infos on the MOC and UIC can be found (and should be read) at:

http://qt-project.org/doc/qt-4.8/moc.html

http://qt-project.org/doc/qt-4.8/uic.html

---

$\text{EXE}$: main.o moc_communicate.o communicate.o
  \texttt{g++ $(LIBPATH) $(OPTIONS) \$^ \text{-o} \$@}

main.o: main.cpp \texttt{ui\_communicate.h}
  \texttt{g++ $(INCPATH) $(CCFLAGS) -c \$<}

moc_communicate.o: moc_communicate.cpp
  \texttt{g++ $(INCPATH) $(CCFLAGS) -c moc\_communicate.cpp}

moc\_communicate.cpp: communicate\_h \texttt{ui\_communicate.h}
  \texttt{moc $(INCPATH) \$< \text{-o} \$@}

communicate.o: communicate.cpp \texttt{ui\_communicate.h}
  \texttt{g++ $(INCPATH) $(CCFLAGS) -c communicate.cpp}

\texttt{ui\_communicate.h}: communicate\_ui
  \texttt{uic \$< \text{-o} \$@}
Qt – Building your application

Usual way: qmake (using Qt Creator)

Alternative - CMake

- Keywords QT4_WRAP_UI(), QT4_WRAP_CPP()

Avoid Makefiles

- Not recommended to call MOC and UIC manually
PROJECT(QtApp)
[...]
FIND_PACKAGE(Qt4 REQUIRED)
INCLUDE_DIRECTORIES(${QT_INCLUDE_DIR})
  ${QT_INCLUDE_DIR}/QtCore
  ${QT_INCLUDE_DIR}/QtGui
  ${QtApp_SOURCE_DIR}
)
LINK_DIRECTORIES(${QT_LINK_DIRECTORIES})
include(${QT_USE_FILE})
[...]
QT4_WRAP_UI(app_UIS_H ${app_UIS})
QT4_WRAP_CPP(app_MOC_SRCS ${app_MOC_HDRS})
INCLUDE_DIRECTORIES (${CMAKE_CURRENT_BINARY_DIR})
ADD_EXECUTABLE(qt4_app ${app_SRCS} ${app_UIS_H}
  ${app_MOC_SRCS})
TARGET_LINK_LIBRARIES(qt4_app ${QT_LIBRARIES})
Use qmake from the terminal:

1. qmake -project
2. qmake
3. make
05_advanced

Play around with application

- Try changing SIGNAL / SLOT connections
- Modify the *.ui file
- Try building it with qmake

Extend the application

- See task sheet for some ideas
Assignment E: GUls with Qt

Task 1: Clone Brush [50 points]

First of all, build the example application (05_advanced) and make it run on your machine.

Then proceed and add two new features:

Extend the program in a way that you do have two images instead of one (so you need to have two instances of DrawingCanvas. Add the functionality that you can select a region in one of the two images (e.g., by clicking CTRL+Left Mouse Button) and then copy pixels from this spot to the other Drawing Canvas.

Two Drawing Canvas Instances. The clone brush is used to copy content into the left image.
Task 2: Color Triangle [50 points]

Take a look at the class QtColorTriangle class in the folder (06_qtcolortriangle).

You can find more details on the class interface here:


Create a project file in the Qt Creator for the files in the folder and compile the simple demo program. You should see the color triangle window.

In the next step, try to build a dynamic library that includes the QtColorTriangle class.

Afterwards add the widget to your application and use the color wheel to pick colors other than blue or red.

Also see how you can import the color wheel into your Qt Designer environment.