CSCI 104
Qt Intro

Mark Redekopp
David Kempe
What is QT?

- Pronounced “cute”
- An cross-platform application development framework built by Nokia
- A toolkit for building Graphical User Interfaces (GUIs)
- GUI toolkits are composed of many classes including many widgets
  - "Widget" is GUI-lingo for a 'control' or graphical component that a user can interact with

QT has bindings available for many languages
- C++, Python, Ruby, Java, etc.

We are using QT v4.8.1
**QApplication**

- Every major QT widget has its own header
  - See QPushButton in the example
- QApplication
  - The main class that controls all the default GUI behavior and manages application resources
  - Every QT GUI application **must** have a QApplication instance (**and only one!**)  
  - QApplication parses the command line input and pulls out any display-related parameters
  - A QApplication must be created **before** any GUI-related features can be used

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

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

- QPushButton
  - A button object that you can click on
- QPushButton button("Hello World!");
  - Creates a clickable button on the GUI
  - We can only do this now that we already created a QApplication to handle all the backend stuff
  - The button is clickable just by nature
  - The button will have the text “Hello World” on it
  - There are all kinds of button function/display attributes we could set if we really wanted to
    - Color, Size, Text/Image, Animation, Border, etc.

#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QPushButton button("Hello world!");
    button.show();
    return app.exec();
}
Display Widgets

- `button.show();`
  - Widgets are always invisible by default when they are created, you must call `show()` to display them
  - Calling `show()` on a widget also calls `show()` on all the widgets it contains (all of its children)
- Some widgets are merely containers for other widgets (i.e. a display grid that display other widgets in some tabular format)

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

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QPushButton button("Hello world!");
    button.show();
    return app.exec();
}
```
Event-Driven Program Flow

- `return app.exec();`
  - At this point, `main()` passes control to the QT framework
  - `exec()` will not return until the window is terminated
- **Question?**
  - What happens to your code flow?
  - How do you get any other code to run?
  - Welcome to the world of event-driven programs
    - You write code (member functions) that is 'automatically' called/executed when an event occurs (e.g. `click()`, `resize()`, `mouseOver()`, ...)
  - More on this later...

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

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QPushButton button("Hello world!");
    button.show();
    return app.exec();
}
```
End Result

• All of this results in...

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

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QPushButton button("Hello world!");
    button.show();
    return app.exec();
}
```
Compiling Qt Applications

- We can't just type 'g++ -o qtex qtex.cpp'. Why?
  - We have external dependencies that aren't part of standard C++
  - How will the compiler find the QT .h files?
  - How will the linker find the QT compiled code?
  - QT has to build Meta-Objects to handle communication between GUI pieces
  - The individual .cpp files need to compile and link separately in some cases

- 'make' and 'qmake' to the rescue
  - We've seen 'make' which helps us specify dependencies, compile order, and compiler commands
  - 'qmake' will examine code in the current directory and help to automatically generate a 'Makefile'
3-Step Qt Compiler Process

• Step 1: Generate a Qt project file with 'qmake'
  – $ qmake -project
  – The command will make Qt examine all the source code in the current directory and make a platform-independent project file (with a .pro extension) that specifies dependencies between your .h and .cpp files

• Step 2: Generate the platform dependent Makefile
  – $ qmake
  – This command will make QT read the .pro file from the current directory and generate a Makefile that contains all the commands for compiling the code and linking it with the QT libraries

• Step 3: Run 'make'
  – $ make
  – If you have any compiler or linker errors, this is the step in the process where you will see them
  – If you only need to recompile, you only need to use this particular step of the 3 step process
Qt Compilation Notes

• Keep each project in a separate directory (this is why we can run qmake with no arguments)
• If you add new .h or .cpp files, you need to re-run the entire compilation process (i.e. Make new .pro and Makefile files again)
• If your object needs slots or signals, then you MUST put it into separate .h and .cpp files
• If you're getting weird linker errors, try make clean or try rebuilding the .pro file and the Makefile
• You may notice that when you compile some projects with QT, it actually generate extra .cpp files
  – These extra files are generated by QT's moc (Meta Object Compiler)
  – QT makes extensive use of the preprocessor to generate code that makes things like its signals and slots mechanisms work
  – Don't bother changing these files. They'll just get overwritten next time you compile.
Qt Organization

• For your programming purposes, the QT windowing framework consists of three major parts (in reality, it's MUCH more complicated than this):
  – Widgets
  – Layouts
  – Signals & Slots
Qt Widgets

• What is a widget?
  – A user interface object that can process input, emit signals and draw graphics
  – A widget can be styled to have a vastly different appearance than its default
  – Most widgets generate signals that can be received by pieces of your code called slots

• QT comes pre-packaged with a ton of pre-made widgets to suit most of your GUI-building needs
  – Buttons, Containers, Menus, etc.
Qt Button Examples

Push Buttons

Tool Buttons

Checkboxes

Radio Buttons

Container Examples

Group Boxes

Tabbed Displays

Frames

Scrolled Displays

User Input Widget Examples

- Text Entry
- Combo Boxes
- Sliders
- Spin Boxes
- Calendars

Qt Layouts

• What is a layout?
  – A layout describe how widgets are organized and positioned in a user interface

• The jobs of a QT layout
  – Positioning of widgets in GUI
  – Choosing sensible default and minimum sizes
  – Handling window resize events
  – Automatic updates when content changes
    • Font size, text or other widget changes
    • Add or removal of new widgets
    • Showing and hiding of existing widgets
More About Layouts

- QT layouts and widgets share numerous parent/child relationships
  - **Widgets** can contain other **widgets** (usually in a layout)
  - **Widgets** can have one primary **layout** (which may contain many other child layouts)
  - **Layouts** can contain **widgets**
  - **Layouts** can contain other **layouts**
  - There can be a gigantic graph of parent and child relationships in a GUI

- The best way to make a complex layout is usually to combine many simpler layouts

- FYI: Getting a layout right is **HARD**
Sample Layouts

• QVBoxLayout
  – Layout all children in a vertical column
  – (top to bottom or bottom to top)

• QHBoxLayout
  – Layout all children in a horizontal row
  – (left to right or right to left)

# Layout Example Code

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

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QWidget *window = new QWidget;

    QPushButton *button1 = new QPushButton("One");
    QPushButton *button2 = new QPushButton("Two");
    QPushButton *button3 = new QPushButton("Three");

    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(button1);
    layout->addWidget(button2);
    layout->addWidget(button3);

    window->setLayout(layout);
    window->show();
    return app.exec();
}
```

More Layouts

• QGridLayout
  – Layout widgets in a 2D grid
  – Widgets can span multiple rows/columns

• QFormLayout
  – Layout children in a 2-column descriptive label-field style.

Event-Based Programming

• GUI-based programs follow a different paradigm than basic command line programs
  – The window will sit there indefinitely until the user does something
  – Your code no longer functions on line-by-line flow, it is triggered by events
• In QT, all widgets are capable of firing events and receiving events
  – **Signals** are used to **notify (emit)** widgets of an event
  – **Slots** are used to **receive (listen for)** widget events
  – connect is used to tie together a signal & a slot
  – Signals & slots can have M-to-N connections
Qt Signals and Slots

- **Signals** and **Slots** provide communication between various object in your application
  - Often when one widget changes, you need another widget to know about it
- A **signal** emitter and a **slot** receiver never need to know about each other!
  - Widgets emit signals whether or not any other widgets are listening
    - e.g. `QPushButton` has a `clicked()` signal
  - Widgets slots listen for signals whether or not there are any being emitted
    - A slot is just a normal class member function!
    - e.g. Create a widget with a `handleClick()` slot
QT Signals & Slots

connect( Object1, signal1, Object2, slot1 )
connect( Object1, signal1, Object2, slot2 )
connect( Object1, signal2, Object4, slot1 )
connect( Object3, signal1, Object4, slot3 )
connect( Object2, signal1, slot1 slot2 )
connect( Object4, slot1 slot2 slot3 )
# Qt Signal/Slot Example

```cpp
#include <QApplication>
#include <QPushButton>
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QPushButton button("QUIT");

    //connect(object1 pointer, object1 signal,
    //         object2 pointer, object2 slot)
    QObject::connect(&button, SIGNAL(clicked()),
                     &app, SLOT(quit()));

    button.show();
    return app.exec();
}
```

QT Signals & Slots Summary

• Using event-driven programming in QT involves three major parts:
• 1. A widget with a **SIGNAL** to emit events when they occur (e.g. `clicked()` on QPushButton)
• 2. A widget with a **SLOT** to receive events that have been emitted (e.g. `quit()` on QApplication)
• 3. A **connect** statement to wire the signal and slot together so that when the signal is emitted, the slot receives it
Qt Tutorial

- A set of 14 example QT tutorials can all be found online here:

- Official? Qt Page
  - http://qt-project.org/doc/qt-4.8/

- Other resources
NEXT PART
Examples

• On your VM
  – Do a pull on your homework-resources repo and look for the qtex folder
  – OR
    • $ mkdir qtex
    • $ cd qtex
    • $ wget http://ee.usc.edu/~redekopp/cs104/qtex.tar
    • $ tar xvf qtex.tar

• 4 examples
  – Reflex (signals & slots)
  – Formex (Form example)
    • Inheritance...deriving new widgets
    • Layouts
  – Lec_ttt (Tic-Tac-Toe example)
  – Multiwin (Multi window example)
Reflex

• Hammer defines a signal function
  – A signal is a function that has no body
  – When you "call"/"emit" it, it will trigger other "connected" functions to be called
    • emit hit(hard)

• Knee defines a slot function
  – A slot function must match the prototype of the signal function that it will be connected to
  – You can do whatever you want in this function

• You must connect signals to slots via connect()
  – See reflex.cpp

• You can have multiple slot functions connected to 1 signal
  – Exercise: in reflex.cpp declare another 'knee' and connect it's reflex to the hammer's signal
Formex

• This program provides QLineEdit textboxes and buttons to prompt the user for their name and age and then saves that data out to a text file named 'data.txt'
• Think about layouts as tables within other tables
• http://doc.qt.io/qt-4.8/widgets-and-layouts.html
Layouts

- Four different layouts are commonly used
  - QVBoxLayout
  - QHBoxLayout
  - QFormLayout
  - QGridLayout

- Each widget (or derived class) can have only one Layout
  - Set by calling: widget->setLayout(pointer to the layout) method

- But a layout may contain either widgets or OTHER LAYOUTS in each of its entries
  - Set by calling: layout->addLayout(pointer to child layout)
  - Set by calling: layout->addWidget(pointer to the child widget)

- So for each widget think about whether you want to add items vertically or horizontally and pick a Vbox or Hbox Layout and then add child layouts within that context
More Notes

- Widgets have a virtual function `sizeHint()`
  - `QSize sizeHint() const;`
  - If you want your widget to start at a particular size, add this to your class and simply have it return a `QSize` object which is just pixel rows x columns
  - `QSize MYCLASS::sizeHint() const { return QSize(400, 400); }`

- Defining your own signals
  - Signals go in the "signals:" section of your class
  - They are just prototypes (you don't write an implementation)
  - Use the 'emit' keyword followed by a "call" to this signal function
  - Whatever slot has been connected to this signal will in turn be called

- Events are not slots (think of them as "slots" that are pre-connected to certain actions/signals)
  - Just override them and usually call the BaseClass version
Tic-Tac-Toe Example

- $ cd lec_ttt
- Look up instructions on the 3 steps from our previous Qt lecture to setup and build/compile the project
Overall structure

• TTTButton models a single square in the grid and contains its type: Blank, Circle, Cross
• TTTBoard models the NxN tic-tac-toe grid
• TTT models the other controls of the game and UI
TTTButton

• Is a derived PushButton
• TTTButton models a single square in the grid and contains its type: Blank, Circle, Cross
  – setType() calls repaint()
  – Repaint() triggers paintEvent() which TTTButton overrides
• Examine TTTButton::paintEvent()
  – What if we don't call the base class version or change the ordering?
Q_OBJECT macro

• Helps Qt preprocessor define the .moc files (meta-objects)
  – If your class derives from a Qt widget/other GUI control or uses signals and slots you should place it in the definition

• Declare on a line (w/o a semicolon to follow)
TTTBoard

- Is derived from QWidget (because it contains other widgets, receives user input, and needs to be drawn/painted)
- Stores the TTT buttons and implements the move AI and win/lose/draw logic
- Examine GridLayout component which controls the display of the tic-tac-toe grid
- finished() signal (no definition)
  - Signals have no definitions in a .cpp file
  - Notice the emit statement in
- Connecting the clicks on buttons via buttonClicked
  - Notice the many-to-one relationship of TTT_Button::clicked() to TTT_Board::buttonClicked()
  - Look at buttonClicked() how do we determine which button was actually clicked?
- updateButtons
  - Notice setEnabled() call...What does that do?
• Models the overall UI and main window
• Is derived from QWidget (because it contains other widgets, receives user input, and needs to be drawn/painted)

QVBoxLayout
  – Each widget is added via addWidget and gets slotted vertically

QLabel: On screen text

QComboBox
  – Items have an ID and a display string usually
  – Selected value from the user can be obtained with currentIndex()

QPushButton
  – Notice we connect the signals and slots (some from TTT_Board, others from ourselves (i.e. TTT) )

newState() controls the string printed on the status label
main

• Instantiates a TTT widget and shows it (then enters the execution loop).
WIDGET REFERENCE
Overview

• The following slides represent a few commonly used widgets and some of their useful functions and signals

• Recall: A SLOT function can be called anytime as a normal function OR it can be connected as a SLOT (OR both)

• The online documentation for the Qt library is THE source to go to. Either google your widgets name or go here: http://qt-project.org/doc/qt-4.8/
QLineEdit

• Provides a generic text box functionality
• Helpful methods
  – text()
    • Returns a QString of the text currently written in the textbox
    • Can convert a QString to a C++ string using toStdString()
  – setText(QString)
    • Changes the text displayed in the textbox to the argument provided
  – clear()
    • Deletes the text currently in the box
QComboBox

• Provides a DropDownBox functionality (list of items that can be displayed when you click the down array and then 1 item can be selected)

• Helpful methods
  – currentText()
    • Returns a QString of the selected item's text
  – addItem(QString)
    • Adds the string argument to the list of items to be displayed in the drop down box

• Useful Signals that you can connect to
  – currentIndexChanged(QString)
    • This signal will be emitted whenever a new item is selected in the drop down box...It will pass the text string of the newly selected item
QListWidget

- Provides a scrollable list of selectable text items

**Helpful Methods**

- `clear()`
  - Removes all the items in the list
- `insertItem(int pos, QString str)`
  - Adds the text item, str, at position, pos, in the list
- `currentItem()`
  - Returns a `QListWidgetItem*` of the currently selected item
- `item(int row)`
  - Returns a `QListWidgetItem*` of the item on the row given by the argument

**Helpful signals**

- `itemClicked(QListWidgetItem* item)`
  - Will call connected SLOT functions whenever an item is clicked in the QListWidget and pass a pointer to the QListWidgetItem that was clicked.
  - You can retrieve the text of the clicked item by calling "item->text()"

**Other helpful functions**

- `itemDoubleClicked()`, `removeItemWidget()`, `indexFromItem()`
QPushButton

• Push/Command button functionality
• Helpful signals:
  – clicked()
    • Will call associated SLOT functions when clicked
QRadioButton

• Implements a 1-of-n selection...each radio button has an automatically associated text label to help the user
• All radio buttons with the same parent widget (usually a layout) will be mutually exclusive (only 1 can be on)
• Usually grouped radio buttons should be in a QGroupBox
  – setChecked(bool val)
    • Sets the radio button value to 'val' (true = on, false = off)
QGroupBox

- Provides a visual grouping of widgets in a boxed frame with a title
  - Title is the argument to the constructor of the QGroupBox
- Make a layout with everything you want to be in this framed area and then set the layout
  - QGroupBox* gb = new QGroupBox("Your Title")
  - // make a layout with all widgets you want in the framed area
  - gb->setLayout(your_layout);

http://qt-project.org/doc/qt-4.8/qgroupbox.html
QFormLayout

- Remember QFormLayout adds a text label and an arbitrary widget in a row-based layout
Other Useful Controls

• QCheckBox
  – Similar to radio buttons but without the restriction of 1-of-n being selected (many can be selected at a time)

• QTextEdit
  – For displaying multi-line text with auto line-wrapping, etc.