Qt Fundamentals

Qt Basics
Qt Basics - Outline

- Here we will cover the basic Qt concepts
  - Basic types
  - Memory management
  - QObject base class
  - Parent/Child relationship
  - Signal/Slot mechanism
Basic Types & Qt Core Classes

- Basic data types are normal C++ data types
  - int, bool, double, char etc.

- Structs and arrays are created normally

- Qt variants of most of the containers in the C++ Standard Library
  - QList<T>
  - QVector<T>
  - QMap<T,C>
  - etc.

- For string types, Qt holds its own type: QString

Qt contains versions of almost all of the standard library - some say with a friendlier API
Basic Types & Qt Core Classes

- **QDate, QDateTime** – Can be compared, converted to strings
- **QChar** – 16-bit Unicode character
- **QString** – Unicode character string. Can be resized, may contain 8-bit \0 terminating strings or binary data
- **QByteArray** - Used instead of QStringList, when memory conservation is important (Qt for embedded Linux)
- **QEventLoop, QEvent** – Used to enter and exit event loop
- **QHash** – Template providing a hash-table-based dictionary
Basic Types & Qt Core Classes

- **QQueue** – Template class implementing a FIFO queue
- **QPoint, QRect** – Rectangle is defined using the top left and bottom right
- **QTimer** – One shot or periodic 1 ms timer (accuracy depends on platform)
- **QVariant** – Union of the common Qt types
- **QVector** – Template class for dynamic arrays (flat), QLinkedList more efficient, if many insertion and deletion operations needed
- Iterator classes – Java (QVectorIterator) and STL-like (QVector<T>::iterator) iterators exist
Creating Your Own Qt Application

- **Widget**
  - A UI building block, base class QWidget
    - Label
    - Text editors
    - Empty window
    - Main window
    - Buttons
    - etc.

- Often your own application UI is a widget of your own which consists of multiple inner widgets
Quick Widget Overview

**Input Widgets**
- Text Edit
- Spin Box: 0
- Double Spin Box: 0.00
- Time Edit: 12:00 AM

**Item Widgets**
- Group Box
- Push Button

**Container Widgets**
- MainWindow

**Display Widgets**
- Hello Qt

**Main Windows**
- MainWindow

**Buttons**
- Push Button
- RadioButton
- CheckBox
- Command Link Button

**Dialogs**
- Dialog
  - What to say: hello
  - This is a group box
  - What a cool dialog you have here!
#include <QtGui>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel *label = new QLabel("this is great");
    label->show();

    return app.exec();
}
QApplication - the core of our app.

- Initializes application settings
  - Palette, font
- Defines application’s look and feel
- Provides localization of strings
- Knows application’s windows

- Derived from QCoreApplication [QtCore]
  - Used in **console applications** (or Qt processes without any UI, servers for instance).

- Performs event handling, receives and dispatches events from the underlying window system
More Widgets?

```cpp
#include <QtGui>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QLabel *labelOne = new QLabel("this is great");
    labelOne->show();

    QLabel *labelTwo = new QLabel("it just works");
    labelTwo->show();

    return app.exec();
}
```

- Not exactly what was intented..
- Each widget **without** a parent becomes a windows of its own
QObject Class Role

● Heart of Qt's object model
  ○ Base class for all object classes
  ○ So, all QWidgets are QObjects also
  ○ Provides object trees and object ownership
  ○ QObject's responsibility is to provide a central location for the most important concepts in Qt

● Has three major responsibilities
  ○ Memory Management
  ○ Introspection (runtime identification of object types)
  ○ Event handling
Parent/Child Relationship

- Each QObject instance may take a parent argument
- Child informs its parent about its existence, upon which the parent adds it to its own list of children
- If a widget object does not have a parent, it is a window
- The parent does the following for its children:
  - Hides and shows children, when hidden/shown itself
  - Enables and disables children when enabled or disabled itself
- Note that a child may be explicitly hidden, although the parent is shown
Memory Management

- The ownership of all child QObjects is transferred to the parent
  - *Automatic deletion* by the parent
  - Allocated from the *heap* with *new*
  - Manual deletion won’t however cause *double deletion* because the child informs its parent of the deletion

- All QObjects without a parent must be deleted manually

- Occasionally it may seem like Qt would hold some sort of automatic garbage collection but this is not true!
  - Always pay attention to ownerships and responsibilities!
Creating Objects

- Objects inheriting from QObject are allocated on the heap using **new**
  - If a parent object is assigned, it takes ownership of the newly created object – and eventually calls delete

  ```cpp
  QLabel *label = new QLabel("Some Text", parent);
  ```

- Objects not inheriting QObject are allocated on the stack, not the heap
  - QStringList list;
  - QColor color;

- Exceptions
  - QFile and QApplication (inheriting QObject) are usually allocated on the stack
  - Modal dialogs are often allocated on the stack, too
Another Try

```
#include <QtGui>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QLabel labelOne("this is great");

    QLabel *labelTwo = new QLabel("it just works", &labelOne);
    labelOne.show();

    return app.exec();
}
```

- Nearly, but not quite enough..
  - Let's try with layouts
Final Try, Now With a Layout

```cpp
#include <QtGui>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QWidget window; // Needed as a layout cannot be a window
    QVBoxLayout *layout = new QVBoxLayout(&window);

    layout->addWidget(new QLabel("this is great"));
    layout->addWidget(new QLabel("it just works"));
    layout->addStretch();

    window.show();
    return app.exec();
}
```
Another Example, Anything Wrong Here?

```cpp
#include <QtGui>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QLabel label("Testing");
    QWidget window;

    label.setParent(&window);
    window.show();

    return app.exec();
}
```
Conclusions So Far

- Widgets are put inside a main widget
  - Main widget becomes a parent for its child widgets
  - No garbage collection, ownership transferships!
- Layout classes are used for non-hard-coded positioning

- But… Do we really need to code everything manually?
  - No, we can use Qt Designer
Qt Designer

- Comes with Desktop Qt as a separate application
  - Also integrated to QtCreator and Carbide.c++
- Drag’n’drop UI designer
- Outputs XML-based .ui files
- .ui files are converted automatically to .h files by uic (ui compiler)
  - myproject.ui -> ui_myproject.h
Do I create my own "main widget"?

- No Qt has just what you need
  - QMainWindow - used by most application user interfaces
  - Easy creation of menus and stuff :)

Let's take a look in Qt Creator:
- File -> New File or Project
  - Select Qt C++ Project and Qt Gui Application
  - Qt Creators project wizard will generate a project with ui forms and everything.
Meta-Object System

- Meta-object system extends C++ with dynamic features – similar to those in Java, for example
- Dynamic features include
  - Mechanism to access any functions in the class
    - Also private ones
    - Used by **signals and slots**
  - Class information
    - Type without RTTI (Run-Time Type Information)
    - Information about base classes
  - Translate strings for internationalization
  - Dynamic properties
Meta-Object System - Example

• A simple class declaration
• Simple QObject subclass
• One slot
• One signal

```cpp
class MyObject : public QObject
{
    Q_OBJECT

public:
    MyObject(QObject *parent = 0);

public slots:
    void mySlot();

signals:
    void mySignal();
};
```
Signals and Slots

- Observer pattern
- Type-safe callbacks
- More secure than callbacks, more flexible than virtual methods
- Many-to-many relationship
- Implemented in QObject
Signals

- A signal is a way to inform a possible observer that something of interest has happened inside the observed class
  - A QPushButton is clicked
  - An asynchronous service handler is finished
  - Value of QSlider is changed

- Signals are *member functions* that are automatically implemented in the meta-object
  - Only the function declaration is provided by the developer

- Signal is sent, or *emitted*, using the keyword emit
  - emit clicked();
  - emit someSignal(7, "Hello");
**Slots**

- A slot is a function that is to be executed when a signal has been emitted.
  - (When QPushButton is pressed), close QDialog
  - (When service is ready), ask for the value and store it
  - (When QSlider value is changed), show a new value in QLCDNumber
- A Slot function is a **normal member function** implemented by the developer
Signals and Slots

The signals and slots mechanism is fundamental to Qt programming. It enables the application programmer to bind objects together without the objects knowing anything about each other.

- Slots are almost identical to ordinary C++ member functions.
  - The main difference is that they can be connected to a signal - in which case it is automatically called when the signal is emitted.
Signals and Slots

- To setup the signal-slot connection, we must first define the connection.
- The `connect()` statement looks like this:

```cpp
connect(sender, SIGNAL(signal), receiver, SLOT(slot));
```

```cpp
int main(int argc, char *argv[]) {
    QApplication a(argc, argv);

    QPushButton button("Quit");
    button.resize(150,150);
    QObject::connect(&button, SIGNAL(clicked()), &a, SLOT(quit()));
    button.show();

    return a.exec();
}
```
Signals and Slots

Main features:

- One signal can be connected to many slots

```cpp
connect(slider, SIGNAL(valueChanged(int)),
        spinBox, SLOT(setValue(int)));
```

```cpp
connect(slider, SIGNAL(valueChanged(int)),
        this, SLOT(updateStatusBarIndicator(int)));
```

- Many signals can be connected to the same slot

```cpp
connect(lcd, SIGNAL(overflow()),
        this, SLOT(handleMathError()));
```

```cpp
connect(calculator, SIGNAL(divisionByZero()),
        this, SLOT(handleMathError()));
```

- A signal can be connected to another signal

```cpp
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SIGNAL(updateRecord(const QString &)));```

- Connections can be removed

```cpp
disconnect(lcd, SIGNAL(overflow()),
          this, SLOT(handleMathError())) ;
```
Find Signals and Slots

- Find the signals and slots defined in Qt classes in the Qt documentation.
Signals and Slots

Exceptionally if a signal has more parameters than the slot it is connected to, the additional parameters are ignored

```cpp
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)), this, SLOT(checkErrorCode(int)));```

If the parameters are incompatible, or the signal or slot does not exit, Qt will issue a runtime warning.

The signal and slot mechanism is provided via Qt's Meta-Object system.
Exercises

- Use a QPushButton instead of a QLabel in our HelloWorld application and connect its `clicked()` signal to the QCoreApplication's `quit()` slot.
- Create a QMainWindow and see if you can add a menu item that also quits the application.