

Qt Fundamentals

Qt Basics

1

Qt Basics - Outline

- Here we will cover the basic Qt concepts
 - Basic types
 - Memory management
 - QObject base class
 - Parent/Child relationship
 - Signal/Slot mechanism
 - Property system

2

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

3

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 QString, 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

4

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), QList more efficient, if many insertion and deletion operations needed
- Iterator classes – Java (QVectorIterator) and STL-like (QVector<T>::iterator) iterators exist

5

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

6

Very First Application

```
#include <QtGui>

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

    return app.exec();
}
```

7

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
- Use global qApp pointer to access QApplication instance
- 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

8

More Widgets?

```
#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 intended..
- Each widget **without** a parent becomes a windows of its own

9

QObject Class Role

- Heart of Qt's object model
 - Base class for all object classes
 - So, all QWidget are QObject 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

10

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

11

Memory Management

- The ownership of all child QObject 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 QObject without a parent must be deleted manually
 - Stack allocation is a good option to avoid problems
- 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!

12

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

```
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");
    label.show();

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

    return app.exec();
}
```

- Nearly, but not quite enough..
 - Let's try with layouts

14

Final Try, Now With a Layout

```
#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();
}
```

15

Another Example, Anything Wrong Here?

```
#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();
}
```

16

Conclusions So Far

- Widgets are put inside a main widget
 - Main widget becomes a parent for its child widgets
 - No garbage collection, ownership transfers!
- Layout classes are used for non-hard-coded positioning
- But... Do we really need to code everything manually?
 - No, we can use Qt Designer

17

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

18

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 :)

Lets 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.

19

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

20

Meta-Object System - Example

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

```
class MyObject : public QObject
{
    Q_OBJECT

public:
    MyObject(QObject *parent = 0);

public slots:
    void mySlot();

signals:
    void mySignal();
};
```

21

Meta-Object System Example

- Now, let's have a look at MOC-generated code
- Declaration in myobject.h
 - qmake generates make rules to run MOC on this class
 - During a normal build (before the actual compilation), MOC generates moc_myobject.cpp
- All generated data is POD
 - qt_meta_data_MyObject
 - qt_meta_stringdata_MyObject
 - QMetaObject MyObject::staticMetaObject
- POD stands for Plain Old Data
- Data which can be easily copied

22

Meta-Object System Example

- Furthermore, MOC generates several MyObject members
 - Declared by Q_OBJECT macro
- MyObject::metaObject()
 - Simply returns the static QMetaObject
 - QMetaObject::className() provides class name as string
- MyObject::qt_metacast()
 - Returns 'this' if passed class name is this class name
- MyObject::qt_metacall()
 - All slots, MyObject::mySlot()
- Implementations for all signals are also generated by MOC

23

Signals and Slots

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

24

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");

25

Slots

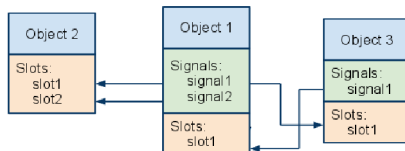
- A slot is a function that is to be executed when a signal has been emitted.
 - (When QPushButton is pressed), close QDialog
 - (When service is ready), ask for the value and store it
 - (When QSlider value is changed), show a new value in QLCDNumber
- A Slot function is a normal member function implemented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

26

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:

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

```
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();
}
```

28

Signals and Slots

Main features:

- One signal can be connected to many slots


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

connect(slider, SIGNAL(valueChanged(int)),
        this, SLOT(updateStatusBarIndicator(int)));
```
- Many signals can be connected to the same slot


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

connect(calculator, SIGNAL(divisionByZero()),
        this, SLOT(handleMathError()));
```
- A signal can be connected to another signal

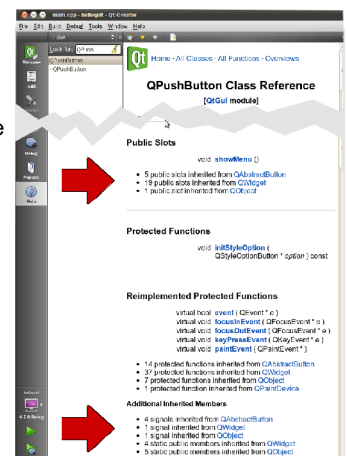

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SIGNAL(updateRecord(const QString &)));
```
- Connections can be removed


```
disconnect(lcd, SIGNAL(overflow()),
        this, SLOT(handleMathError()));
```

29

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

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

If the parameters are incompatible, or the signal or slot does not exist, 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 QApplication's *quit()* slot.
- Create a QMainWindow and see if you can add a menu item that also quits the application.