Introducing Qt

Éric Languénou

2012-2013

UNIVERSITÉ DE NANTES

Introducing Qt

(from M.Christie's slides)

- www.trolltech.com : C++ toolkit)
- many system and hardware support : one source many compilers
- Event based mechanism (signals /slots)
- Design tool ($QtDesigner$)
- Advanced support for
  - 3D ,3D graphism (OpenGL),
  - easy internationalization,
  - XML, SQL, MDI (multiple document interfaces),
  - network, threads,
  - scripting language.

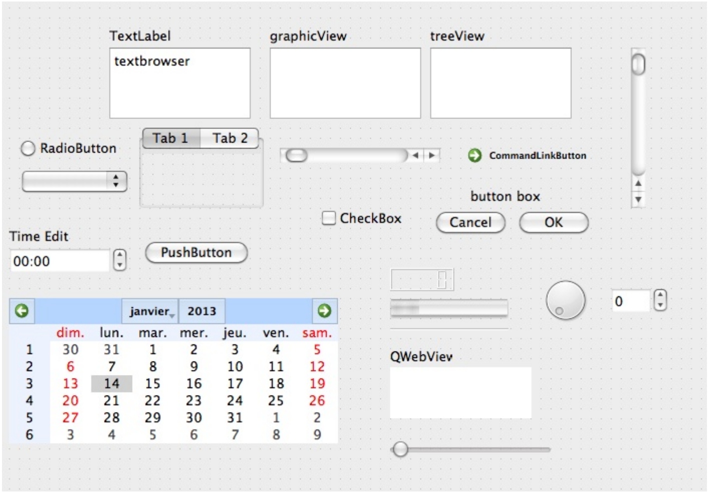
Introducing Qt (2)

- support for mobile system
- Commercial licence (Adobe, IBM, Motorola, NASA, Volvo)
- Open source community (base of KDE)
- Qt is licensed under a commercial and open source license (GNU Lesser General Public License version 2.1).
- Model-View-Controller design pattern with Qt4
  `http://doc.qt.digia.com/qt/model-view-programming.html`

Plan

- many librairies (focus on HCI in this course)
- widgets : tree organization
- event based
- compilation through meta-compiler `moc`
- componants handling by the `layout`
- design through *QtDesigner*
- Internationalisation

Widgets Samples

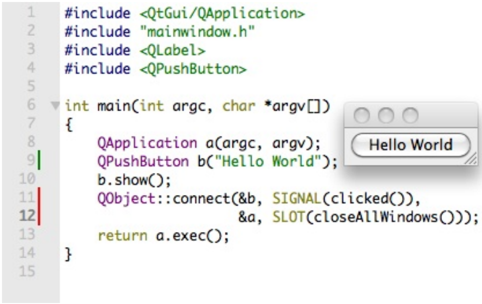


A tree organization

Objects are organized by tree and are extension of an abstract class :`QWidget`

- when an object is created, it is attached to his ancestor
- when the ancestor is destroyed the attached object are destroyed as well;
- the root is an object of type `QApplication` (derived from `QWidget`) which allows the communication between graphical and non-graphical objects;

A simple sample: main.cpp



```
#include <QtGui/QApplication>
#include "mainwindow.h"
#include <QLabel>
#include <QPushButton>

int main(int argc, char *argv[])
{
        QApplication a(argc, argv);
        QPushButton b("Hello World");
        b.show();
        QObject::connect(&b, SIGNAL(clicked()),
                                 &a, SLOT(closeAllWindows()));
        return a.exec();
}
```

An event based handling

The communication is performed through a signal/slot model;

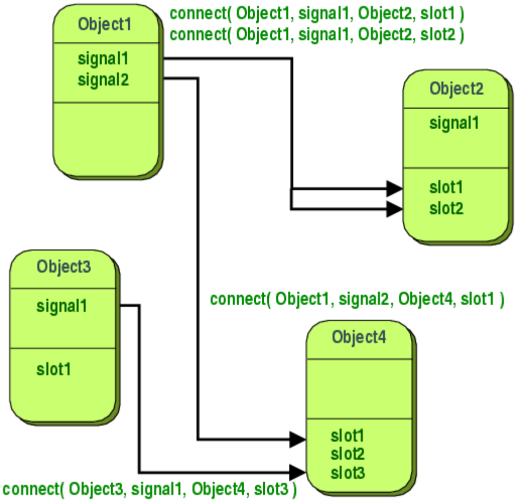
- every components (deriving from `QObject`) is allowed to send *signals*
  - signal may contain data;
  - common components have many default signalsQPushButton::clicked() or `QPushButton::stateChanged ( int state )`
- every component (deriving from `QObject`) is allowed to receive signals:
  - receiver are called *slots*
- connecting *signals* and *slots* permits the communication (function `connect()`)

An Event Based Management

- technically:
  - signals and slots are class methods;
  - when a signal method is fired, the connected slot methods are fired too (order is not guaranteed)
- avantages:
  - writing of our signal and slots within a class;
  - inheritance support

Connection

UNIVERSITÉ DE NANTES


connect(Object1, signal1, Object2, slot1)
connect(Object1, signal1, Object2, slot2)
Object1
signal1
signal2
Object2
signal1
slot1
slot2
Object3
signal1
slot1
connect(Object1, signal2, Object4, slot1)
Object4
slot1
slot2
slot3
connect(Object3, signal1, Object4, slot3)

Signals

- a signal can be emit by its class or by the derived classes;
- when a signal is fired, the connected slots are executed soon after; (no guaranty on order)
- a signal does not have a source code nor a return type;
- the various graphical components are all emitting signals, a few are connected;
- a signal emission can be forced (function `emit signal(value)`)

Slots

- are class methods as well (which could be executed out of the communication system)
- are executed on signal reception
- An object is not aware of other connected objects (this allows to write independant components)
- slots allow encapsulation :
  - `public slots` : every signals can fire these slots
  - `protected slots` : limited to its class and derived classes signals
  - `private slots` : limited to its class signals
- slots allow inheritance and polymorphic connection

Connexion

In order to establish a link between two entities the static method `connect()` is used:

```
QObject::connect( QObject *src, SIGNAL(sig),
                  QObject *dest, SLOT(slo) );
```

- `src` is the object that emit the signal sig
- `dest` is the object that receive the signal slo;
- methods `sig` and `slo` must have arguments of same types.
  It is possible to connect to signals:

  ```
  QObject::connect( QObject *src, SIGNAL(sig1),
                    QObject *dest, SIGNAL(sig2) );
  ```

Sample (1)

```
class Foo : public QObject {
Q_OBJECT
public:
   Foo();
   int value() const { return val; }
// declare a possible connection from outside
public slots:
   void setValue( int );
// emitted signal
signals:
   void valueChanged( int );
// private field
private:
   int val;
};
```

Sample(2)

The signal informs the outside that the state of an object has changed

```
void Foo::setValue( int v ) {
       if ( v != val ) {
              val = v;
              emit valueChanged(v);
       }
}
```

Sample(3)

```
Foo a, b;
connect(&a, SIGNAL(valueChanged(int)),
        &b, SLOT(setValue(int)));
b.setValue( 11 ); // a == unknown b == 11
a.setValue( 79 ); // a == 79 b == 79
b.value();        // returns 79
```

The call to `a.setValue(79)` emits a signal `valueChanged()`
which is received by the object `b` ,
which emits a signal ,too (ignored because not connected)

Event Management

Every graphical components (derived from QWidget) handles the events:

- which relate to the window manager
  `closeEvent(), focusInEvent(), enterEvent(), paintEvent()`
- which relate to keyboard
  `keyPressEvent(), keyReleaseEvent()`
- which relate to the mouse
  `mouseMoveEvent(), mousePressEvent()`
- which relate to a clock: `timerEvent()`
- which relate to user-defined events : `event()`

Sample

```
bool MyClass::keyPressEvent( QKeyEvent *e) {
      if (e->key() == Key_F1) {
            ....
            e->accept();
      }
      if (e->key() == Key_Escape) {
            e->ignore(); // send the event to the ancestor
            }
}
```

Meta-compiler compilation

Signal/Slot communication provided by Qt are handled by a meta-compiler: `moc` (Meta Object Compiler)

- `moc` takes as input C++ files that contains the declaration : `Q_OBJECT`
- `moc` generates a C++ file that implements mechanisms for :
  - communication on object during the executing of the soft ;
  - dynamic libraries management;
- the resulting file is then parsed by a traditional compiler.

Compilation through the command make

(c.f
http://doc.qt.digia.com/qt/moc.html#command-line-options)

```
moc_%.cpp: %.h
        moc $(DEFINES) $(INCPATH) $< -o $@
```

or by invidual rules:

```
moc_foo.cpp: foo.h
        moc $(DEFINES) $(INCPATH) $< -o $@
```

Compilation through the command make (2)

- You must also remember to add
  - `moc_foo.cpp` to your `SOURCES` (substitute your favorite name) variable
  - and `moc_foo.o` or `moc_foo.obj` to your `OBJECTS` variable.
- Both examples assume that
  - `$(DEFINES)` and `$(INCPATH)` expand to the define and include path options that are passed to the C++ compiler.
  - These are required by `moc` to preprocess the source files.

Compilation through the command `qmake`

(`http://doc.qt.digia.com/qt/qmake-manual.html#qmake`)

Qt provides us with the tool `qmake` to create `Makefile` which are platform specific (files with `.pro` extensions)

```
SOURCES = hello.cpp main.cpp
FORMS = hello.ui
HEADERS = hello.h
CONFIG += qt
```

a qmake command looks like:

```
qmake -o Makefile hello.pro
```

and then simply

```
make
```

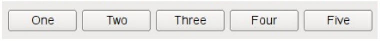
The Layout

The *Layout* mechanism allows :

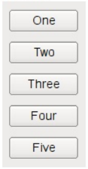
- the location of the attached components;
- the default size of components;
- the minimum size of components;
- the management of resizing ;
- the management of updates (contents of components, adding, suppress...).

Some Layouters

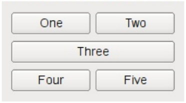
- A QHBoxLayout lays out widgets in a horizontal row, from left to right (or right to left for right-to-left languages).



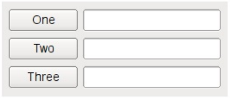
- A QVBoxLayout lays out widgets in a vertical column, from top to bottom.



- A QGridLayout lays out widgets in a two-dimensional grid. Widgets can occupy multiple cells.



- A QFormLayout lays out widgets in a 2-column descriptive label- field style.



Some Layouters

- `QHBoxLayout` presents elements on horizontal basis;
- `QVBoxLayout` presents elements on vertical basis;
- `QFormLayout` presents elements on a two-colums grid ;
- `QGridLayout` presents elements on complex grid (elements spreading has to be specified)

Placement Policy

1. every components possess a spreading space according to their properties;
2. the `stretchfactor` is taken into account if it is greater than 1;
3. if the `stretchfactor` is equal to 0, the remaining space is allowed;
4. a component is never smaller than its default minimum size;
5. a component is never greater than its default maximum size;

Sample : QHBoxLayout

from http://doc.qt.digia.com/qt/layout.html

```
QWidget *window = new QWidget;
     QPushButton *button1 = new QPushButton("One");
     QPushButton *button2 = new QPushButton("Two");
     QPushButton *button3 = new QPushButton("Three");
     QPushButton *button4 = new QPushButton("Four");
     QPushButton *button5 = new QPushButton("Five");

     QHBoxLayout *layout = new QHBoxLayout;
     layout->addWidget(button1);
     layout->addWidget(button2);
     layout->addWidget(button3);
     layout->addWidget(button4);
     layout->addWidget(button5);

     window->setLayout(layout);
     window->show();
```

Sample : Grid Layout

from `http://doc.qt.digia.com/qt/layout.html`

```
QWidget *window = new QWidget;
QPushButton *button1 = new QPushButton("One");
QPushButton *button2 = new QPushButton("Two");
QPushButton *button3 = new QPushButton("Three");
QPushButton *button4 = new QPushButton("Four");
QPushButton *button5 = new QPushButton("Five");

QGridLayout *layout = new QGridLayout;
layout->addWidget(button1, 0, 0);
layout->addWidget(button2, 0, 1);
layout->addWidget(button3, 1, 0, 1, 2);
layout->addWidget(button4, 2, 0);
layout->addWidget(button5, 2, 1);

window->setLayout(layout);
window->show();
```

The third QPushButton spans 2 columns. This is possible by specifying 2 as the fifth argument to QGridLayout::addWidget().

Sample : QFormLayout

from `http://doc.qt.digia.com/qt/layout.html`

```
QWidget *window = new QWidget;
QPushButton *button1 = new QPushButton("One");
QLineEdit *lineEdit1 = new QLineEdit();
QPushButton *button2 = new QPushButton("Two");
QLineEdit *lineEdit2 = new QLineEdit();
QPushButton *button3 = new QPushButton("Three");
QLineEdit *lineEdit3 = new QLineEdit();

QFormLayout *layout = new QFormLayout;
layout->addRow(button1, lineEdit1);
layout->addRow(button2, lineEdit2);
layout->addRow(button3, lineEdit3);

window->setLayout(layout);
window->show();
```

Personal Layouts

It is possible to create your own *layouts*, you just have to :

- derive from the class `QLayout`
- overload the method`resizeEvent(QEvent *e)`
- re-calculate the size of every objects using `setGeometry()`

Menus (Qt 4.7)

`http://doc.qt.digia.com/4.7/qmenu.html`

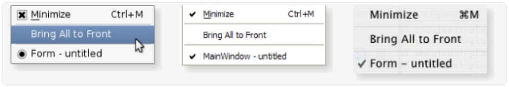


Fig. A menu shown in Plastique widget style, Windows XP widget style, and Macintosh widget style.

- A menu consists of a list of action items.
- added with the addAction(), addActions() and insertAction() functions,
- actions can have :
  - a text label,
  - an optional icon drawn on the very left side,
  - and shortcut key sequence such as "Ctrl+X".
- When inserting action items you usually specify a receiver and a slot.
  - The receiver will be notifed whenever the item is triggered().
  - In addition, QMenu provides two signals, activated() and highlighted(), which signal the QAction that was triggered from the menu.

Menu Sample (Qt 4.7)

```
#include "menus.h"
#include <QAction>
#include <QAxFactory>
#include <QMenuBar>
#include <QMessageBox>
#include <QTextEdit>
#include <QPixmap>
#include "fileopen.xpm"
#include "filesave.xpm"
```

Menu Sample(Qt 4.7)

```
QMenus::QMenus(QWidget *parent) : QMainWindow(parent, 0)
    // QMainWindow's default flag is WType_TopLevel
{
    QAction *action;
    QMenu *file = new QMenu(this);

    action = new QAction(QPixmap((const char**)fileopen),
                         "\&Open", this);
    action->setShortcut(tr("CTRL+O"));


    connect(action, SIGNAL(triggered()),
            this,  SLOT(fileOpen()));
    file->addAction(action);

    action = new QAction(QPixmap((const char**)filesave),
                         "\&Save", this);
    action->setShortcut(tr("CTRL+S"));
    connect(action, SIGNAL(triggered()),
            this, SLOT(fileSave()));
    file->addAction(action);
```

Menu Sample(Qt 4.7)

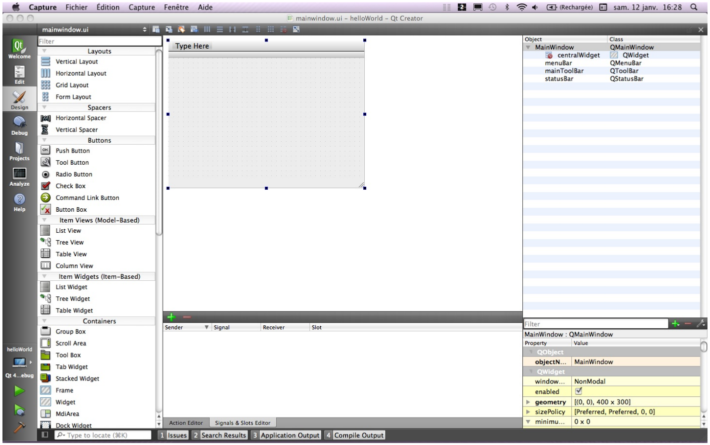
```
    if (!QAxFactory::isServer())
        menuBar()->addMenu(file)->setText("\&File");
    editor = new QTextEdit(this);
    setCentralWidget(editor);

   statusBar();
}
 void QMenus::fileOpen()
{editor->append("File Open selected.");
}

void QMenus::fileSave()
{ editor->append("File Save selected.");
}
```

QtDesigner

UNIVERSITÉ DE NANTES



Object	Class
MainWindow	QMainWindow
centralWidget	QWidget
menuBar	QMenuBar
mainToolBar	QToolBar
statusBar	QStatusBar

MainWindow : QMainWindow

Property	Value
QObject	
objectN...	MainWindow
QWidget	
window...	NonModal
enabled	✓
geometry	[(0, 0), 400 x 300]
sizePolicy	[Preferred, Preferred, 0, 0]
minimu...	0 x 0

Using *QtDesigner*

QtDesigner is a full and easy to use tool that permits the design of graphical apps.

- QtDesigner generates files with `.ui` extensions that describe the interface while respecting the XML format;
- a source genrator `uic` create the associated Qt source
- the user derives the created class to write the application.

Internationalization

Internationalization is easily handled by Qt :

- using 16 bits characters, class `QString`
- providing a support for translation :
  - `tr` is a static method which handle the translation `saveButton->setText(tr ("Enregistrer"));`
  - during the execution, the translation will be performed according to environment variables;
  - translations are handled by the application `QtLinguist`.

A few limitations

- Multiple Inheritance Requires QObject to Be First
- Function Pointers Cannot Be Signal or Slot Parameters
- Enums and Typedefs Must Be Fully Qualified for Signal and Slot Parameters
- Type Macros Cannot Be Used for Signal and Slot Parameters
- Nested Classes Cannot Have Signals or Slots
- Signal/Slot return types cannot be references
- Only Signals and Slots May Appear in the signals and slots Sections of a Class

Custom Widgets in Layouts

(c.f `http://doc.qt.digia.com/qt/layout.html`)

- if you your own widget class, you must communicate its layout properties.
- If the widget has a one of Qt's layouts, this is already taken care of
- If the widget does not have any child widgets, or uses manual layout, you can change the behavior of the widget using any or all of the following mechanisms:
  - Reimplement QWidget::sizeHint() to return the preferred size of the widget.
  - Reimplement QWidget::minimumSizeHint() to return the smallest size the widget can have.
  - Call QWidget::setSizePolicy() to specify the space requirements of the widget.
  - Call QWidget::updateGeometry() whenever the size hint, minimum size hint or size policy changes. This will cause a layout recalculation.