Qt – Localization & testing

Timo Strömmer, May 28, 2010

Contents

Internationalization

i18n & l10n

<symbio>

- Internationalization
  - Process of making generic program, which is suitable for all languages and cultures
- Localization
  - Adapting the program for a specific language and culture
    - Not just language, but also for example number, date and time formats

Getting started

- Qt provides two pre-processor definitions, which help locate parts of program that need to be internationalized
  - Prevent automatic conversion from *const char* * to *QString* and vice versa

```
DEFINES += QT_NO_CAST_TO_ASCII QT_NO_CAST_FROM_ASCII
```

```
38          QMessageBox::information(this, "Message", "Got OK");
39      } else {
40          QMessageBox::information(this, "Message", "Got cancel");
41      }
42  }
```

Build Issues

'QString::QString(const char*)' is private
within this context
/home/tilli/qtprojects/localedialogs/mainwindow.cpp
'QString::QString(const char*)' is private
within this context

Preparing strings

<symbio>

- A literal string can be internationalized with help of *tr* function
  - Available to all *QObject* subclasses
  - Non-QObject functions can use *QCoreApplication::translate*
  - Global data can use QT_TRANSLATE_NOOP

```
QMessageBox::information(this, tr("Message"), tr("Got OK"));

const QString globalString = QT_TRANSLATE_NOOP("Globals", "Some global text...");

void globalHello()
{
    QMessageBox::information(0, QCoreApplication::translate("Globals", "Global"),
                             QCoreApplication::translate("Globals", "Global hello"));
}
```

Translation properties

- String that's being translated can have following properties
  - *Context name*, which in case of *tr* is the class name of the QObject subclass
    - In other cases needs to be provided manually
  - The string to be translated
    - If translation is not found, this is shown in UI

Translation properties

<symbio>

- More translation properties
  - *Disambiguation*, which is used if same *context* has similar strings to be translated
    - Possibility to translate them in different ways

```
QString msg = QInputDialog::getText(this, tr("Query", "messagequery"), tr("Enter message"));
int value = QInputDialog::getInt(this, tr("Query", "valuequery"), tr("Enter value"), 0, -100, 100);
```

  - *Plurality* identifier

```
QMessageBox::information(this, "Message", "Do it " + QString::number(value) + " time(s)");
```

Bad

```
QMessageBox::information(this, tr("Message"), tr("Do it %n time(s)", "", value));
```

Good

Strings from parts

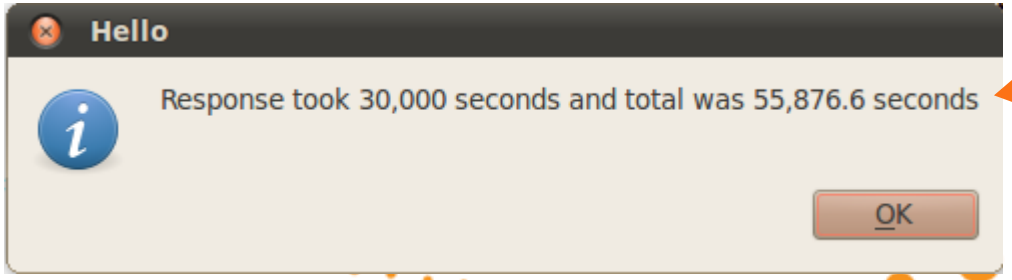
- Don’t use ’+’ to concatenate strings
  - What if the numbers need to be displayed in opposite order?

```
QString msg = tr("Response took ")
            + QString::number(resp)
            + tr(" seconds and total was ")
            + QString::number(total)
            + tr(" seconds");
```

- Use *arg()* instead

```
QString msg = tr("Response took %L1 seconds and total was %L2 seconds").arg(resp).arg(total);
```

  - Now the order of *%1* and *%2* can be changed
    - Note the ’L’ prefix, which localizes integers



That’s actually 30000 and 55876,6…

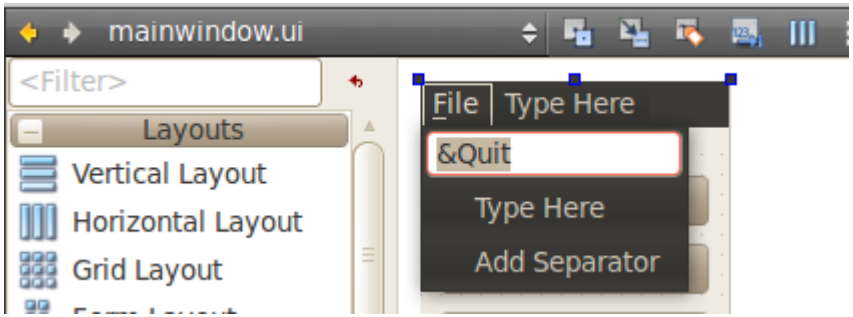
QLocale class

<symbio>

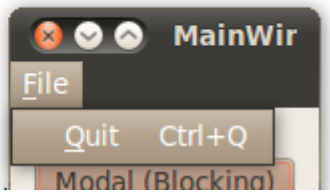
- *QLocale* contains various functions for number, date, time etc. conversions
  - QString functions usually *do not* use locale-specific formatting
    - toInt, toDouble are exeptions. Also arg with %L
    - Use QLocale::toString for numbers
- Creating a *QLocale* object without arguments initializes it to *system locale*

Key acceleratos

- Accelerators and keyboard shortcuts also need to be internationalized
  - &-character goes into *tr()*
  - Prefer *QKeySequence* for keyboard shortcuts
    - Also helps when porting to different platforms

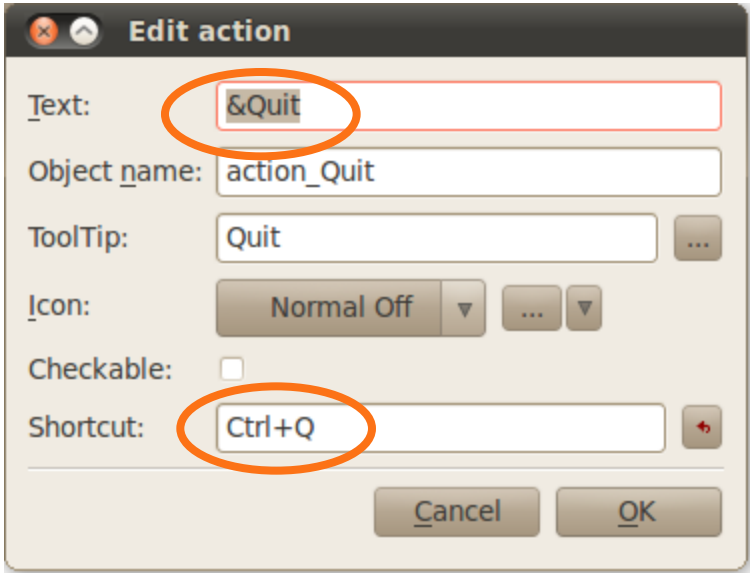



```
ui->action_Quit->setShortcut(QKeySequence::Quit);

nWindow::~MainWindow()
 delete ui;
```



Key accelerators

- The Qt Designer action editor can also be used to specify key accelerators
  - Will be translatable



Source text
&Quit
Finnish translation
&Lopeta

Source text
Ctrl+Q
Finnish translation
Ctrl+L

Point to note

<symbio>

- GUI widgets might need to be resized according to different text length in different languages
  - Use layouts so there’s a possibility to adjust

Translation comments

<symbio>

- Comments are a good idea, otherwise there's going to be misunderstandings between translator and programmer
- Special comment syntax

```
//: The first parameter is an integer
//: Second parameter is a decimal number
QString msg = tr("Response took %L1 seconds and total was %L2 seconds").arg(resp).arg(total);
```

Project files

- All supported languages are added to project *.pro* file

```
TRANSLATIONS = localedialogs_fi.ts
```

- Language is identified by two-character code, *fi* in above example
  - <project-name>_<code>.ts

Translation process

Translation process

<symbio>

- After project has been internationalized, it can be localized
- Open a terminal, go to project directory and run *lupdate <projectname>.pro*
  - No integration with QtCreator
  - Creates a *.ts* file or updates the existing one

```
Updating 'localedialogs_fi.ts'...
    Found 21 source text(s) (21 new and 0 already existing)
```

Translation file

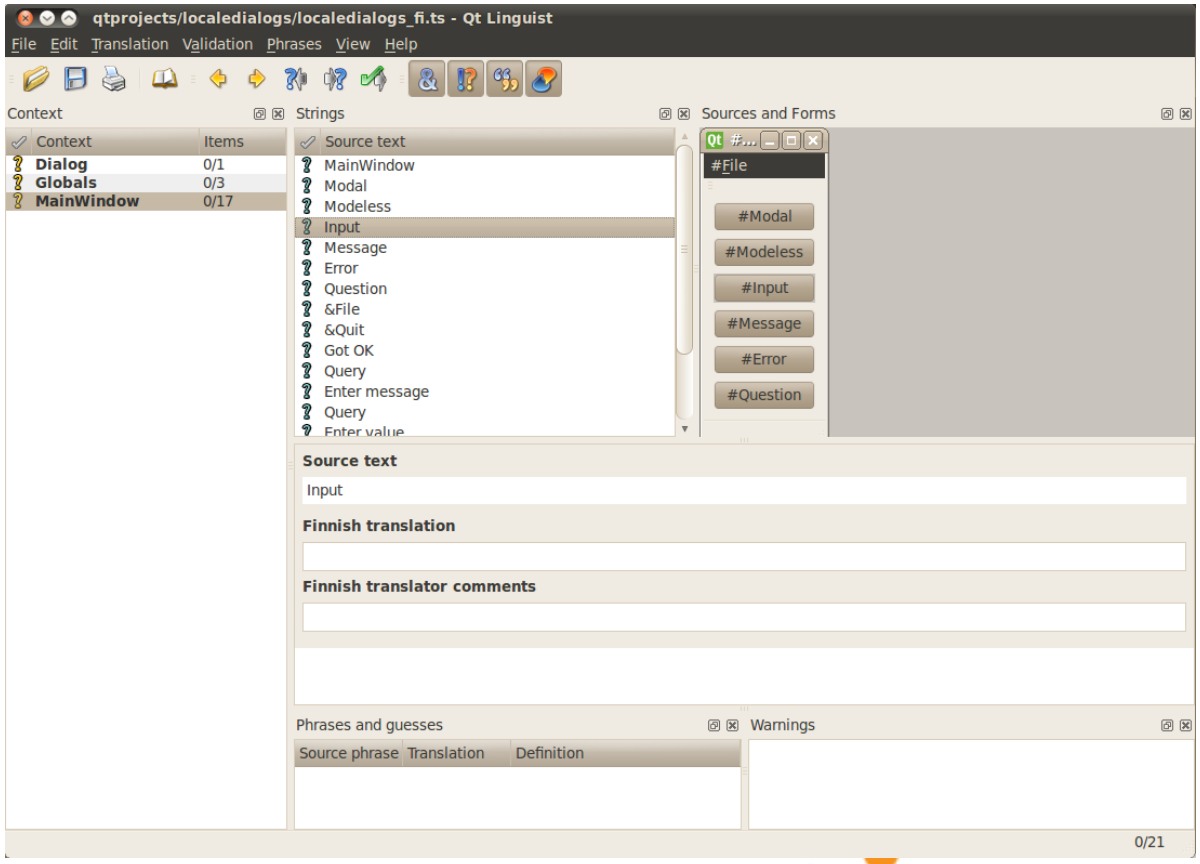
<symbio>

- The *.ts* file is a XML representation of the strings to be translated

```
<context>
    <name>MainWindow</name>
    <message>
        <location filename="mainwindow.ui" line="14"/>
        <source>MainWindow</source>
        <translation type="unfinished"></translation>
    </message>
    <message>
        <location filename="mainwindow.ui" line="21"/>
        <source>Modal</source>
        <translation>Blokkaa</translation>
    </message>
    <message>
        <location filename="mainwindow.ui" line="28"/>
        <source>Modeless</source>
        <translation>Ei blokkaa</translation>
```

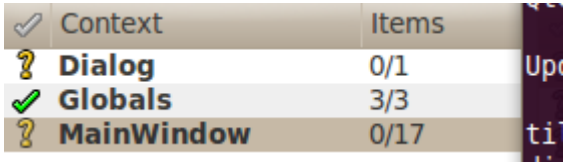
Translation process

- Use *Qt Linguist* tool and open *.ts* files



Translation process

- After translation, open a terminal again
  - Run *lrelease <project-name>.pro* in the project directory to create a binary file from the translations
    - A *.qm* file is created




```
Updating '/home/tilli/qtprojects/localedialogs/localedialogs_fi.qm'...
    Generated 3 translation(s) (3 finished and 0 unfinished)
    Ignored 18 untranslated source text(s)
```

Using translations

<symbio>

- The translations *(.qm)* need to be shipped with the program and loaded when run
  - Use *QTranslator* object
  - Preferably the application would have some kind of preferences dialog or use system locale

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

    QTranslator translator;
    translator.load("localedialogs_fi");
    a.installTranslator(&translator);

    MainWindow w;
    w.show();
    return a.exec();
}
```

```
QTranslator translator;
translator.load("localedialogs_" + QLocale::system().name());
a.installTranslator(&translator);
```

Using translations

- In addition to application-specific translations, Qt provides its own set of translations
  - For example the shortcuts from *QKeySequence*

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

    QTranslator qtTranslator;
    qtTranslator.load("qt_" + QLocale::system().name(),
            QLibraryInfo::location(QLibraryInfo::TranslationsPath));
    a.installTranslator(&qtTranslator);

    QTranslator appTranslator;
    appTranslator.load("localedialogs_" + QLocale::system().name());
    a.installTranslator(&appTranslator);

    MainWindow w;
    w.show();
    return a.exec();
}
```

Qt Linquist

Qt Linquist

<symbio>

- A tool, which is used to write translations for the strings found from the sources
  - No knowledge about programming needed
  - Can work with one or two languages simultaneously
    - In case the translator doesn't know English
- Has certain data validation rules to help avoid problems

Qt Linquist validations

<symbio>

- Accelerator validation
  - Ampersand (&) is there if the original has one
- Punctuation validation
  - If original string ends for example with ’?’, the translated string probably also should

Qt Linquist validations

<symbio>

- Phrase validation
  - Translated strings can be collected into a *phrase book*
  - If phrase book already contains a translation for a phrase, but new translation differs, a warning is issued
- Place marker validation
  - Arguments (%1, %2) should match

Qt Linquist validations

- Can be enabled / disabled from toolbar

Qt phrase book

- A phrase book is a collection of translations from one language to another
  - Can be distributed between projects
  - Use *Ctrl+T* to add a translation to phrase book

Translating a program

<symbio>

- This is interactive part
  - Create a GUI program with menu, widgets, dialogs etc.
  - Add *tr* statements
    - Try QT_NO_CAST_FROM_ASCII
  - Run *lupdate <project>.pro*
  - Translate with Qt Linquist (next slide)
  - Run *lrelease <project>.pro*

Qt Linquist

<symbio>

- This is interactive part
  - Contexts
  - Translations
  - Phrase book usage
  - Validation rules

Testing

QTestLib introduction

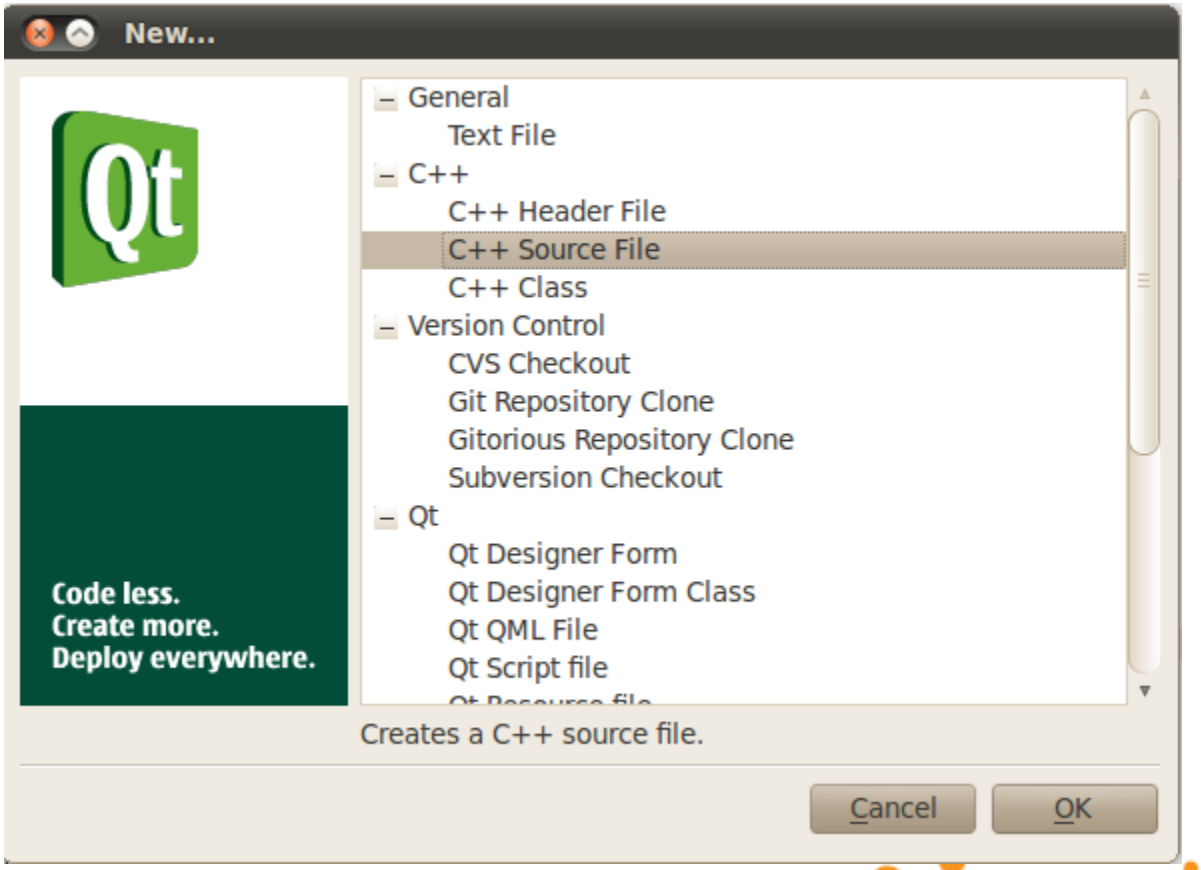
QTestLib

<symbio>

- QTestLib is a library for creating test cases
  - Can be used outside of a library or integrated into a library or GUI program
  - Provides functionality to simulate events (mouse, keyboard) when testing a GUI
  - Easy to read output that can also be interpreted by tools

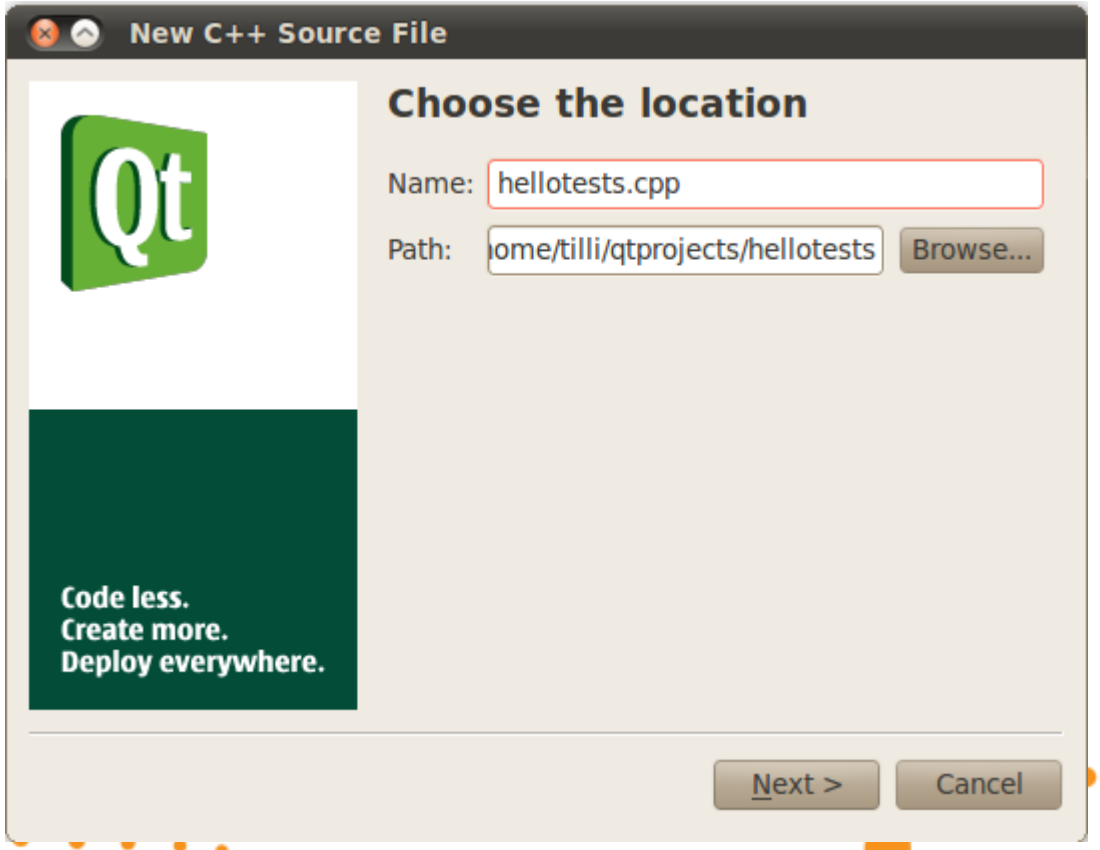
Creating a test program

- Create a new C++ source file



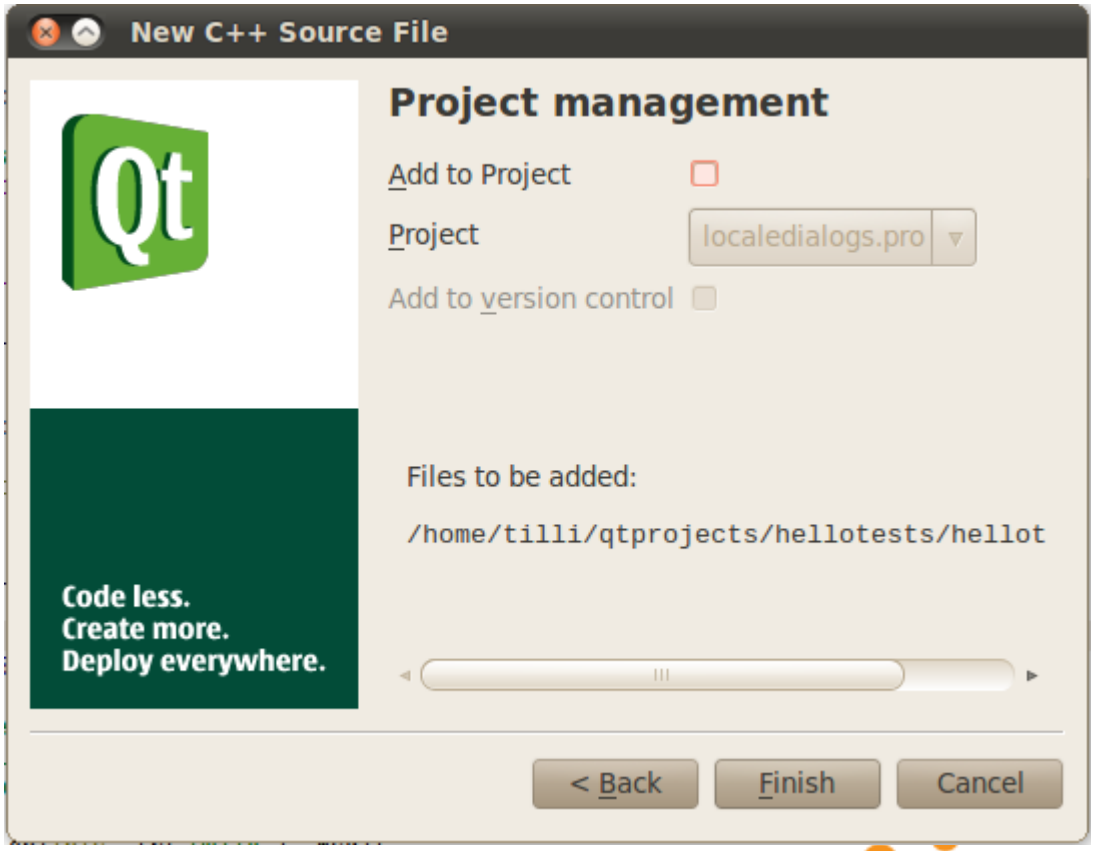
Creating a test program

- You could just as well use file browser or *touch* etc. This just creates an empty file



Creating a test program

- Do NOT add it to a project



Add test object

- Slightly different way to use QObject
  - The test object is added to the *source file* that was just created
  - Each test function is implemented as a private slot
  - QTEST_MAIN macro is used to declare a main function
  - Source file includes a *.moc* file, which is generated by meta-object compiler

```
#include <QtTest/QtTest>

class HelloTest : public QObject
{
    Q_OBJECT

private slots:
    void testHello()
    {
    }
};

QTEST_MAIN(HelloTest)
#include "hellotests.moc"
```

Create a project

- In this case the project is created *after* the first source file
- Open terminal, go to directory of source file and run following to create the project

```
qmake -project "CONFIG += qtestlib"
```

- Open the *.pro* file to QtCreator, build and run

```
********* Start testing of HelloTest *********
Config: Using QTest library 4.6.2, Qt 4.6.2
PASS   : HelloTest::initTestCase()
PASS   : HelloTest::testHello()
PASS   : HelloTest::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped
********* Finished testing of HelloTest *********
```

Test functions

- A test function is a *private slot*
  - Each test function within the object is run once after the program is started
- Special test functions
  - *initTestCase, cleanupTestCase* – Run once at beginning and end
  - *init, cleanup* – Run before and after each test function

Test functions


```
private slots:

    void initTestCase() { qDebug("Run first, can be used to allocate data"); }

    void cleanupTestCase() { qDebug("Run last, cleanup what was allocated in initTestCase"); }

    void init() { qDebug("Run before each test function"); }

    void cleanup() { qDebug("Run after each test function"); }

    void testHello()
    {
        qDebug("Hello");
    }

    void testHello2()
    {
        qDebug("Hello 2");
    }
```

```
QDEBUG : HelloTest::initTestCase() Run first, can be used to allocate data
PASS   : HelloTest::initTestCase()
QDEBUG : HelloTest::testHello() Run before each test function
QDEBUG : HelloTest::testHello() Hello
QDEBUG : HelloTest::testHello() Run after each test function
PASS   : HelloTest::testHello()
QDEBUG : HelloTest::testHello2() Run before each test function
QDEBUG : HelloTest::testHello2() Hello 2
QDEBUG : HelloTest::testHello2() Run after each test function
PASS   : HelloTest::testHello2()
QDEBUG : HelloTest::cleanupTestCase() Run last, cleanup what was allocated in initTestCase
```

Doing something useful

<symbio>

- Verifying conditions

```
void testHello()
{
    QVERIFY( 5 + 5 != 10 );
}

void testHello2()
{
    QVERIFY( QString("a") < QString("b") );
}
```

```
PASS   : HelloTest::initTestCase()
FAIL!  : HelloTest::testHello() '5 + 5 != 10' returned FALSE. ()
   Loc: [hellotests.cpp(16)]
PASS   : HelloTest::testHello2()
PASS   : HelloTest::cleanupTestCase()
Totals: 3 passed, 1 failed, 0 skipped
```

Doing something useful

- Comparing data

```
void testHello()
{
    QCOMPARE( 5 + 5, 9 );
}

void testHello2()
{
    QCOMPARE( QString("a") + "B", QString("aB") );
}
```

```
PASS   : HelloTest::initTestCase()
FAIL!  : HelloTest::testHello() Compared values are not the same
   Actual (5 + 5): 10
   Expected (9): 9
   Loc: [hellotests.cpp(16)]
PASS   : HelloTest::testHello2()
PASS   : HelloTest::cleanupTestCase()
Totals: 3 passed, 1 failed, 0 skipped
```

Integrating with GUI

- Create GUI project with a form
  - Press ”Up” and ”UP” is shown on text field
  - Press ”Down” and ”DOWN” is shown
- Add the SOURCES (except main.cpp) and HEADERS of the GUI project into the test library project
  - Basically you’re just building the same program with a slightly modified main.cpp
  - *.pri* files are your friends

```
INCLUDEPATH += . ../simplegui

# Input
SOURCES += hellotests.cpp \
    ../simplegui/mainwindow.cpp

HEADERS += ../simplegui/mainwindow.h
```

Generate some events

- *QTest* API provides functions to simulate GUI events
  - Use *QObject::findChild* to search for target

```
    void initTestCase() { win = new MainWindow(); }
    void cleanupTestCase() { delete win; }

    void testUp()
    {
        QTest::mouseClick(win->findChild<QWidget *>("upButton"), Qt::LeftButton);
        QCOMPARE(win->findChild<QLineEdit *>("lineEdit")->text(), QString("UP"));
    }

    void testDown()
    {
        QTest::mouseClick(win->findChild<QWidget *>("downButton"), Qt::LeftButton);
        QCOMPARE(win->findChild<QLineEdit *>("lineEdit")->text(), QString("UP"));
    }

private:
    MainWindow *win;
```

Test results

- In previous example *downButton* was pressed, but ”UP” was expected from text field

```
PASS   : HelloTest::initTestCase()
PASS   : HelloTest::testUp()
FAIL!  : HelloTest::testDown() Compared values are not the same
   Actual (win->findChild<QLineEdit *>("lineEdit")->text()): DOWN
   Expected (QString("UP")): UP
   Loc: [hellotests.cpp(23)]
PASS   : HelloTest::cleanupTestCase()
Totals: 3 passed, 1 failed, 0 skipped
```

  - Note that there’s no need to show the window to modify widgets
    - But can be done

```
void initTestCase()
{
    win = new MainWindow();
    win->show();
    QTest::qWaitForWindowShown(win);
}
```

Pseudolocalization

- Pseudolocalization is a way of testing possible localization issues without having real translations
  - Could for example use machine-translated strings for particular language
    - Or random characters
  - Obviously not replacement for real localiztion testing

Pseudolocalization example

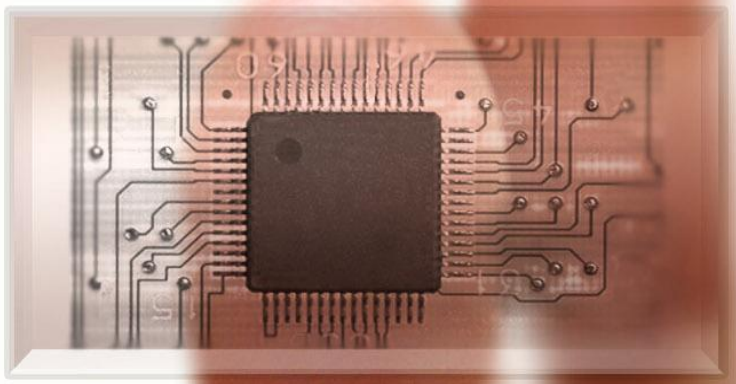

```
    void testLoc()
    {
        // Test some strings
        recursiveSetText(win);
        QTest::qWait(1000); // Non-blocking wait for window to resize
        QMessageBox::StandardButton btn = QMessageBox::question(
                0, "Test", "Did it work?", QMessageBox::Yes | QMessageBox::No);

        // Reset back before verifying conditions
        recursiveResetText(win);

        // If tester says no, ask for a reason
        if (btn != QMessageBox::Yes) {
            QString input = QInputDialog::getText(0, "Test", "Why?");
            QFAIL(qPrintable("Failed by tester, reason: " + input));
        }
    }

private:

    void recursiveSetText(QObject *parent)
    {
        foreach (QObject *child, parent->children()) {
            QString original = child->property("text").toString();
            if (!original.isEmpty()) {
                child->setProperty("text", QString("gjtdtkj ouiyfrf"));
                child->setProperty("originalText", original);
            }
            recursiveSetText(child);
        }
    }
```

```
FAIL!  : HelloTest::testLoc() Failed by tester, reason: Just looks bad
   Loc: [hellotests.cpp(78)]
PASS   : HelloTest::cleanupTestCase()
Totals: 3 passed, 2 failed, 0 skipped
```

Almost finished

Programming exercise

- No special tasks for today, start preparing for your next week assignment



SERIOUS ABOUT SOFTWARE