Lab 1 – The Basics of Qt

Aim: This lab will take you through all the steps required to build a fully fledged Qt application. The focus is to understand how a Qt application is structured and to learn to find your way round the Qt documentation.

Duration: 3 h

Goals and Background

The goal of this lab is to apply the knowledge from the first lectures and put it in a context. The task at hand is to develop a fully fledged desktop application – a text editor with support for multiple documents, saving and loading, undoing, the clipboard, etc.
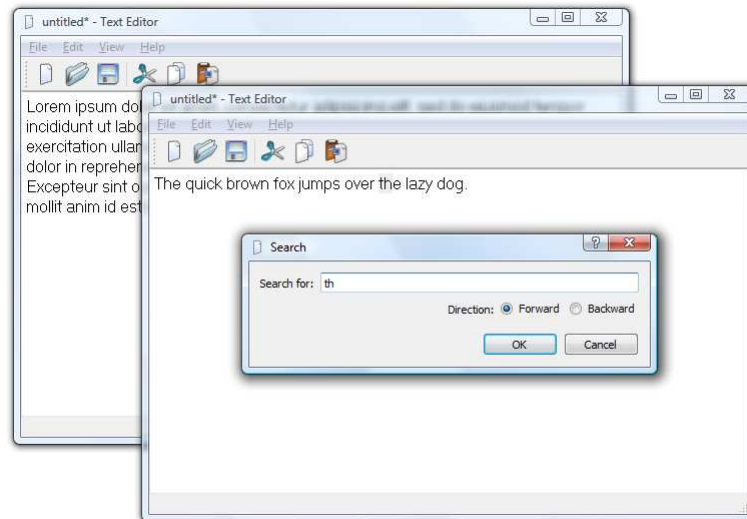

Throughout this lab, new features will be added to the application gradually. The purpose of this is twofold. First, this is how real world software is developed. The feature set grows over time. Secondly, you will have a working application from the very first step so that you can test, verify and debug the functionality continuously. Do not keep a known bug to a later step, make sure that the application works at all times. It will only be harder to properly troubleshoot in a more complex application.

In the lab, the instructions will be less detailed than in the exercises. This means that you are expected to look up classes in the Qt reference documentation, and that adding header files and other trivial code is up to you.

The Basic Application

The basic application is built from the *Qt4 Gui Application* template. Create such a project and include the *QtCore* and *QtGui* modules. Base the skeleton on the `QMainWindow` class. In this lab, the project is called *TextEditor*.
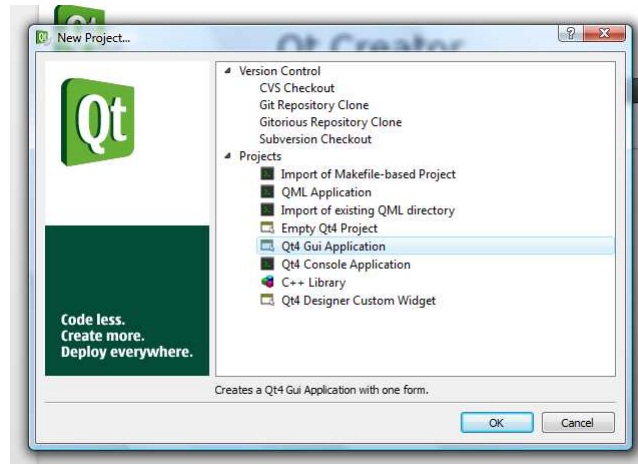
The resulting project consists of the following files:

- ⚔ `TextEditor.pro` – the project definition file.
- ⚔ `mainwindow.ui` – the user interface for the `MainWindow` class.
- ⚔ `mainwindow.cpp/h` – the implementation and declaration of the `MainWindow` class.
- ⚔ `main.cpp` – the `main` function. Gets everything initialized and started.

Review these files and verify that the project builds and runs.

Adding User Interface Elements

The next step is to add a `QTextEdit` widget and the basic user operations *New*, *Close* and *Exit* to the main window.

First of all, we need to add an icon resource to the project in question. This is because the icons will be used in the user interface which you are about to design.

Add a new Qt Resource file to the project. Name it *icons.qrc*. Open the resulting file, add the prefix `icons` and add the `*.png` files for this lab.

Now you are ready to start working on the user interface. Start by opening the `mainwindow.ui` file in Designer mode.

Add a `QTextEdit` widget to the main window and lay it out. Select the window itself to alter the margins of the layout to zero to get the new widget to fill the central area fully.

When the widget is in place you can try the dialog using *Tools – Form Editor – Preview* (*Ctrl+Alt+R*).

The user operations are represented by `QAction` objects. These are administered through the *Action Editor* shown below.
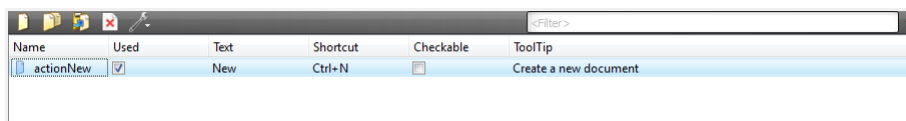
Create the following actions.

Text	Name	Icon	Shortcut	ToolTip
New	`actionNew`		Ctrl+N	Create a new document
Close	`actionClose`		Ctrl+W	Close current window
Exit	`actionExit`			Exit application

Actions are added to the user interface through drag and drop. Drag the `actionNew` action onto the toolbar.

Add the entry *File* to the menu bar. Click on the *type here* text and enter *File*. Now drag and drop the `actionNew`, `actionClose` and `actionExit` to the menu. Add separators between each item by dragging the separator into position (you find it on the menu you are editing).

> If you enter "`&File`" as the title of the menu you get a shortcut, "<u>F</u>ile", automatically. To get an ampersand in a menu text, enter "`&&`".

Implementing Functionality

Each of the actions added need to be implemented. You do this by right clicking on the action in question to bring up its context menu. From the menu, pick the *Go to slot* option and pick the `triggered()` signal from the list that pops up.
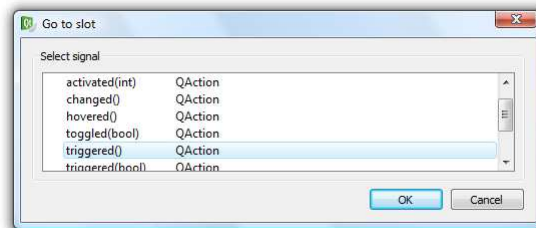
Start by implementing the `actionNew`, this brings you to the following source code.

```
void MainWindow::on_actionNew_triggered()
{

}
```

In this slot, create a new `MainWindow` object on the heap and show it.

The other two actions, `actionClose` and `actionExit`, already have matching slots implemented by Qt. Make the following connections in the `MainWindow` constructor, after the `setupUi` call.

Source	Signal	Destination	Slot
actionClose	triggered()	this (current window)	close()
actionExit	triggered()	qApp (current application)	closeAllWindows()

Also, make sure to call `setAttribute` and set the `Qt::WA_DeleteOnClose` attribute in the `MainWindow` constructor.

Self Check

- ⚔ Ensure that you can open new windows.
- ⚔ Ensure that *File – Close* closes the current window.
- ⚔ Ensure that *File – Exit* closes all windows and thus terminates the application.
- ⚔ Explain why the `Qt::WA_DeleteOnClose` attribute is set. Where does it come into play and why is that important?

Modifying and Closing

Now, the user can close a window with unsaved changes by mistake. That is not the expected behavior. Instead, the user expects to be prompted if risking to lose unsaved work. A two stage solution is needed to address this. First, the modified status of the document needs to be monitored. Second, the user must be prompted when trying to close a window with a modified document.

By monitoring the QTextEdit's textChanged signal document modifications can be tracked. Create a private slot called documentModified in your MainWindow class. In the constructor, connect the editor's signal to your slot. In the slot, set the windowModified property to true.

> When reading the documentation, the windowModified property is not found in the QMainWindow class. Instead, it is defined in the QWidget class which QMainWindow inherits. You spot this by reading the properties section of QMainWindow, where "...*properties inherited from QWidget*" are mentioned.

The windowModified property interacts with the windowTitle property. For Mac OS X, the modified state is indicated by a dot in the window's red (left-most) button. On most other platforms, an asterisk in the title indicates that the document has been modified. This asterisk can be added to the windowTitle as "[*]". Then Qt will synchronize the windowModified property and the windowTitle automatically.

To use this, set the windowTitle to "TextEditor[*]" in the MainWindow constructor and verify that modifications to the document triggers the document modified indication.

The other half of the solution is to prompt the user when the document is closed. This is done by re-implementing the protected closeEvent method of MainWindow. Start this by adding the function to your class declaration.

```
protected:
    void closeEvent(QCloseEvent *e)
```

Then add an empty function body in your class implementation.

```
void MainWindow::closeEvent(QCloseEvent *e)
{

}
```

The event, e, can be accepted or ignored using e->accept() and e->ignore(). Implement the function so that the event is accepted for all unmodified documents. If the document is modified, use the QMessageBox::warning function to prompt the user and accept or ignore accordingly.

The warning method takes the following arguments (look in the documentation if you need the actual types).

```
QMessageBox::warning( parent widget,
                      dialog title, dialog text,
                      buttons to show, default button )
```

The dialog should show the QMessageBox::Yes and QMessageBox::No buttons, where *No* is default.

The return value from the function is the button clicked by the user – but as the user can close the dialog by closing its windows as well as clicking *No*, test for the *Yes* button.

Having implemented the close method, the *File – Close* and *File – Exit* menu actions should work as expected.

Self Check

- ⚔ Ensure that the document modified indication (asterisk or dot in the red button) is triggered by modifying the document.

- ⚔ Open a document, modify it, attempt to close the window using the following methods: the current window manager (i.e. click the "X" in Windows, etc), *File – Close, File – Exit.* Ensure that you are prompted.

- ⚔ Ensure that when prompted, you can prevent the window from being closed by clicking *No* or rejecting the dialog in any other way (i.e. close it using the current window manager).

- ⚔ Open a document, do not modify it, attempt to close the window using the methods above. Ensure that you are not prompted.

- ⚔ Open several documents, modify some, activate *File – Exit.* Ensure that only modified windows are closed and that the entire closing procedure is canceled if you answer *No* when prompted.

Settings and Customization

Almost all applications available today have some sort of user settings. It can be everything from which measurement system to use, the preferred order of the toolbars, down to tuning very specific details. In this step we will let the user choose the font used for displaying the text document being edited.
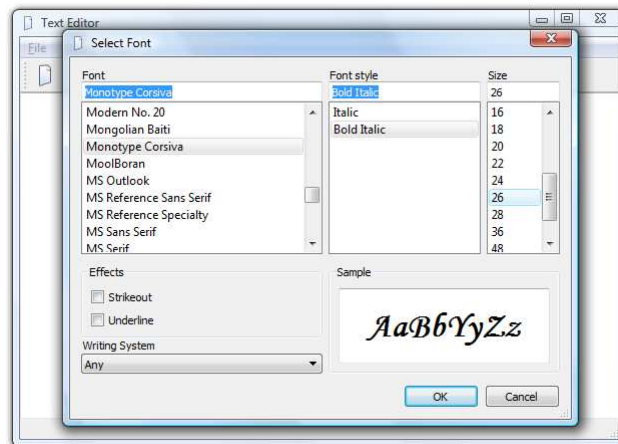
The setting will be stored using the platform's preferred method, i.e. the registry for Windows and hidden files for Linux, etc.

The Ground Work

It is possible to specify where to store settings and in which format in Qt, but a more convenient method is to set some basic properties to the `QApplication` object and then rely on `QSettings`' default behavior.

The required settings concern the application name and version, the (producing) organization's name and domain. Simply add the following code to your main function, where `a` is the `QApplication` instance.

```
a.setApplicationName("Text Editor");
a.setApplicationVersion("0.1");

a.setOrganizationName("ExampleSoft");
a.setOrganizationDomain("example.com");
```

While tweaking the properties of the `QApplication` object, set the `windowIcon` property to use the `":/icons/new.png"` as icon.

Adding Actions

Add the following action to the main window.

Text	Name	Icon	Shortcut	ToolTip
Select Font...	`actionSelectFont`			Select the display font

Add the actions to a menu called *View* (you will have to create the menu in the menu bar).

Right click on the action and go to the slot for the `triggered` signal. The slot body will look like this.

```
void MainWindow::on_actionSelectFont_triggered()
{

}
```

In this slot, use the `QFontDialog::getFont( *ok, initial, parent )` method to acquire a new `QFont`. Use the `ui->textEdit->font()` as the initial font. If the user accepted the dialog, i.e. `ok` is `true`, update the `font` property of `ui->textEdit`. Also, use a `QSettings` object to set the key "viewFont" to hold the current font as its value.

> You can allocate the `QSettings` object on the stack, as the construction and destruction of the object is very fast and all its settings are acquired from the `QApplication` object.

Finally, to retrieve the stored font, use a `QSettings` object to set the font property of the `ui->textEdit` widget to the set value. Do this in the constructor of `MainWindow`, and use `QApplication::font()` as the default value for the setting.

> The value from a `QSettings` object is a `QVariant`. Use the `QVariant::value<T>()` function, with `T=QFont`, to get the `QFont` from the `QVariant`.

Self Check

- ⚔ Ensure that all windows have the new icon as window icon.
- ⚔ Test that you can change the font of a given window.
- ⚔ Test that you can cancel the font dialog.
- ⚔ Change font and open a new window. Ensure that the new window gets the new font by default.
- ⚔ Verify that the font selection is kept if you exit the application and relaunch it.

The Clipboard and Change History

Adding the expected clipboard operations *Cut*, *Copy* and *Paste* along with *Undo* and *Redo* is very simple. It is so simple that we will spend some extra time on adding an *About* dialog and more.

Most Qt widgets are prepared to be used in real world situations and provide interfaces to be easy to use. The `QTextEdit` widget is no exception to this. It provides slots for copy, cut, paste, undo and redo, as well as signals for enabling the different actions. Still, you need to add actions for all these operations and connect them.

Adding Actions

Start by adding the following actions to the main window

Text	Name	Icon	Shortcut	ToolTip
About	`actionAbout`			
About Qt	`actionAboutQt`			
Cut	`actionCut`		Ctrl+X	Cut
Copy	`actionCopy`		Ctrl+C	Copy
Paste	`actionPaste`		Ctrl+V	Paste
Undo	`actionUndo`		Ctrl+Z	Undo the last action
Redo	`actionRedo`		Ctrl+Y	Redo the last action

Place the About actions in a menu called *Help* (you will have to create the menu). Place the other actions in a menu called *Edit* (you will have to create the menu). Rearrange the menus of the menu bar by dragging and dropping so that the order is *File*, *Edit*, *View* and *Help*.

Add the clipboard actions to the toolbar as well. You can right click on the toolbar to add a separator between `actionNew` and the clipboard actions.

Implementing Functionality

All actions except *About* need only to be connected to work. This means that the functionality already is implemented by Qt.

In the constructor of the `MainWindow` class, make the following connections.

Source	Signal	Destination	Slot
`actionAboutQt`	`triggered()`	`qApp`	`aboutQt()`
`actionCut`	`triggered()`	`textEdit`	`cut()`
`actionCopy`	`triggered()`	`textEdit`	`copy()`
`actionPaste`	`triggered()`	`textEdit`	`paste()`
`actionUndo`	`triggered()`	`textEdit`	`undo()`
`actionRedo`	`triggered()`	`textEdit`	`redo()`
`textEdit`	`copyAvailable(bool)`	`actionCopy`	`setEnabled(bool)`

textEdit	copyAvailable(bool)	actionCut	setEnabled(bool)
textEdit	undoAvailable(bool)	actionUndo	setEnabled(bool)
textEdit	redoAvailable(bool)	actionRedo	setEnabled(bool)

The `enabled` properties of `actionCopy`, `actionCut`, `actionUndo` and `actionRedo` are updated by the corresponding `QTextEdit` signal. However, they are not initialized. In the constructor, add code to initialize them all with their `enabled` property set to `false`.

The only action left to react to is `actionAbout`. Right click on the action and go to the slot for the `triggered` signal. The slot body will look like this.

```
void MainWindow::on_actionAbout_triggered()
{

}
```

In the slot, use `QMessageBox::about` to show an about dialog for your application.

Self Check

- ⚔ Check that *Help – About* shows an about dialog for your application.
- ⚔ Check that *Help – About Qt* shows a dialog about the used Qt version.
- ⚔ Check that *Undo* and *Redo* work as expected.
- ⚔ Check that *Undo* and *Redo* are enabled as expected – even in a freshly started application.
- ⚔ Check that *Cut*, *Copy* and *Paste* work as expected.
- ⚔ Check that *Cut* and *Copy* are enabled as expected – even in a freshly started application.

File Operations

The biggest drawback with the text editor application is its lack of file operations. You cannot load documents, nor save them. In this section this will be addressed.

Saving and loading files has impact on other parts of the application as well. Each window must now have a file name, and the `closeEvent` must take the ability to save and close a document into account.

Loading Documents

Start by adding the private `QString m_fileName` to the `MainWindow` class declaration.

Modify the constructor of `MainWindow` to have the following signature.

```
MainWindow(const QString &fileName=QString(), QWidget *parent=0);
```

Finally, at the end of the constructor, add the line below.

```
loadFile(fileName);
```

As you understand, the task at hand is to build the `loadFile` method. Start by adding it as a private method of the `MainWindow` class. The method body should look like this.

```
void MainWindow::loadFile(const QString &fileName)
{

}
```

In this function, do the following:

1. If the `fileName` is empty, call `setFileName(QString())` (`setFileName` has not yet been implemented) and return.

2. Create a `QFile` object for the file `fileName`.

3. Attempt to open file `QFile` object for reading a text file, i.e. with the flags `QIODevice::ReadOnly` and `QIODevice::Text`.

4. If the file cannot be opened, show an error message using `QMessageBox::warning`, call `setFileName(QString())` and return.

5. If the file is opened, create a `QTextStream` object working on the `QFile` object.

6. Set the `text` property of the `textEdit` widget to the result of the `readAll` function of the

```
QTextStream.
```

7. Close the `QFile` object.

8. Call `setFileName(fileName)`.

9. Set the `windowModified` property to false.

Now add the missing `setFileName(const QString &)` method as a private member of `MainWindow`. In it, assign `m_fileName` with the file name given and set the `windowTitle` property to the following string value.

```
QString("%1[*] - %2")
    .arg(m_fileName.isNull()?"untitled":QFileInfo(m_fileName).fileName())
    .arg(QApplication::applicationName())
```

Now, add the following action to the `MainWindow` user interface. Add the action to the toolbar and *File* menu.

Text	Name	Icon	Shortcut	ToolTip
Open...	`actionOpen`		Ctrl+O	Open a document

Go to the automatically generated slot for the triggered signal. The function body is shown below.

```
void MainWindow::on_actionOpen_triggered()
{

}
```

In the function, implement the following.

1. Use the following function call to get a file name to open.

```
QString fileName = QFileDialog::getOpenFileName(this,
    "Open document", QDir::currentPath(), "Text documents (*.txt)");
```

2. If the file name `isNull`, return.

3. If the current window's `m_fileName isNull` and the current document is unmodified, call `loadFile(fileName)`, i.e. load the document into the current window.

4. Otherwise, create a `new QMainWindow(fileName)` and show it, i.e. load the document into a new window.

Now experiment with this functionality. Open documents, create new documents and open from them, etc. Also close documents and ensure that the prompt only appears when applicable.

Saving Documents

Saving documents is slightly more complex than loading them. One half of this is that the user can *Save* or *Save As*, the other half is that it is important that the save succeeds and that failure is reported to the user.

Start the implementation of the save functionality by adding the following private slots to the `MainWindow` class declaration. Also, add some basic function bodies in the `MainWindow` implementation.

```
private slots:
    bool saveFile();
    bool saveFileAs();
```

As you can tell, both functions return a boolean. The idea is that `true` is returned if the file actually was saved and `false` if it failed. This is something that we will use later when re-implementing the handling of the user closing modified document.

The responsibilities of the two slots are that `saveFile` does the actual saving, while `saveFileAs` requests a new file name and then uses `saveFile` to save.

Start by implementing `saveFileAs` according to the flow described below.

1. Use the following line to get a file name from the user.

```
QString fileName = QFileDialog::getSaveFileName(this, "Save document",
    m_fileName.isNull()?QDir::currentPath():m_fileName, "Text documents (*.txt)");
```

2. If the file name acquired `isNull`, return false, the file has not been saved.

3. If the file name is valid, call `setFileName` to set the file name, then call `saveFile`. Return the value returned from `saveFile`.

The next step is to implement the `saveFile` slot. Implement the function as follows.

1. If `m_fileName` `isNull`, call `saveFileAs` and return the value returned from that call.

2. If `m_fileName` not `isNull`, create a `QFile` object for the file name.

3. Attempt to open file `QFile` object for writing text files, i.e. with the flags `QIODevice::WriteOnly` and `QIODevice::Text`.

4. If the `QFile` object could not be opened, use `QMessageBox::warning` to inform the user, call `setFileName(QString())` and return `false`.

5. If the `QFile` object opened, create a `QTextStream` object for the file.

6. Write the `textEdit->toPlainText()` to the `QTextStream`.

7. Close the `QFile`.

8. Set the `windowModified` to `false`.

9. Return `true`.

The functions `saveFile` and `saveFileAs` call each other. Explain how you guarantee that they do not get stuck in an infinite loop of calling each other.

Now, add the following actions to the `MainWindow` user interface. Add both to the *File* menu and `actionSave` to the toolbar.

Text	Name	Icon	Shortcut	ToolTip
Save	`actionSave`		Ctrl+S	Save document
Save As...	`actionSaveAs`		Ctrl+Shift+S	Save document as

In the constructor of `MainWindow`, connect these slots to the `saveFile` and `saveFileAs` slots.

Make sure to build and run the application. Experiment with the new functions. Ensure that documents can be loaded, saved and saved as. Also, ensure that the modified flag is turned on and off as expected and that the prompt on close is shown as expected.

Closing Windows

Right now, the user is asked whether to close or not to close a window with modified contents. The most common alternatives are 'discard changes and close', 'save changes and close' or 'do not close'. In order to achieve this, we must incorporate the file saving functionality in the `closeEvent` method.
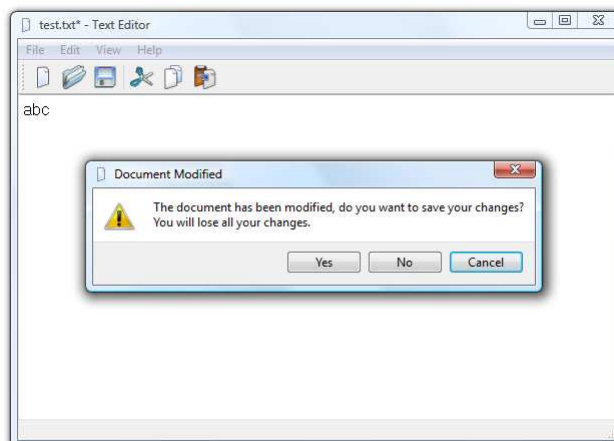
Start with the following `closeEvent` function body.

```
void MainWindow::closeEvent(QCloseEvent *e)
{
    if(m_modified)
    {
        switch(QMessageBox::warning(this, "Document Modified",
            "The document has been modified. "
            "Do you want to save your changes?\n"
            "You will lose and unsaved changes.",
            QMessageBox::Yes | QMessageBox::No | QMessageBox::Cancel,
            QMessageBox::Cancel))
        {
        case QMessageBox::Yes:
            // [1]
            break;
        case QMessageBox::No:
            // [2]
            break;
        case QMessageBox::Cancel:
            // [3]
            break;
        }
    }
    else
    {
        // [4]
    }
}
```

In the locations marked [1], [2], [3] and [4], ignore or accept the close event as appropriate. In one of the cases you need to call `saveFile` and ignore or accept depending on the success of the file saving operation.

Self Check

- ⚞ Verify that you can load documents.

- ⚞ Verify that you can save documents (and properly reload them).

- ⚞ Verify that, when loading, saving and saving as, the window title is updated with the file name as expected.

- ⚞ Verify that, when loading, saving and modifying documents, the `windowModified` state is updated as expected.

- ⚞ Try closing a modified but unnamed document window. Ensure that you get to pick a file name when asking the application to save the changes.

- ⚞ Try closing a modified but unnamed document window. Answer that you want to save your changes but cancel the file name picking dialog. Ensure that the window is not closed.