GUIDED PROJECT

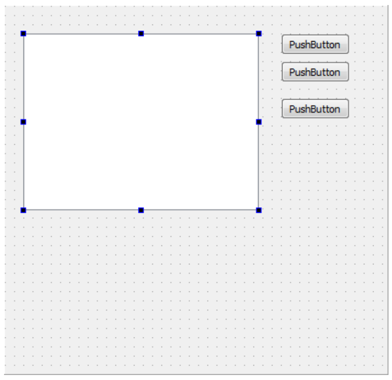
This tutorial will guide through some features of Qt and will specify the steps you need to do for creating a simple GUI application.

NOTE: In writing the program don't forget to add the Qt headers you need to include (e.g. *#include <QInputDialog>*) at the top of the file in which you use it. Remember that the code entered in the header (*.h*) file is prepended to code of the *.cpp* file by the compiler. Therefore, if you make an inclusion in the header file, you don't have to make the same inclusion in the *.cpp*.

Create a new **Qt GUI Project**. Name it ListDemo or similar.

Create a template application based on **QWidget**. Call the class *Widget* for simplicity.

Now add three QPushButtons and a QListWidget as shown below.



Rename and re-title the buttons according to the table below. You can change the name in the property editor. The text can be changed from the property editor or by double clicking on the button and type.

Button	**Property**	**New value**
Top	Name	addButton
Top	Text	Add...
Middle	Name	deleteButton
Middle	Text	Delete
Bottom	Name	clearButton
Bottom	Text	Clear

NOTE: You can see a preview from the menu **Tools – Form Editor**. You can preview it in different styles (Windows, Plastique, etc).

Connect the clear button's *clicked()* signal to the list widget's *clear()* slot.

Right click on the add button and pick *Go To Slot...*

Pick the *clicked()* signal from the dialog that pops up.

Add the following code in the method frame created by the program:

```
void Widget::on_addButton_clicked()
{
  QString newText = QInputDialog::getText(this, "Enter text", "Text:");
      if( !newText.isEmpty() )
                  ui->listWidget->addItem(newText);
}
```

Create an *on_deleteButton_clicked()* slot associated to the delete button. Follow the same procedure as before.

Enter the code below in the slot method. Note the use of *foreach* macro and that deleting an item removes it from the list. The QListWidgetItem tells the parent list (listWidget) about its deletion.

```
void Widget::on_deleteButton_clicked()
{
  foreach (QListWidgetItem *item, ui->listWidget->selectedItems())
      delete item;
}
```

You can notice that the delete button always is enabled, that is not correct. It needs to be enabled or disabled as soon as the selection of the list changes. To implement this functionality do the following:

Create a private slot called *updateDeleteEnabled* in the headers.

Create the method frame in the source and write the following code:

```
void Widget::updateDeleteEnabled()
{
ui->deleteButton->setEnabled(ui->listWidget->selectedItems().count()!=0)
}
```

Now, you need to connect the signal emitted by the listWidget with the slot you have just created.

After the *ui->setupUi* call, enter the following code.

```
connect(ui->listWidget->selectionModel(),
   SIGNAL(selectionChanged(QItemSelection,QItemSelection)), this,
   SLOT(updateDeleteEnabled()));
```

!!! Does the application works as expected? If not, try to work out yourself why it does not work and try to fix it !!!

Cross-platform test:

Copy the relevant file (.pro, .ui, .h, .cpp) on an external support and switch Operating System.

Open the .pro file with the Qt Creator and build and run the application (try to solve any building issue yourself, if you can). Be sure it runs as expected under the other OS.

CALCULATOR PROJECT

TASK 1

In this tutorial you have to create a calculator that allows the user to sum 2 integer numbers.

You are required to design your project by following a Model/View approach. This means that you have to separate the operations made by the calculator on the data (numbers), e.g. sum, and the operation of the user interface that you need in order to enter and display the data.
To do this, you need to have at least two classes on your project, one for the GUI and one for the actual calculation. As an example, you can have:

- the class *CalculatorInterface* that inherits from QWidget. This is the class that holds all the user interaction and displayoperations.
- The class *Calculator* that performs calculations.

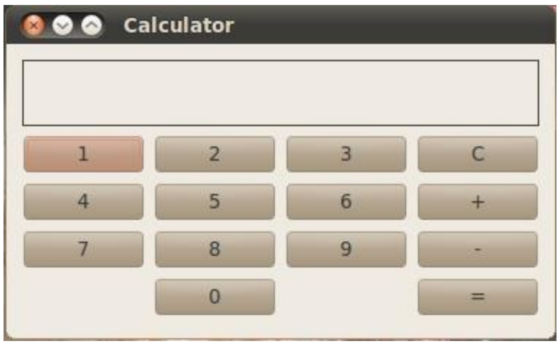
Note that the two classes have to communicate in some way, since you need to pass the numbers from the class *CalculatorInterface* to the class *Calculator* in order to make the calculation. The same form of communication is needed to display the results of the calculation of the screen.

Your calculator should have a
- *Clear button*, to reinitialize the data and give the user the possibility to make a new calculation;
- *Plus button*, for adding to number;
- *Minus button*, for subtracting two numbers;
- *Equal button*, that displays the result.

As a general requirement, you need to add all the buttons for entering numbers.

Design carefully the structure of the project. This will reduce enormously the time spent on writing the code. Try to keep in mind possible extensions and design it accordingly.

The user interface should look like this:



Note that I used a label for displaying numbers, not a lineEdit. In addition, I used a grid layout for organizing the widgets on the form.

TASK 2

Copy the relevant file (.pro, .ui, .h, .cpp) on an external support and switch Operating System.

Open the .pro file with the Qt Creator and build and run the application (try to solve any building issue yourself, if you can). Be sure it runs as expected under the other OS.

After the integer calculator is fully functional, you can improve the current application with the following extensions:

- more operations (e.g. multiplication and division);
- support for entering negative numbers;
- support for floating point numbers.

Depending on the initial design of your application, extending its functionalities can be easy (good modular design) or quite hard, since it might require to rewrite many part of the code.

Cross-platform test:

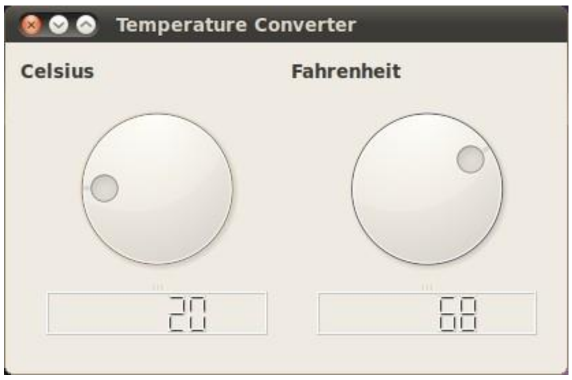
Copy the relevant file (.pro, .ui, .h, .cpp) on an external support and switch Operating System.

Open the .pro file with the Qt Creator and build and run the application (try to solve any building issue yourself, if you can). Be sure it runs as expected under the other OS.

TEMPERATURE CONVERTER PROJECT

This tutorial will guide you through the main steps that you need to do in order to create an application that converts temperatures (Celsius / Fahrenheit). You will be asked to complete some parts of the code alone. Some obvious c++ instructions, such as including headers files, are not specified.

The program you are going to write up implements two *dial* Widgets which allow the user to input the temperature values. Two *LCDNumber* widgets show the values in Celsius and Fahrenheit, respectively. The user interface of the application is the following:



Doing this tutorial will allowyou:

- to practice with input and displaywidgets.
- to use *signal* and *slot* feature provided by Qt, since you will need to connect behaviours of widgets and datamanipulators.
- To practice with a simple model/viewdesign.

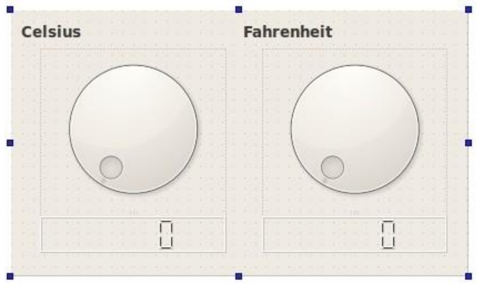
Following the requirement of previous tutorial, this project is based on a model/view design, in which one class (the *Widget*) controls the input/output operations and the interactions with the users and another class (called *TempConverter* in this project), is in charge of data manipulations.

While you input the code in the IDE, be aware of what you are doing all the time. Always, try to understand how the program is globally implemented and why certain procedures have been used.

Create a new **Qt GUI Application**. Name it as you like, eg. *Converter*.

Create a template application based on **QWidget**. Call the class *Widget* for simplicity.

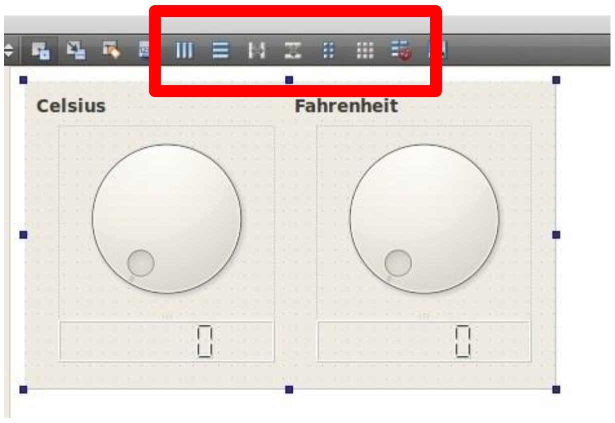
Now add two *QGroupBox* widgets, each containing a *QDial* widget and a *QLCDNumber* widget.



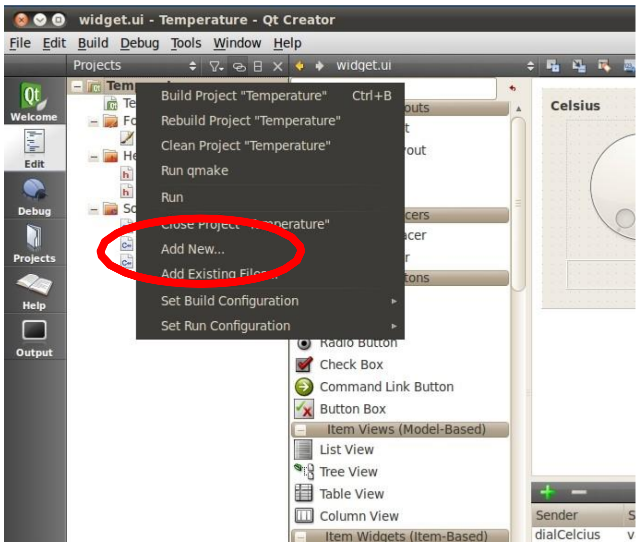
Rename and re-title the widget according to the table below. You can change the name in the property editor. The text can be changed from the property editor.

Widget	**Property**	**New value**
Dial Left	Name	dialCelsius
Dial Right	Name	dialFahrenheit
LCDNumber Left	Name	LCD_Celsius
LCDNumber Right	Name	LCD_Fahrenheit

Try to use the layout facility of the IDE to organize those widgets properly on the form. Explore yourself what happens if you click those buttons:



Now, create a new Class in the project. You can do this by right-clicking on the project name and clicking on "*Add New…*". Click on *"C++ Class"* and name it *TempConverter*. Use *QObject* as base class for the new class.


widget.ui - Temperature - Qt Creator
File Edit Build Debug Tools Window Help
Build Project "Temperature" Ctrl+B
Rebuild Project "Temperature"
Clean Project "Temperature"
Run qmake
Run
Close Project "Temperature"
Add New...
Add Existing Files...
Set Build Configuration
Set Run Configuration
Check Box
Command Link Button
Button Box
Item Views (Model-Based)
List View
Tree View
Table View
Column View
Item Widgets (Item-Based)
Celsius
Sender
dialCelcius

Note: It is a good practice in Qt to derive new classes from the *QObject* class, whenever possible. This is very helpful, especially in term of memory management and Qt-specific features usability.

Input the class declaration as follow:

```
class TempConverter : public QObject
{
    Q_OBJECT

public:
    TempConverter(QObject *parent = 0);

    int tempFahrenheit();

public slots:
    void setTempCelsius(int);
    void setTempFahrenheit(int);

signals:
    void tempCelsiusChanged(int);
    void tempFahrenheitChanged(int);

private:
    int m_tempCelsius;      //notation: "m_" private member variable
};
```

Note the QObject parent argument on the constructor and *Q_OBJECT* macro. You need this in order to add signals and slots to the class.
To avoid infinite loops we must have a "current" temperature. In this example we have decided to keep it in Celsius (`m_tempCelsius`). As we use integers throughout, the application will not be very accurate from a temperature conversion point of view, but you can make it better later on, if you like.

After the class declaration, move to the implementation and write the two methods below:

```
void TempConverter::setTempCelsius(int tempCelsius)
{
    if(m_tempCelsius == tempCelsius) // this is to avoid infinite loop
        return;

    m_tempCelsius = tempCelsius;

    emit tempCelsiusChanged(m_tempCelsius);
    emit tempFahrenheitChanged(tempFahrenheit());
}
```

And

```
void TempConverter::setTempFahrenheit(int tempFahrenheit)
{
    int tempCelsius = (5.0/9.0)*(tempFahrenheit-32);
    setTempCelsius(tempCelsius);
}
```

At this point, the implementation of your class related to the model is almost ready. Now you need to connect the user interface (the *Widget* class) with you model class *TempConverter*. For doing this, you need to use the instruction *connect* provided by Qt, which you already know from previous tutorials.

The following are the connections you need, in order to connect the *dialCelsius* widget with your model class *TempConverter*. You should write these instruction in the *Widget* constructor, right after *ui->setupUi(this)*.

```
QObject::connect(ui->dialCelcius, SIGNAL(valueChanged(int)), tc, SLOT(setTempCelsius(int)));
QObject::connect(tc, SIGNAL(tempCelsiusChanged(int)), ui->dialCelcius, SLOT(setValue(int)));
```

Of course, since you need to connect the *Widget* with the model class, you also need a *TempConverter* instance in your *Widget* class. This means that you need to create an association between the two classes. In practice, your Widget class declaration must have a pointer to a *TempConverter* object, called *tc* in this example:

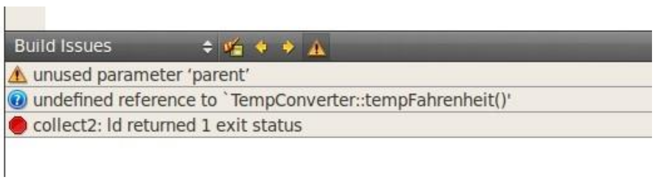
```
TempConverter *tc.
```

Work out yourself where you should write the actual creation of the object with the command *new*, in this way:

```
tc = new TempConverter(this);
```

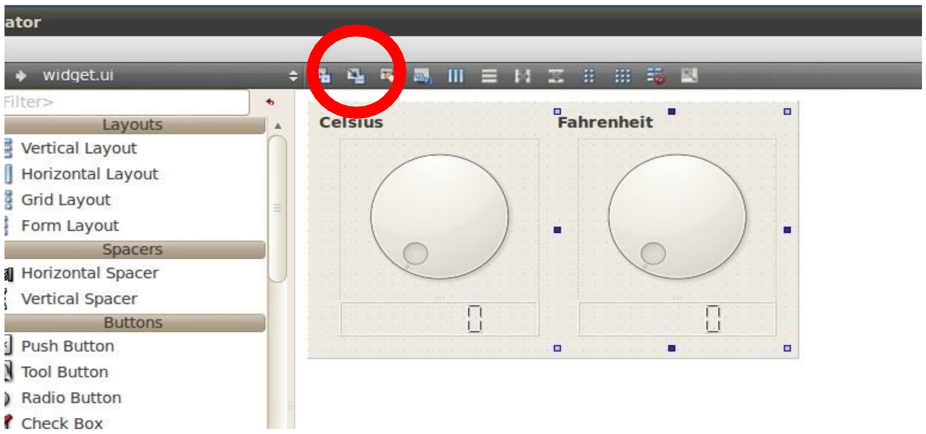
Now, if you have done everything properly, try to compile. You should get an error. Below how it looks like in Linux:

tempconverter.h
widget.cpp
widget.h
widget.ui

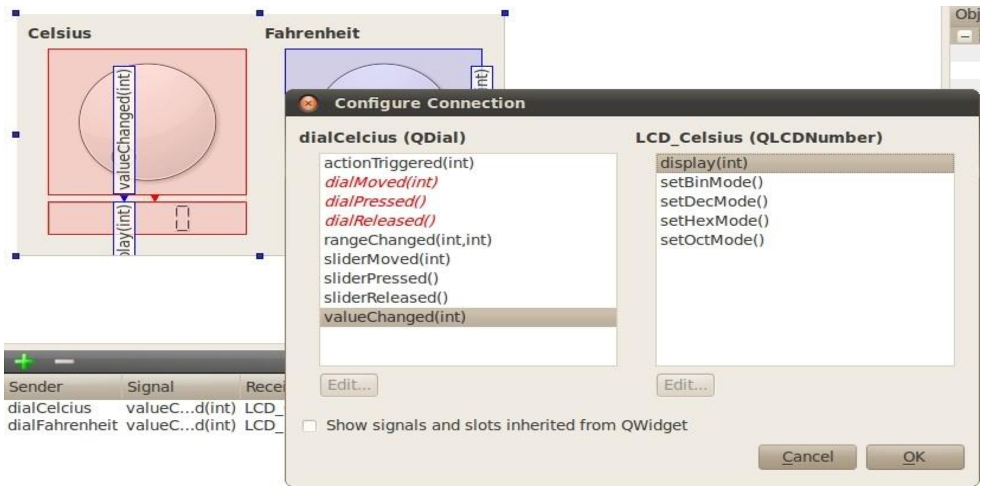


```
unused parameter 'parent'
undefined reference to `TempConverter::tempFahrenheit()'
collect2: ld returned 1 exit status
```

The declaration of *TempConverter* declares the method *int tempFahrenheit()* which is not implemented. We need to implement this method. This is something that you can do yourself (you can find the implementation at the end of this document, but try as much as you can to imagine how it should be implemented).
After the required implementation is done you can compile and the program would run. You can notice that the two dials work, but they are not connected to the LCDNumbers and to each other, therefore you don't get anyconversions.
You can connect the dials with the LCDNumbers directly from the IDE. Just go to the widget form and click on the button shown below. The IDE will switch to the *signal/slot editor*:



Now, click on a dial and drag it to the LCDNumber, a red arrow will show you the connection and a window will pop-up. On the window, click on the signal *valueChanged(int)* of the *Dial* and on the slot *display(int)* of the *LCDNumber.* Click OK to connect the two widgets:



This will make the two dials work. When you activate the dial, the connected LCDNumbers will show the correspondingnumber.

To conclude, you need to make the last two connections that will enable the program to convert temperatures....you just have to write the last two lines of code: Two more connections very similar to those already written.

Code implementation for the missing method

```
int TempConverter::tempFahrenheit()
{
    int tempFahrenheit = m_tempCelsius * (9.0/5.0) + 32;
    return tempFahrenheit;
}
```

FILE LIST PROJECT

This tutorial asks you to write a simple application that allows the user to populate a list with strings added by the user itself. The content of the list can be saved in a file and loaded into the application, by reading from a file.

The user interface of the application is the following:

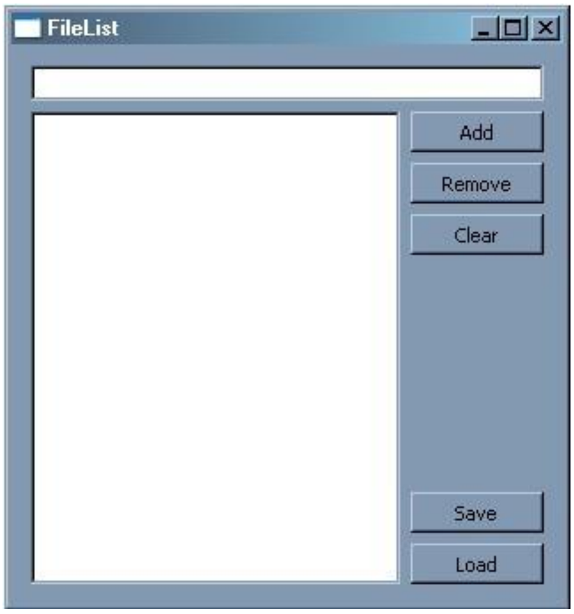


Figure. 1 User interface for the application FileList.

Doing this tutorial will allow you:

- to practice with input and display widgets.

- To practice with writing and reading from files in Qt

While you input the code in the IDE, be aware of what you are doing all the time. Always, try to understand how the program is globally implemented and why certain procedures have been used.

Create a new **Qt GUI Project**. Name it as you like, eg. FileList.

Create a template application based on QWidget. Call the class Widget for simplicity.

Arrange the widgets on the form according to Figure 1.

You will need a LineEdit widget, a ListWidget and 5 buttons, labeled “Add”, “Remove”, “Clear”, “Save”, and “Load”, respectively (see also Figure2).

After the form is created, check that you can resize it correctly. This can be obtained by adding a “layout” to the form. You can try and choose the layout that you prefer. The following resizing behaviours shouldbe obtained:

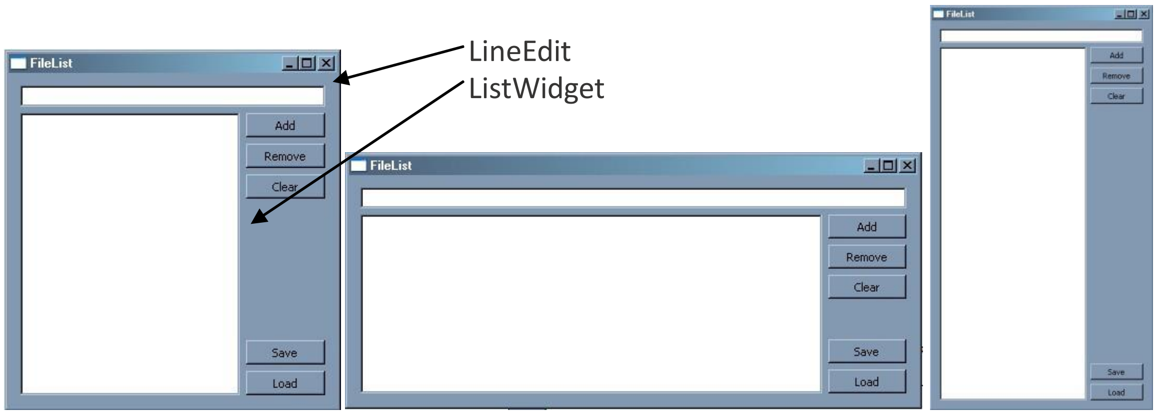


Figure 2.

Now, you have to implement the slots for every buttons.

The behaviours that you are required to implement are the following:

-Button *Add*: A string written on the LineEdit is added and correctly displayed in the ListWidget;

- Button *Remove*: The item of the list selected by the user is removed from the list;

- Button *Clear*: All items in the list are removed and permanently deleted.

- Button *Save*: The content of the list is saved on a file named “*FileList.txt*”;

- Button *Load*: The content of the file is loaded and displayed in the ListWidget.

Since you should be now familiar with ListWidget and LineEdit, you can implement the slots for the buttons Add, Remove, and Clear on your own.

Notice that, when selecting an item in the list, the returned value of the method QListWidget::selectedItems() is a QList<QListWidgetItem *> containing all the selected items in the list. If you need an help on this, you can find a solution at the end of the document.

After the behaviours of the three buttons are implemented, the task now is to implement the slot for buttons Save and Load, in which you need to write and read from a file.

a) The slot Save shall save the list as a text file using QTextStream. Each item of the list shall be given a line in the text file. This lets you verify the results using a plain text editor.

In the following, you will find the steps you need to do for implementing the functionality. As usual, refer to the documentation for the correct implementation of the single C++ instructions. At the end of the document you will find a solution.

The code shall perform the followingsteps:

Create a QFile object for the file name “FileList.txt” and open that file object for writing.

When the file has been opened, create a QTextStream object for that file.

Write the items to the stream using the << operator. Use the following code to iterate over the items.

```
for (int i=0; i < listWidget->count(); i++)
{
// your code for writing to the stream
}
```

When all items have been written, close the file object using QFile::close().

Before continuing, verify that the file does contain the items of the list, one item per line.

b) The next step is to implement the Load slot.

The code shall be implemented much like the save slot, but read the file. Use the QTextStream::atEnd() function to determine whether all lines of the file have been read. Use the QTextStream::readLine() function to read a complete line of the file.

As before, refer to the documentation for the correct implementation of the single C++ instructions. At the end of the document you will find a solution for the slot.

Do not forget to close the file object when the entire file has been read.

Now verify that the program loads and saves as expected. Also, ensure that you can load the same file repeatedly correctly, i.e. press the Load button several times in a row.

After this you should be able to add and remove items from the list and clear the list.

Moreover, you should be able to save the content of the list in a file and populate the list by loading the content from afile.

As a final task, you may want to give the user the possibility of choosing the name of the file she wants to save, by writing the file name on an additional LineEdit widget. This is an easy task, try to work out yourself how to do it.

Cross-platform test:

Copy the relevant file (.pro, .ui, .h, .cpp) on an external support and switch Operating System.

Open the .pro file with the Qt Creator and build and run the application (try to solve any building issue yourself, if you can). Be sure it runs as expected under the other OS.

Try to load a file created on one OS within another and observe the application behaviour.

Solutions for slots:

Remove

```
void Widget::on_m_removeBtn_clicked()
{
      foreach (QListWidgetItem *item, ui->m_listWidget->selectedItems())
      delete item;
}
```

Save

```
void Widget::on_m_saveBtn_clicked()
{
      QFile m_file("mytext.txt");
      m_file.open(QFile::WriteOnly);
      QTextStream m_stream(&m_file);
      for (int i=0; i<ui->m_listWidget->count(); i++)
      {
            m_stream << "item " << ui->m_listWidget->item(i)->text() <<
            "\n";
      }
      m_file.close();
}
```

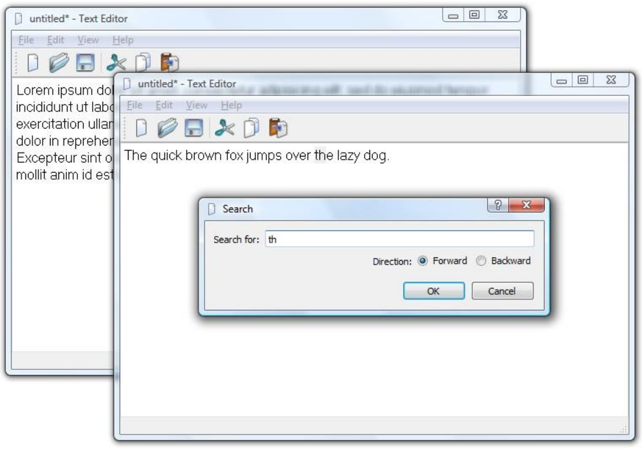
Load

```
void Widget::on_m_loadBtn_clicked()
{
      QFile m_file("mytext.txt");
      m_file.open(QFile::ReadOnly);
      QTextStream m_stream(&m_file);
      int i=0;
      while (!m_stream.atEnd())
      {
            ui->m_listWidget->addItem(m_stream.readLine());
            i++;
      }
 }
```

WORD PROCESOR PROJECT – PART 1

This tutorial is extracted by an official Qt tutorial prepared by NOKIA.

The task is to develop a fully fledged desktop application – a text editor with support for multiple documents, saving and loading, undoing, the clipboard, etc. Below you can see the final interface.



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.

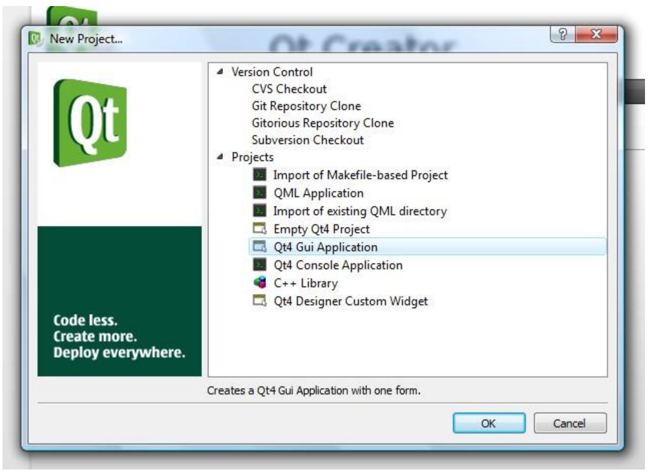
Since you have already some experience with Qt, here the instructions will be less detailed. This means that you are expected to look up classes in the Qt reference documentation, and that adding header files and other trivial code is up to you.

The implementation of the fully functional program will easily take more than one hour. Nevertheless, I suggest you to finish all this tutorial. It contains many feature you will need for the preparation of yourcoursework.

While you input the code in the IDE, be aware of what you are doing all the time. Always, try to understand how the program is globally implemented and why certain procedures have been used.

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.

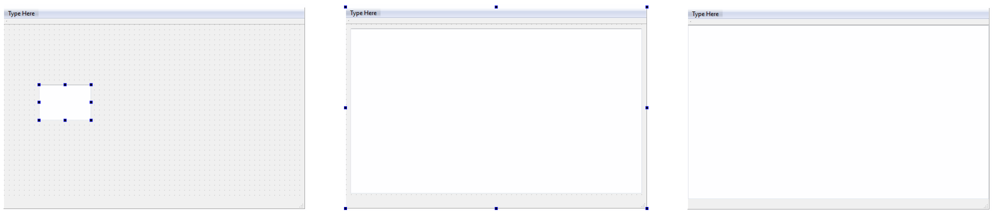
Review these files and verify that the project builds and runs.

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.

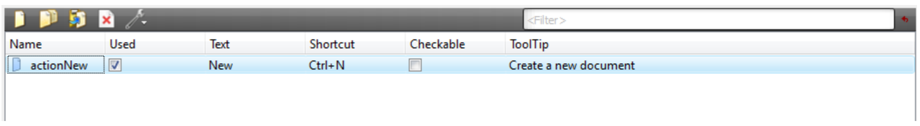
Now you are ready to start working on the user interface. Start by opening the mainwindow.ui file in Designer mode.

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.



Name	Used	Text	Shortcut	Checkable	ToolTip
actionNew	✓	New	Ctrl+N	☐	Create a new document

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

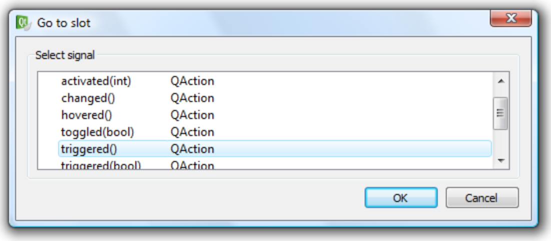
Actions are added to the user interface through drag and drop. Drag the actionNew action onto the toolbar.

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).

NOTE: If you enter "&File" as the title of the menu you get a shortcut, "File", 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.



```
activated(int)    QAction
changed()         QAction
hovered()         QAction
toggled(bool)     QAction
triggered()       QAction
triggered(bool)   QAction
```

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?

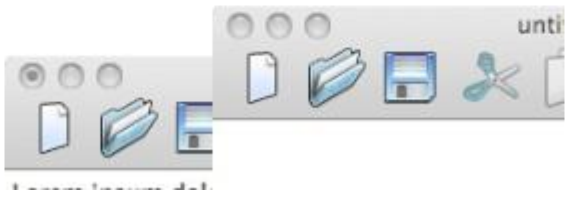
Modifying and Closing

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.

NOTE: 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 MacOS 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.

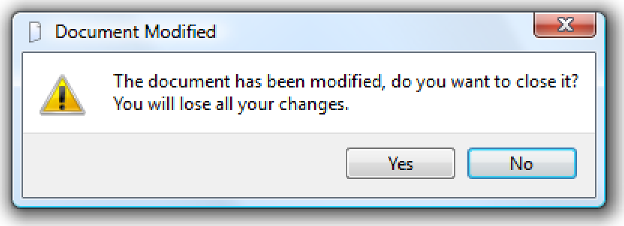
```
void MainWindow::closeEvent(QCloseEvent *e)
{

}
```

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.


Document Modified

The document has been modified, do you want to close it?
You will lose all your changes.

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.

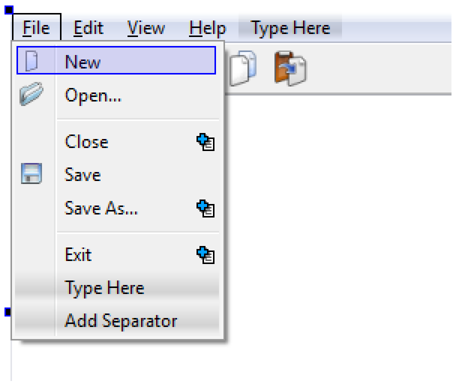
Having implemented the close method, the *File – Close* and *File – Exit* menu actions should work as expected.

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.

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 filename.
3. Attempt to openfile 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.

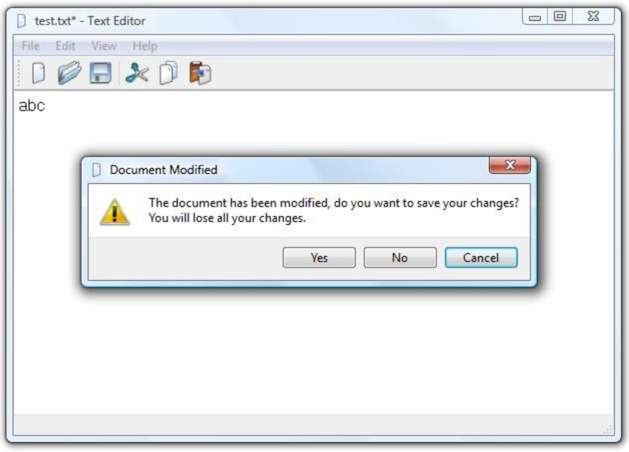
Text	Name	Icon	Shortcut	ToolTip
Save	actionSave	💾	Ctrl+S	Save document
Save As...	actionSaveAs		Ctrl+Shift+S	Save document as

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.


test.txt* - Text Editor

abc

Document Modified

The document has been modified, do you want to save your changes?
You will lose all your changes.

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

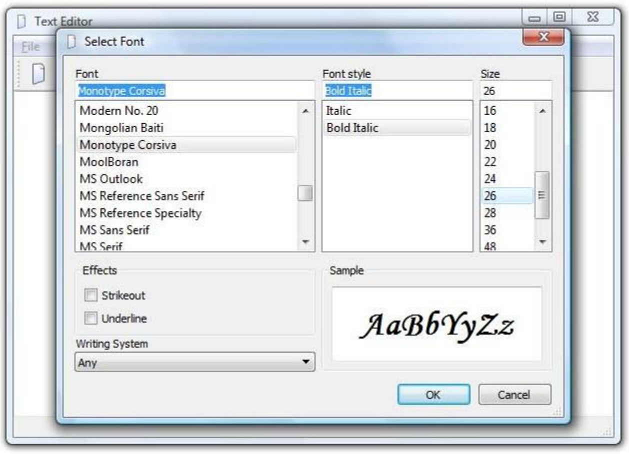
- Verify that you can load documents.
- Verify that you can save documents (and properly reload them).
- Verify that, when loading, saving and saving as, the window title is updated with the file name as expected.
- Verify that, when loading, saving and modifying documents, the windowModified state is 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.

WORD PROCESSOR PROJECT – PART 12

This tutorial is extracted by an official Qt tutorial prepared by NOKIA.

You will work on the previous project and will add new advanced setting functionalities, such as modifying the font and implementing clipboards operations. 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.

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.

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.

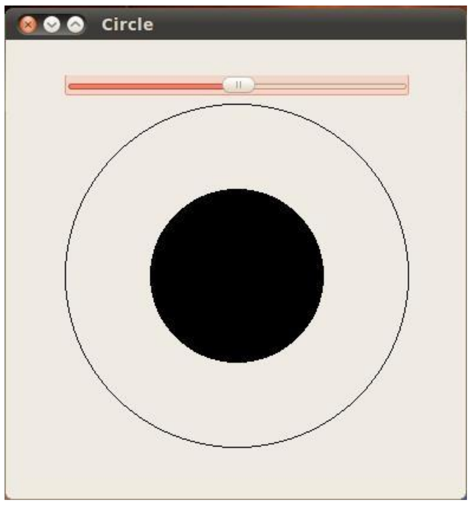
Self Check

- Check that *Help – About 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.

DRAW FIGURE PROJECT

This tutorial will guide you through the main steps that you need to do in order to create an application that displays a figure on a Widget. As usual, you will be asked to complete some parts of the code alone. Obvious c++ instructions, such as including headers files or some declarations, are not specified.

The program you are going to write is very simple. It is constituted by a main Widget that contains a Horizontal Slider and an additional widget that displays a circle. The inner part of the circle can be filled in black. The size of the inner filled circle can be changed by moving the slider on the windows. The following is the user interface:



This tutorial is based on the creation of a new widget (the circle) that you have to write and add to the main widget manually. That is, you will create the widget interface from scratch, without the help of QtCreator. The created widget will be then added to the project manually.

Doing this tutorial will allow you:

- to practice with input and display widgets;
- to create manually your own widget without the IDE;
- to implement *signals* and *slots* for creating the custom widget interface;
- to paint a widget by the code. This is useful in any Qt graphic application;
- to practice with the use of inheritance.

The tutorial is divided in two tasks. The first task is mostly guided, and allows to create the custom widget. The second task will ask you to write your own code for adding another figure to display in the main widget. This must be done by inheriting from the previous class.

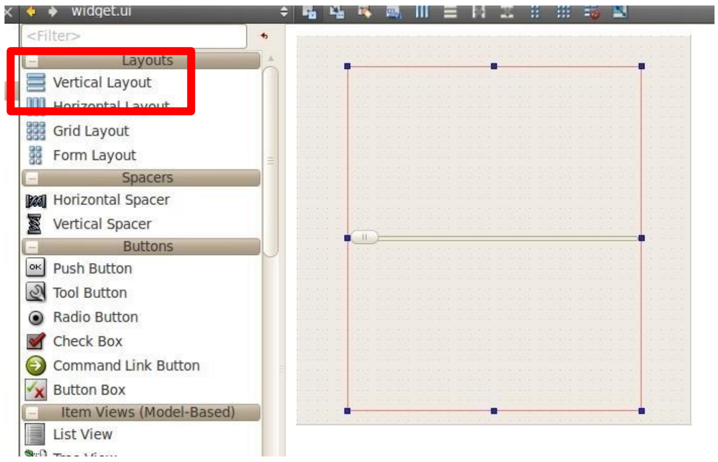
While you input the code in the IDE, be aware of what you are doing all the time. Always, try to understand how the program is globally implemented and why certain procedures have been used.

TASK 1

Create a new **Qt4 GUI Application**. Name it as you like.

Create a template application based on **QWidget**. Call the class *Widget* for simplicity.

Now add a *Vertical Layout* to the widget. Call it *layout* for simplicity. Add a Horizontal Slider to the layout and call it *hSlider*.



Now you are ready to write the code of your custom widget.
First of all, add a new class to the project from the IDE (see tutorial_3) and call the class *MyCircle*. This class must inherit from *QWidget* class, since you want to create a new widget.

MyCircle widget needs some interface, since you need to modify its behavior at run-time with the slider. For this purpose, you need a *setValue(int)* slot and valueChanged(int) signal. To complete the interface you need also a *value()* method to read the value held by the instance. Moreover, since the widget is painted by the code, you need to <u>reimplement</u> the *paintEvent* method. A method *drawMyFigure()* does all the graphics operations and is called by *paintEvent*. In the current implementation, a method called *heightForWidth* will keep the widget square and a *sizeHint* method provides a starting size.

The following is the class declaration of *MyCircle* widget class:

```
class MyCircle: public QWidget
{
    Q_OBJECT

public:

    MyCircle(int value, QWidget *parent);

    int value() const;
    int heightForWidth(int) const;
    QSize sizeHint() const;

public slots:
    void setValue(int);

signals:
    void valueChanged(int);

protected:
    void paintEvent(QPaintEvent *);

private:
    int m_value;
    void drawMyFigure();
};
```

Now, we can write the implementation of the class.
The constructor is the following:

```
MyCircle::MyCircle(int value, QWidget *parent) : QWidget(parent)
{
    m_value = value;
}
```

The methods that manage the size of the widget:

```
int MyCircle::heightForWidth(int width) const
{
    return width;
}

QSize MyCircle::sizeHint() const
{
    return QSize(100, 100);
}
```

Now, the implementation of the slot and the getter method of the class follow:

```
int MyCircle::value() const
{
    return m_value;
}

void MyCircle::setValue(int value)
{
    if (value < 0)
        value = 0;

    if (value > 100)
        value = 100;

    if (m_value == value)
        return;

    m_value = value;

    update();

    emit valueChanged(m_value);

}
```

Note the call to *update()*. By calling *update*, a repaint of the widget is triggered and a call to *paintEvent* is sent. Always remember that a you cannot draw the widget outside the *painEvent* method. Therefore, calling update and reimplementing *paintEvent* is the standard way of drawing a widget with Qt.

Now, it's the time to reimplement *paintEvent* and write the code for *drawMyFigure()*. Here is the simple code for *painEvent*:

```
void MyCircle::paintEvent(QPaintEvent *event)
{
    drawMyFigure();
}
```

The following is the code for *drawMyFigure()* :

```
void MyCircle::drawMyFigure()
{
    int radius = width()/2;
    double factor = value()/100.0;

    QPainter p(this);
    p.setPen(Qt::black);
    p.drawEllipse(0,0,width()-1,width()-1);
    p.setBrush(Qt::black);
    p.drawEllipse((int)(radius*(1.0-factor)),
                  (int)(radius*(1.0-factor)),
                  (int)((width()-1)*factor)+1,
                  (int)((width()-1)*factor)+1);
}
```

After the class implementation, you have to display the widget in the main widget of your project (*Widget*). Here is the code. !!!Remember to declare the *MyCircle* object in the class declaration!!!

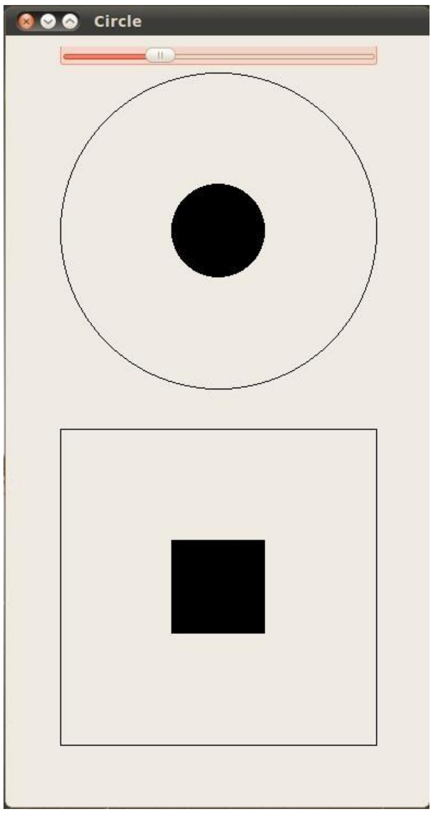
```
Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    c = new MyCircle(0, this);

    ui->setupUi(this);

    ui->layout->addWidget(c);

    QObject::connect(ui->hSlider,SIGNAL(valueChanged(int)),c,SLOT(setValue(int)));

}
```

At this point you should be able to compile and run the program. By moving the slider you should be able to modify the radius of the inner circle.

If your program works fine and the code written so far is clear to you, you can move now to the second task.

TASK 2

This task requires to build upon the code you have written for task1, by adding a new functionality. The user interface is similar to the one you used before. The new feature is just the fact that now you should be able to display a circle and a rectangle. Both must be resizable by acting on the slider. Here is an example of the GUI:



NOTE: This is a real toy problem and it is not a good example of hierarchal structure. Nevertheless, it gives you the possibility to experiment in a simple set-up some properties of inheritance and to practically implement inheritance in a c++ environment. The basic implementation this task suggests can be easily reorganised in a more correct structure. This can be a kind of TASK 3, in case you want to try a different implementation.

It is better to use a new project for developing the new program. Just copy the previous project in a new folder and open the new copied project with QtCreator.

You are free to write your own code for doing this task. The only requirement is that the new class for drawing the rectangle must inherit from *MyCircle* class. This means that the only method you should write is actually *drawMyFigure()* for the new class. Of course, you need to adjust your codeaccordingly.
Look at the lecture notes and on the help to find out the logic and the correct syntax.
NOTE: from a derived class you cannot access private members of the base class.
NOTE: you can invoke the base class constructor from the sub-class constructor

OPTIONAL TASK 3

As you can notice, inheriting a rectangle form a circle it is not the best logical way of organizing an hierarchal structure for the type of objects we are dealing with. You can better organize the logic of your code by creating an abstract class, eg. *MyFigure*, which contains the methods you need for every specific figure. Then, every class related to a given figure will inherit from *MyFigure* and you only will have to implement the method that draw the figure itself. This organization saves you a lot of work and makes the code flexible.