Qt Cross-Platform Graphical User Interface (GUI)

Graphical user interfaces, GUIs, are very dependent on the underlying operating system, particularly in terms of look and feel, but all provide more or less the same basic functionality. To avoid learning how to develop user interfaces for each of the different environments, we can use the multi-platform build system called Qt, which integrates the functionality of CMake and of your favorite IDE. For example, the user interface for QtCalc1 application comprises a main window, with a title bar and the standard buttons to quit, minimize, and maximize the application, a menu bar, and a separate pop-up "About" dialog window. In Windows the menu bar is located inside the main window, just below the title bar. In OSX the menu bar replaces the system menu bar at the top of the screen. Note that the menu bar is not visible in the figures shown below, because it was running under OSX in a Mac.

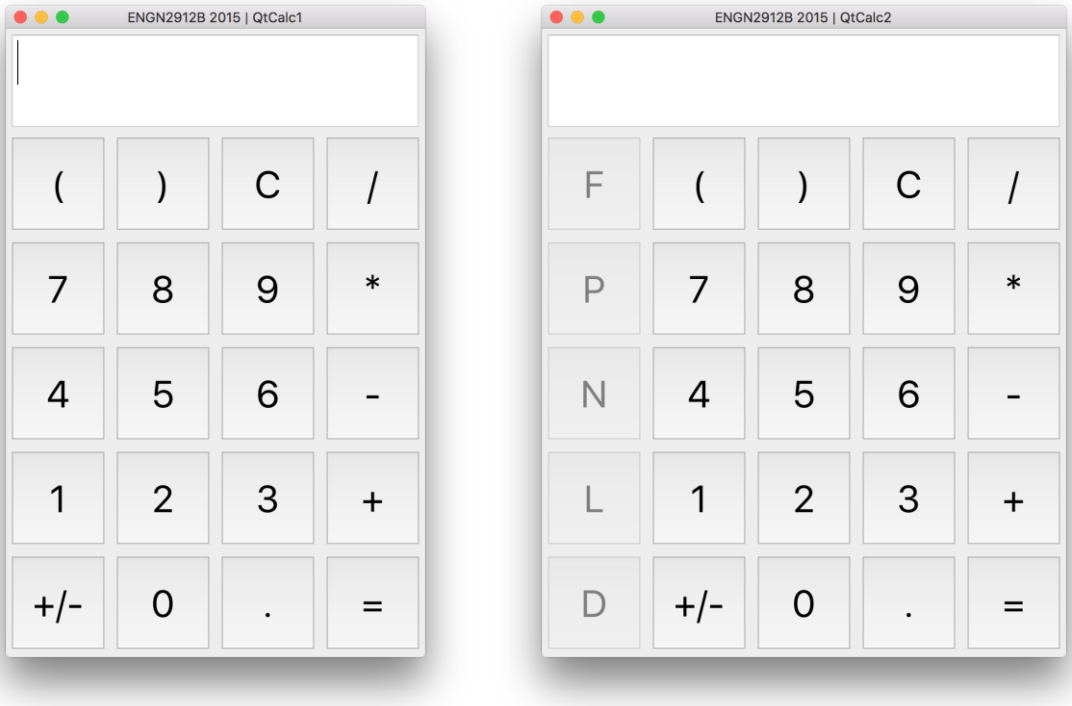


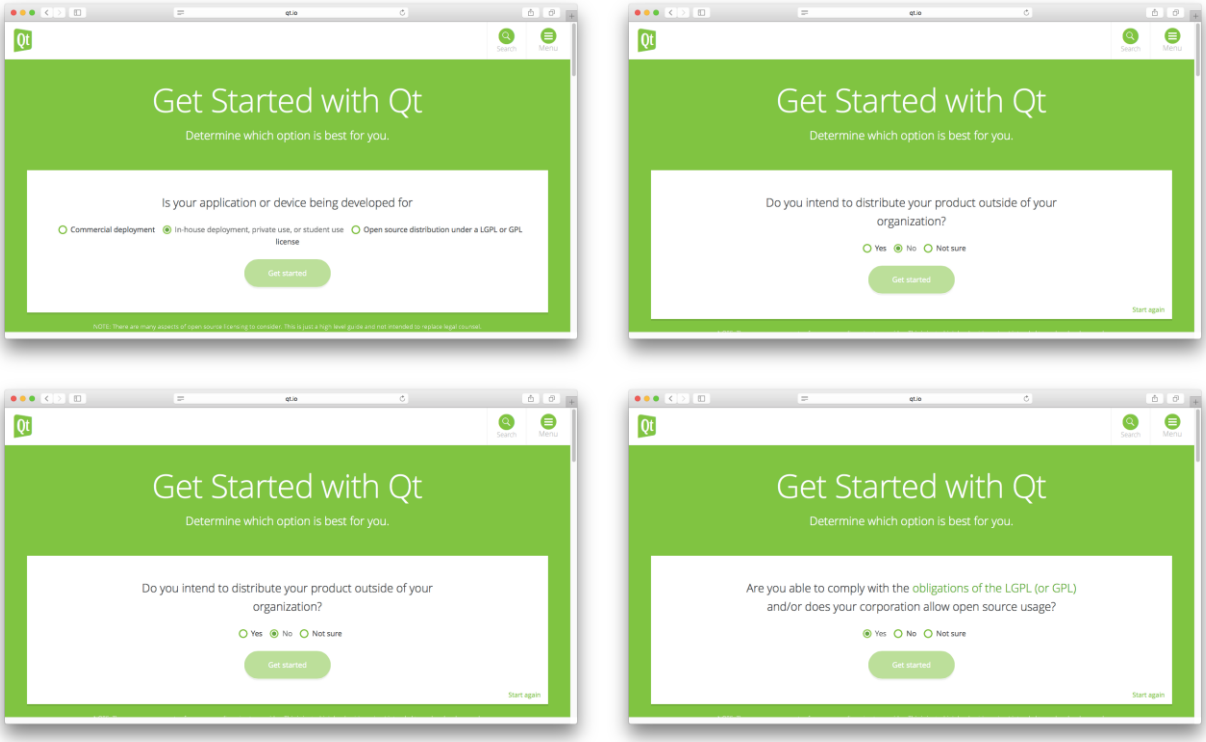
Figure 1: The QtCalc1 (left) and QtCalc2 (right) GUI applications

In both OSX and Windows, the main GUI window contains a number of "Widgets." For example, the QtCalc1 calculator has a QTextEdit widget at the top, where input expressions and output results will be shown, and twenty QPushButton widgets. An interactive application is an infinite loop which spends most of its time waiting for the user to interact with the widgets. Each widget is an instance of a particular class in a large hierarchy. In addition to the visual appearance, widgets can generate signals in response to user inputs. For each of these signals there is a custom signal handler, which will do something in response to the user action. In the QtCalc application the handlers for all the push button click events are implemented by the programmer.

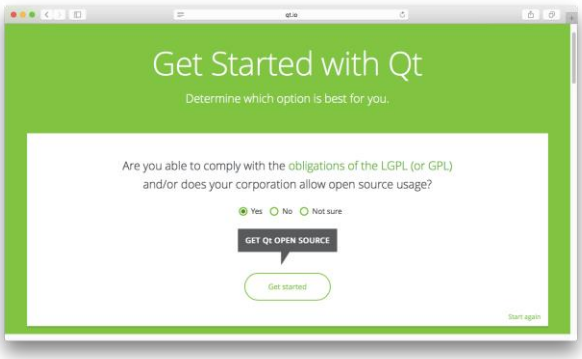
Qt uses a message passing paradigm comprising SIGNALS and SLOTS. Objects (widgets or other classes) can send signals and can have slots where signals can be received. Signals have to be connected to slots to make the application work. The user interface code of an application can be quite complex, and often much longer than the rest of the application code, but it can be designed independently and there are tools to create user interfaces which automatically generate most, if not all, of the user interface code. This is the case in Qt. This lecture will not allocate the substantial amount of time necessary for learning how to create user interfaces in Qt, but will provide an initial framework that can serve as a starting point. The remainder of this lecture describes how the user interface in QtCalc works.

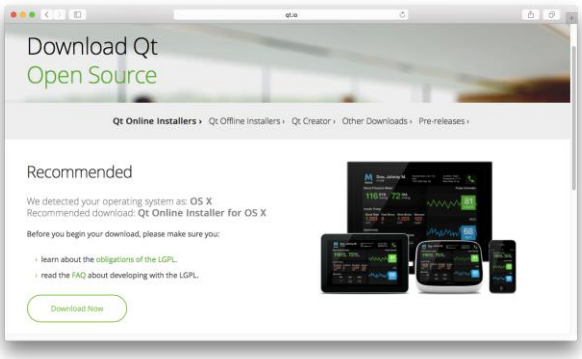
Installing Qt Creator

The first step is to install Qt Creator[1]. This is an integrated development environment, which can replace Xcode or Visual Studio, and includes functionality similar to CMake. Visit the Qt Download web site http://www.qt.io/download/ and follow the instructions to download the installer for you operating system. Select the free student version and accept the LGPL/GPL licenses. You will be asked to create an account. You can skip this step or use your Brown email address as your user ID.

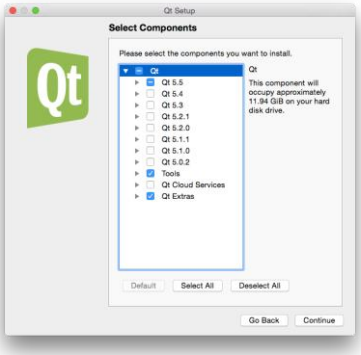


[1] Note that the Brown CCV has modules available for Qt version 4.8.3 and 5.7.0. All that is needed to compile and run Qt on the CCV is to type "module load qt/5.7.0" or add this line to your .modules file.



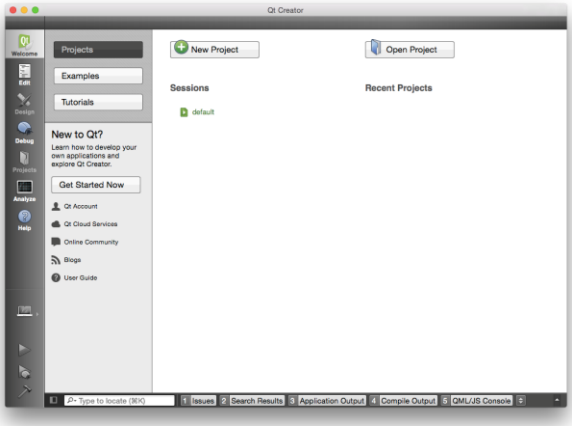


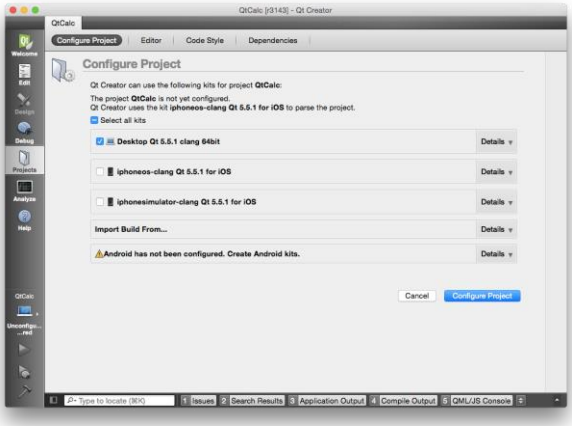
Once the download finishes, run the installer, wait until the installation process ends (this may take 20 to 30 minutes), and run Qt Creator for the first time.



You will be presented with the Qt Setup window, where you will have to login using your Qt Account user ID. If you don't have one, you can create it here. Then you will be presented with the Select Components panel. Leave the defaults for Tools and Qt Extras, and select only the latest version of Qt, which right now is Qt 5.5. Press Continue, leave the Launch Qt Creator selected, and press Done.

Qt Creator will start, and you will be presented with the user interface. Click the Open Project button, and select the QtCalc1.pro file located in the QtCalc1/src directory. You will be presented with the Configure Project panel. Select the Desktop Qt kit, and deselect the others (you should be able to build apps for your smartphone or tablet using the other kits). You should now be ready to compile, debug, and run your application.





The qmake tool

The installation of Qt Creator includes the command line application qmake, which generates Qt project files just as cmake creates IDE project files to compile your project for different platforms and operating systems. For example, it can be used to generate unix style Makefiles for your architecture. Then you would use the make command to actually compile and install your application in the bin directory. If you want to use qmake from the command line in a terminal, you first need to make sure that this directory is included in your PATH environment variable. You may need to edit your .bashrc file, located in your home directory so that every interactive bash shell can locate qmake. To compile your application in this way, after you included the directory where qmake is stored in your PATH, open a terminal, and go to your QtCalc1/build directory. At the command prompt, type the commands

```
$ qmake ../src
$ make
```

If the compiler is able to compile without errors, you should find your compiled application in your QtCalc1/bin directory.

In Windows the qmake tool is stored in a similar location, but you would need a command line window with the VisualStudio environment set up to operate in this way. Making qmake work with Cygwin doesn't seem to be so easy. The easiest solution here is to compile within Qt Creator as described above, which works exactly in the same way in OSX and in Windows.

Automatic Generation of GUI Code

It is often the case that the length of the code dedicated to the graphical interface is longer than the actual application code. But once the design of the GUI is completed, including the graphical layout of widgets, the code required to build the GUI can be generated automatically. You are not required to use it but Qt can generate the necessary code from a compact description of the layout contained written in the xml description language (similar to html, but more generic).

You will find two files named MainWindow.ui and AboutDialog.ui in the QtCalc1/forms directory. Open the MainWindow.ui file and take a look at the content. Try to understand the structure of this file. The AboutDialog.ui file describes the structure of the popup window which provides information about the application, author, compilation time, etc. You can write your own .ui files using a text editor. You can also use an additional tool integrated with Qt Creator, named Qt Designer, to create these files interactively. Qt Designer is a GUI that allows you to create layouts for user interfaces. It results into .ui files. We will not be able to spend much time exploring GUI design.

The Qt installation includes lots of examples of applications developed within Qt for you to learn more. They are located in subdirectories of the directories Qt/Examples and Qt/Extras. For this example you don't need to modify the provided .ui files. The files created automatically by Qt to implement the GUI will be saved to your QtCalc1/build directory, and then they will be compiled to build your application. But you can look at them to learn how the user interface code looks like. You can also write this code directly, skipping the automatic generation of code from .ui files. This is the content of the Qt project file MainWindow.pro, which is stored in the QtCalc1/src directory, which is equal to the file MainWindow.pro stored in the QtCalc2/src directory.

```
BASEDIR   = ..
DESTDIR   = $$BASEDIR/bin
FORMSDIR  = $$BASEDIR/forms
SOURCEDIR = $$BASEDIR/src
ASSETSDIR = $$BASEDIR/assets

NAME      = QtCalc1

CONFIG   += qt c++11
QT       += widgets

macx {
   ICON = $$ASSETSDIR/QtCalc.icns
}

HEADERS += \
      $$SOURCEDIR/MainWindow.hpp \
      $$SOURCEDIR/AboutDialog.hpp \
$$(NULL)

SOURCES += \
      $$SOURCEDIR/main.cpp \
      $$SOURCEDIR/MainWindow.cpp \
      $$SOURCEDIR/AboutDialog.cpp \
$$(NULL)

FORMS = \
      $$FORMSDIR/MainWindow.ui \
      $$FORMSDIR/AboutDialog.ui \
      $$(NULL)
```

If you prefer to write you own GUI code, you should remove the .ui file names from the FORMS variable. You should start by copying all of the AlgebraicTree*.h and AlgebraicTree*.cpp to the QtCalc1/src

directory, and editing the MainWindow.pro file so that your files are included in the HEADERS and SOURCES variables. Just to be consistent, instead of the .h extension, we will use the .hpp extension for C++ header files. Although it is not a requirement, many people use the .h extension only for C header files (.h and .c vs. .hpp and .cpp).

QtCalc1

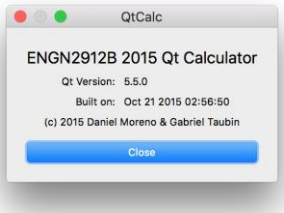
Qt contains a lot of libraries, some of which include classes which replace classes defined in the C++ standard libraries. For example, Qt defines the QString class, which we will have to use in this example because all Qt widgets this class to specify text. You are free to use std::string as well, and even to mix the two representations in your programs. In fact, QString has methods to convert to std::string. The application comprises a main window with a border, a title bar with the standard buttons, and a menu bar, which in this case we corresponds to a class which we have named MainWindow. This is the initial content of the QtCalc1/src/MainWindow.hpp file

```
#ifndef __MAINWINDOW_HPP__
#define __MAINWINDOW_HPP__

#include <QMainWindow>
#include "ui_MainWindow.h"
// #include "AlgebraicTreeExpression.hpp"

class MainWindow : public QMainWindow, public Ui::MainWindow {

  Q_OBJECT

public:

  MainWindow(QWidget * parent = 0, Qt::WindowFlags flags = 0);
  ~MainWindow();

public slots:

  // menu actions
  void on_quit_action_triggered();
```

```
    void on_about_action_triggered();

    // push button actions
    void on_number0Button_clicked();
    void on_number1Button_clicked();
    void on_number2Button_clicked();
    void on_number3Button_clicked();
    void on_number4Button_clicked();
    void on_number5Button_clicked();
    void on_number6Button_clicked();
    void on_number7Button_clicked();
    void on_number8Button_clicked();
    void on_number9Button_clicked();
    void on_addButton_clicked();
    void on_subtractButton_clicked();
    void on_multiplyButton_clicked();
    void on_divideButton_clicked();
    void on_leftParenButton_clicked();
    void on_rightParenButton_clicked();
    void on_decimalPointButton_clicked();
    void on_changeSignButton_clicked();
    void on_clearButton_clicked();
    void on_evaluateButton_clicked();

};

#endif  /* __MAINWINDOW_HPP__ */
```

The file ui_MainWindow.h is generated automatically. Q_OBJECT is preprocessor macro which is required. Note that the line does not end with a semi-colon. This MainWindow class is a subclass of QMainWindow and of UI::MainWindow. The class UI::MainWindow is defined in the automatically generated file ui_MainWindow.h. The programmer needs to implement some of the menu and pushbutton actions. The implementation of most of these functions require only one line of code, and are already implemented for you as examples. Complete Qt documentation and tutorials is available at http://doc.qt.io. In particular, you should read the descriptions of the QString class and of the QTextEdit class at http://doc.qt.io/qt-5/qstring.html and http://doc.qt.io/qt-5/qtextedit.html.

The on_quit_action_triggered() function is already implemented. It is called from the application menu to quit the application. The on_about_action_triggered() function is also already implemented. It pops up the AboutDialog window



You should edit the AboutDialog.cpp file and change the authors' names to your name. The following functions are all already implemented. Each one of them inserts one character at the cursor position into the string being displayed by the QTextEdit widget. With the cursor at its default position at the end of the string, a complete expression can be edited using the buttons.

```
    void on_number0Button_clicked();
    void on_number1Button_clicked();
    void on_number2Button_clicked();
    void on_number3Button_clicked();
    void on_number4Button_clicked();
```

```
void on_number5Button_clicked();
void on_number6Button_clicked();
void on_number7Button_clicked();
void on_number8Button_clicked();
void on_number9Button_clicked();
void on_addButton_clicked();
void on_subtractButton_clicked();
void on_multiplyButton_clicked();
void on_divideButton_clicked();
void on_leftParenButton_clicked();
void on_rightParenButton_clicked();
void on_decimalPointButton_clicked();
```

Note that you can also use the keyboard to edit the string, you can use the mouse or the keyboard to reposition the cursor, and you can cut and paste strings edited somewhere else into a QTextEdit widget. The on_clearButton_clicked() function clears the string shown by the QTextEdit widget.

You have to implement the on_changeSignButton_clicked() function to toggle the a sign between '-' and '+'. It should behave properly depending on whether the current string being displayed is empty or not, and whether the character preceding the cursor position is '+' or '-'. Read the comments in the QtCalc1/src/MainWindow.cpp files.

You also have to implement the on_evaluateButton_clicked() function. First of all, this function has to get the current string from the QTextEdit widget (as a QString). Then it should convert the string to a C-style string, and create an instance of your AlgebraicTreeExpression using the corresponding constructor. If the parser succeeds, it should evaluate the resulting tree, convert the resulting value to a QString, and display the QString in the QTextEdit widget. On the other hand, if your parser fails because the string syntax is not valid, we would like you to redisplay the original string, but painting in black color the characters that the parser was able to process before it encountered the error, and the painting the remaining of the character in red color. This task may require you to modify your implementation of your AlgebraicTreeExpression::_parse() function, in such a way that it would be possible to know where along the input string a parsing error had occurred. You may need to add new variables and public and/or private function to your AlgebraicTree classes to be able to accomplish this. To print a string in two colors you can use the QTextEdit::insertHtml() function. For example, assuming that the expression has already been split into two parts, qStrParsed and qStrNotParsed, both represented as instances of the QString class, the following statement will construct a new instance of the QString class, where qStrParse and qStrNotParsed are concatenated, but qStrParsed is painted black and qStrNotParsed is painted red. Note the use of \" to include the symbol " within a const string.

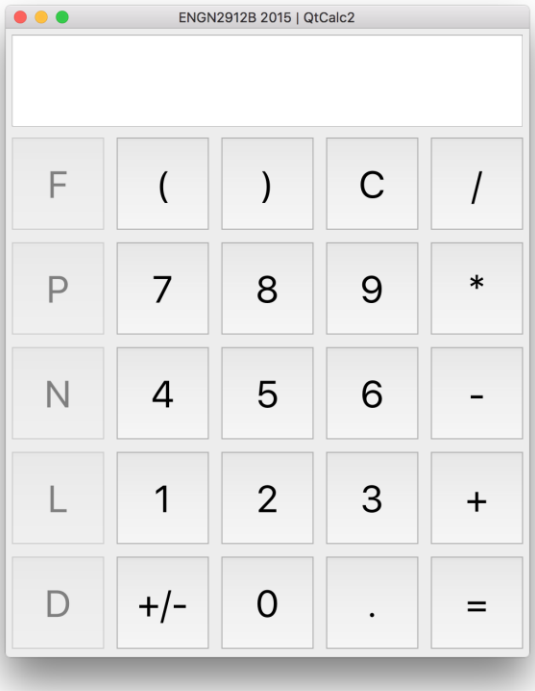
```
QString  qStr =
    "<font color=\"black\">" + qStrParsed    + "</font>"+
    "<font color=\"red\">"   + qStrNotParsed + "</font>";
```

Make sure that you reset the color to black when you call the on_clearButton_clicked() function, because even an empty string may be painted red, and when you insert more characters, they will all be painted red. That is, the QTextEdit::clear() method clear the string, but it does not clear the colors.

QtCalc2

After you complete your implementation of QtCalc1, you should copy your new files from the QtCalc1/src directory to the QtCalc2/src directory, and you should also copy the changes that you made to the MainWindow class from the QtCalc1 version to the new QtCalc2 version. Make sure that you can compile QtCalc2, and that all the functionality that you had implemented in

QtCalc1 is still working in QtCalc2. Only after that you should attempt to add new functionality.



We have added a new column of push buttons, which will require you to implement five new functions

```
void on_firstExprButton_clicked();
void on_prevExprButton_clicked();
void on_nextExprButton_clicked();
void on_lastExprButton_clicked();
void on_delExprButton_clicked();
```

The goal here is to preserve the expressions trees in a container, without repetition, so that we can retrieve any previous expression, modify it, and evaluate it again. You have to select a proper STL container, such as vector<AlgebraicTreeExpression*>, list<AlgebraicTreeExpression*>, or set<AlgebraicTreeExpression*> to store the expression trees, and make it a member of the MainWindow class. Here you will have to allocate your expression trees in the heap, rather than in the stack as you were in QtCalc1. Then you need methods to navigate the container to select a previous expression for display. Note that once you start typing a new expression after pressing the clear button, or when you start modifying an expression retrieved from the container, you are dealing with a new expression which is not yet stored in the container. The first, previous, next, and last buttons imply that the container is linearly ordered. You may want to make the container circular or not. That is, if you press the previous button after you retrieve the first element, you have to decide whether to go to the last element of the container or to stay with the first. You should enable or disable these buttons depending on these conditions, on whether the container is empty or not, etc. The delete button will remove from a container an expression tree retrieved from the container and being displayed. The delete button should be disabled whenever the container is empty and when the expression being displayed is not one of those stored in the container. To prevent duplication in the container, you may need to implement a function to decide whether or not two expressions are equal or not. That is, you may need to implement the operator

```
Bool AlgebraicTreeExpression::operator == (AlgebraicTreeExpression& rhs);
```