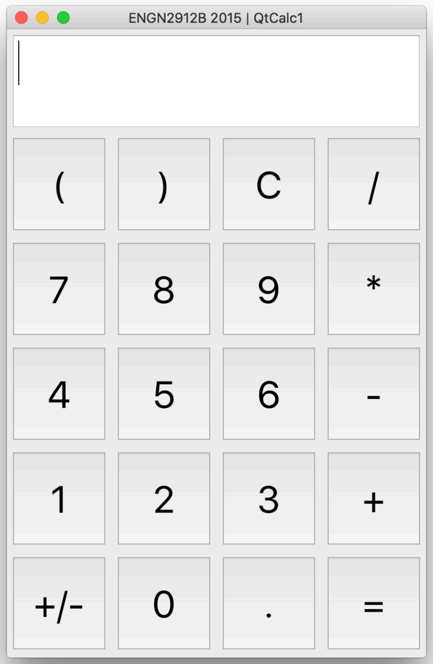
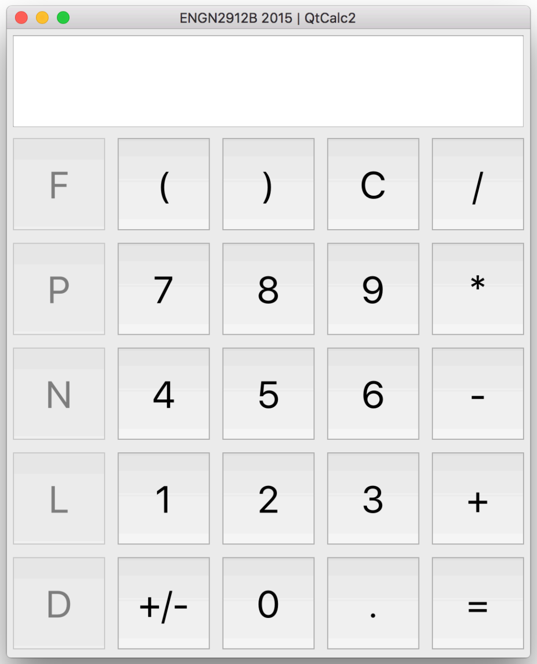
Brown University ENGN2912B Fall 2017

Scientific Computing in C++

Assignment 08 | QtCalc1 and QtCalc2

In this assignment we will create a simple user interactive calculator by adding a user interface to the code that you have developed in previous assignments. In fact we are going to develop two versions; the second one with additional functionality. This is how these applications will look like in OSX.





User interfaces 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 the different environments, we are going to 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 kill, 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. The menu bar is not visible in the figure shown above, because I am running under OSX in a Mac. In both cases the main window contains a number of so-called 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 you have to implement a signal handler, which will do something in response to the user action. In the QtCalc application you will have to implement the handlers for all the push button click events. 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. That is the case in Qt. We will not be able to spend too much time learning how to create user interfaces. In the case of the QtCalc1 application we have done it for you. But we will explain how it works.

Downloading the Assignment files & Special Submission Instructions

Download the file `08_Gabriel_Taubin.zip` file from the canvas course site, and expand it, resulting in a directory named `08_Gabriel_Taubin`. As usual, you should first change the name of the top directory from `08_Gabriel_Taubin` to `08_FirstName_LastName_OS`. This directory should contain two subdirectories named `QtCalc1`, and `QtCalc2`. Each one of these two subdirectories will have subdirectories named `assets`, `bin`, `build`, `forms`, and `src`. The `bin`, `build`, and `src` directories will serve the same functions as before. The `assets` directories are used here to store icon files, and the `forms` directories are used to store xml files, which describe the user interface design in a compact form. Qt uses these files to automatically create the C++ interface and implementation files needed to construct the graphical user interface. We will discuss the details later on. You will first develop `QtCalc1`, by adding your `AlgebraicTree` implementation, perhaps with some changes and additions, and later you will extend the functionality into `QtCalc2`. The two projects can be compiled and run using CMake as received, and you should do that to test your Qt installation. The user interfaces are complete, but the pushbuttons will not perform any function. In this assignment your task is to implement these functions. You will add your `AlgebraicTree` files and then you will modify some of your own and some of the given files. Once you complete your work, you should empty the `build` and `bin` directories, and pack the remaining files for submission. Zip the top directory `08_FirstName_LastName_OS` and submit the resulting zip file `08_FirstName_LastName_OS.zip`. . For example, if I had to submit this assignment, I would name my top directory `08_Gabriel_Taubin`, containing my own versions of QtCalc1 and QtCalc2, and would submit the file `08_Gabriel_Taubin.zip` If you don't follow these rules, your assignment will be returned. If you need to explain anything, add comments to the source file.

Installing Qt

Before proceeding you need to install Qt in your computer. Visit the following page and follow the instructions:

https://www1.qt.io/download-open-source

Using CMake to build the application

As in previous assignments you should use CMake to build the two applications. You will find a CMakeLists.txt file in the QtCalc1/src directory, and another one in the QtCalc2/src directory. In both cases the AlgebraicTree files are commented out. Open the interactive CMake application, specify the sources and build directories, and press the Configure button. Select the type of project files that you want, and press the Done button. If CMake starts the configuration process but fails, open the CMakeLists.txt file, edit the following path to the location of the Qt cmake directory in your computer

```
# set to the proper directory in your machine
set(Qt5_cmake_DIR /Users/taubin/Qt/5.7/clang_64/lib/cmake)
```

and uncomment this block:

```
# This is needded for CMake to find the Qt libraries
# set(CMAKE_PREFIX_PATH
#     ${Qt5_cmake_DIR}
#     CACHE PATH "Qt5 cmake directory" FORCE)
```

Close the CMake gui, reopen it, and press the Configure button again. The new CMakeFiles.txt also solves the problem of having to specify the value of CMAKE_INSTALL_PREFIX by hand, as you had done in previous assignments.

After these changes CMake should be able to populate the build directory without errors, and the application should compile as provided, using the native IDE or unix makefiles.

Automatic Generation of GUI Code

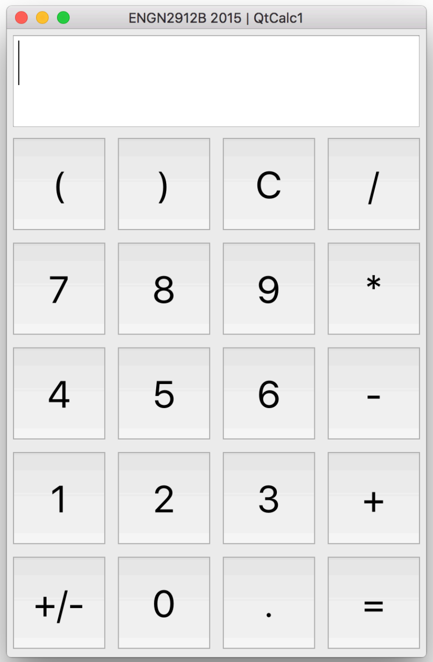
It is often the case that the length of the code dedicated to the graphical interface is longer than the actual application code. But once the design of the GUI is completed, including the graphical layout of widgets, the code required to build the GUI can be generated automatically. You are not required to use it but Qt can generate the necessary code from a compact description of the layout contained written in the xml description language (which is similar to html web page description language). 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 assignment 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 CMakeLists.txt file, which is stored in the QtCalc1/src directory, which is almost identical to the CMakeLists.txt file stored in the QtCalc2/src directory.

```
cmake_minimum_required(VERSION 2.8.11)

set(NAME QtCalc1)

project(${NAME})

# Find includes in corresponding build directories
set(CMAKE_INCLUDE_CURRENT_DIR ON)
# Instruct CMake to run moc automatically when needed.
set(CMAKE_AUTOMOC ON)

# This variable is used to locate the bin directory where the
# application will be installed
set(CMAKE_INSTALL_PREFIX
    ${PROJECT_SOURCE_DIR}/..
    CACHE PATH "Project bin directory" FORCE)

# you can comment the following line
message("CMAKE_INSTALL_PREFIX = ${CMAKE_INSTALL_PREFIX}")
```

```
# set to the proper directory in your machine
set(Qt5_cmake_DIR /Users/taubin/Qt/5.7/clang_64/lib/cmake)

# This is neededed for CMake to find the Qt libraries
# set(CMAKE_PREFIX_PATH
#     ${Qt5_cmake_DIR}
#     CACHE PATH "Qt5 cmake directory" FORCE)

# you can comment the following line
message ("CMAKE_PREFIX_PATH = ${CMAKE_PREFIX_PATH}")

# alternative way to make CMake find the Qt libraries
# set(Qt5core_DIR    ${Qt5_cmake_DIR}/Qt5core)
# set(Qt5Widgets_DIR ${Qt5_cmake_DIR}/Qt5Widgets)
# set(Qt5Svg_DIR     ${Qt5_cmake_DIR}/Qt5Svg)

# Find the QtWidgets library
find_package(Qt5core)
find_package(Qt5Widgets)
find_package(Qt5Svg)

set(FORMS_DIR ../forms)
set(ASSETS_DIR ../assets)

add_definitions(-DNOMINMAX       -D_CRT_SECURE_NO_WARNINGS       -D_SCL_SECURE_NO_WARNINGS       -
D_USE_MATH_DEFINES)

#add current dir to include search path
include_directories(${PROJECT_SOURCE_DIR})

#library list
set(LIB_LIST Qt5::Core Qt5::Widgets Qt5::Svg)

set(HEADERS
  MainWindow.hpp
  AboutDialog.hpp
#  AlgebraicTreeExpression.hpp
#  AlgebraicTreeNode.hpp
#  AlgebraicTreeNumber.hpp
#  AlgebraicTreeOperation.hpp
) # HEADERS

set(SOURCES
  main.cpp
  MainWindow.cpp
  AboutDialog.cpp
#  AlgebraicTreeExpression.cpp
#  AlgebraicTreeNode.cpp
#  AlgebraicTreeNumber.cpp
#  AlgebraicTreeOperation.cpp
) # SOURCES

qt5_wrap_ui(FORMS
 ../forms/MainWindow.ui
 ../forms/AboutDialog.ui
) # FORMS

qt5_add_resources(ASSETS
) # ASSETS

if (APPLE)
  add_executable(${NAME} MACOSX_BUNDLE ${APP_ICON}
      ${HEADERS}
      ${SOURCES}
      ${FORMS}
      ${ASSETS}
  ) # add_executable
else(APPLE)
  add_executable(${NAME}
      ${HEADERS}
      ${SOURCES}
      ${FORMS}
      ${ASSETS}
  ) # add_executable
endif(APPLE)

target_compile_features(${NAME} PRIVATE cxx_right_angle_brackets cxx_lambdas)

target_link_libraries(${NAME} ${LIB_LIST})

# install application
set(BIN_DIR ${CMAKE_INSTALL_PREFIX}/bin)
install(TARGETS ${NAME} DESTINATION ${BIN_DIR})
```

If you prefer to write you own GUI code, you should remove the .ui file names from the FORMS variable. You should start your assignment by copying all your AlgebraicTree*.hpp and AlgebraicTree*.cpp to the QtCalc1/src directory, and editing the CMakeLists.txt file so that your files are included in the HEADERS and SOURCES variables.

QtCalc1



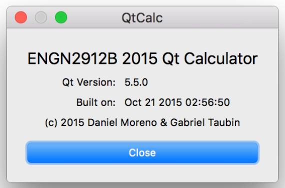
Qt includes a lot of libraries, some of which include classes which replace classes defined in the C++ standard libraries. For example, Qt defines the QString class, which we will have to use in this assignment because all Qt widgets use the QString class to specify text. You are free to use std::string 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. Your task is 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



ENGN2912B 2015 Qt Calculator

Qt Version: 5.5.0

Built on: Oct 21 2015 02:56:50

(c) 2015 Daniel Moreno & Gabriel Taubin

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.

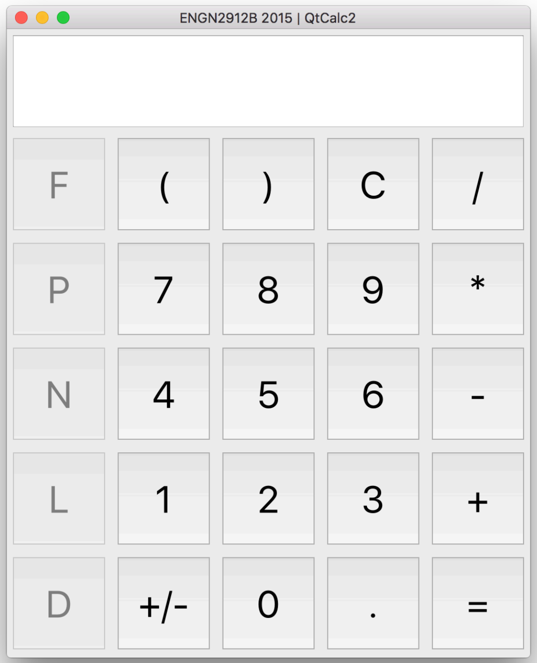
You also have to implement the `on_evaluateButton_clicked()` function. First of all, this function has to get the current string from the QTextEdit widget (as a QString). Then it should convert the string to a C-style string, and create an instance of your AlgebraicTreeExpression using the corresponding 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 painting the remaining of the characters 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);
```