9. GUI

Ján Dugáček

November 22, 2017

Table of Contents

Motivation

- For most users, command line programs are counter-intuitive and annoying to use
- It's practical to have a window-based application
- Many libraries exist to deal with this issue, we'll use Qt, because it allows the same code to be compiled on most platforms

Event-driven programming

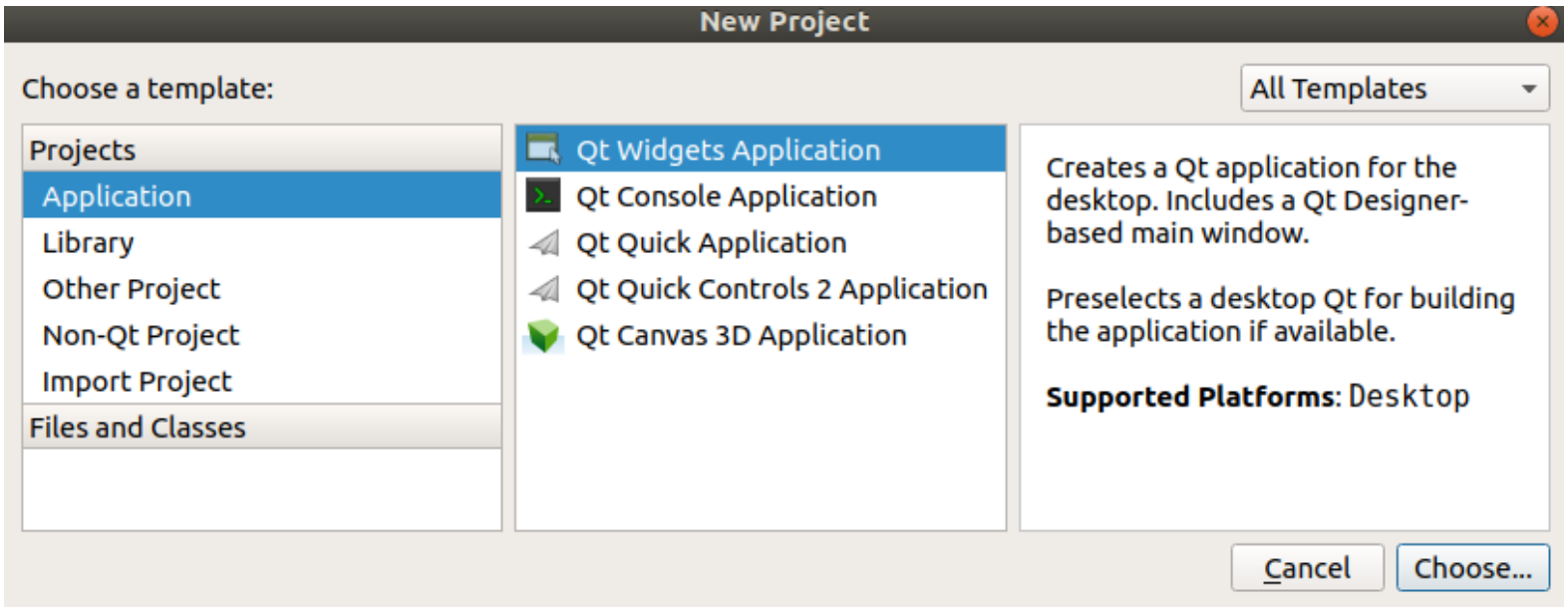
- Usually, a program with a *Graphical User Interface* (GUI) waits until the user interacts with it
- Interaction with the window triggers *callback functions* if they are set up; usually most of the program is executed when these functions are called
- This is different from terminal programs that are usually supposed to do their job and exit
- GUI programs require support from other programs to be drawn on screen, so it cannot be done without using libraries

Introduction

- Qt is a set of libraries that gives access to operating system specific functionality through a unified interface
- Its interface tends to be much more practical than the operating systems' interface
- It's easy to install (as long as you use QtCreator)
- Some of its features are quite weird (like: using its own *string* class that isn't very compatible with `std::string`), and it introduces elements alien to C++ (it converts these to C++ before compilation)

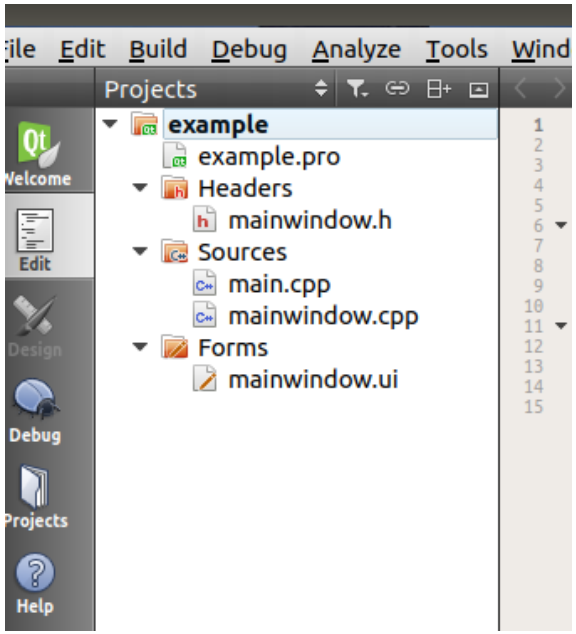
Creating a GUI program

- While normal projects can be converted to GUI programs in QtCreator, here's a way to create such a program at the beginning
- Go into the *New Project* window, select *Application* and *Qt Widgets Application*



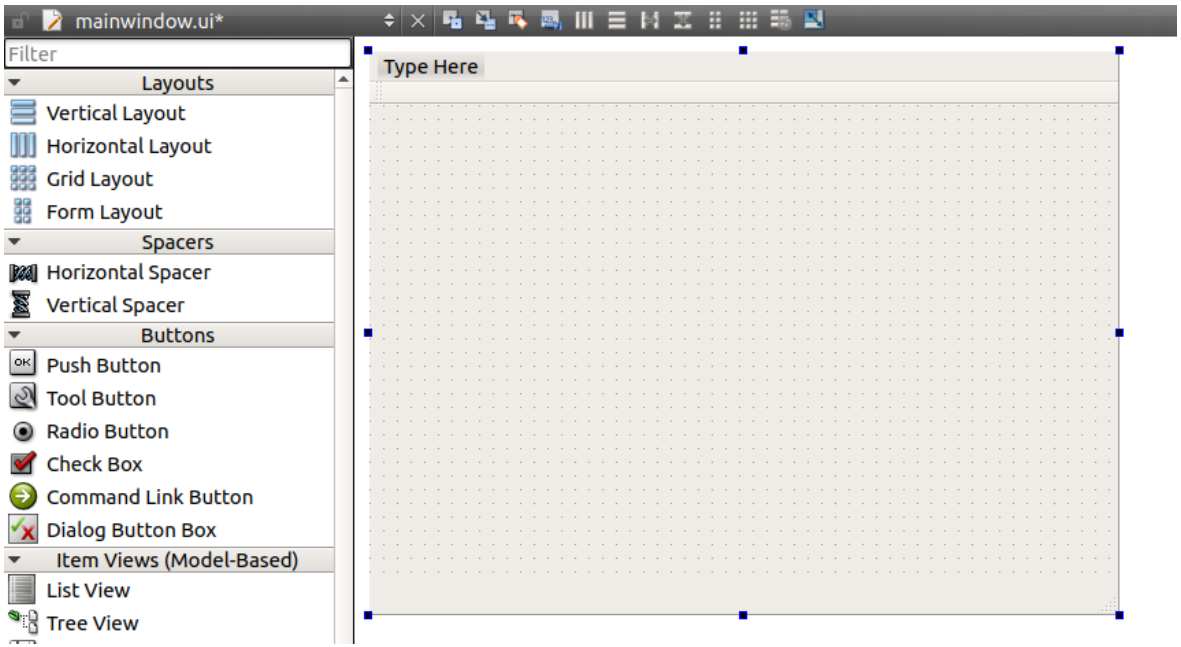
Editing the GUI

- Between the files created, double click the on ending with *.ui*



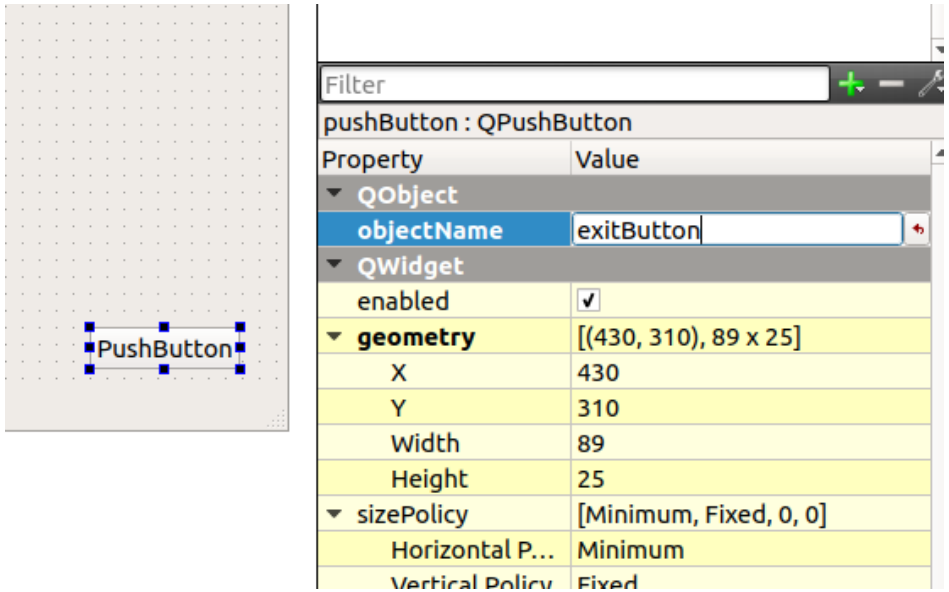
Editing the GUI #2

- The buttons, text fields and other things you put in the window are called *widgets*
- Add widgets to the window by dragging them on their place



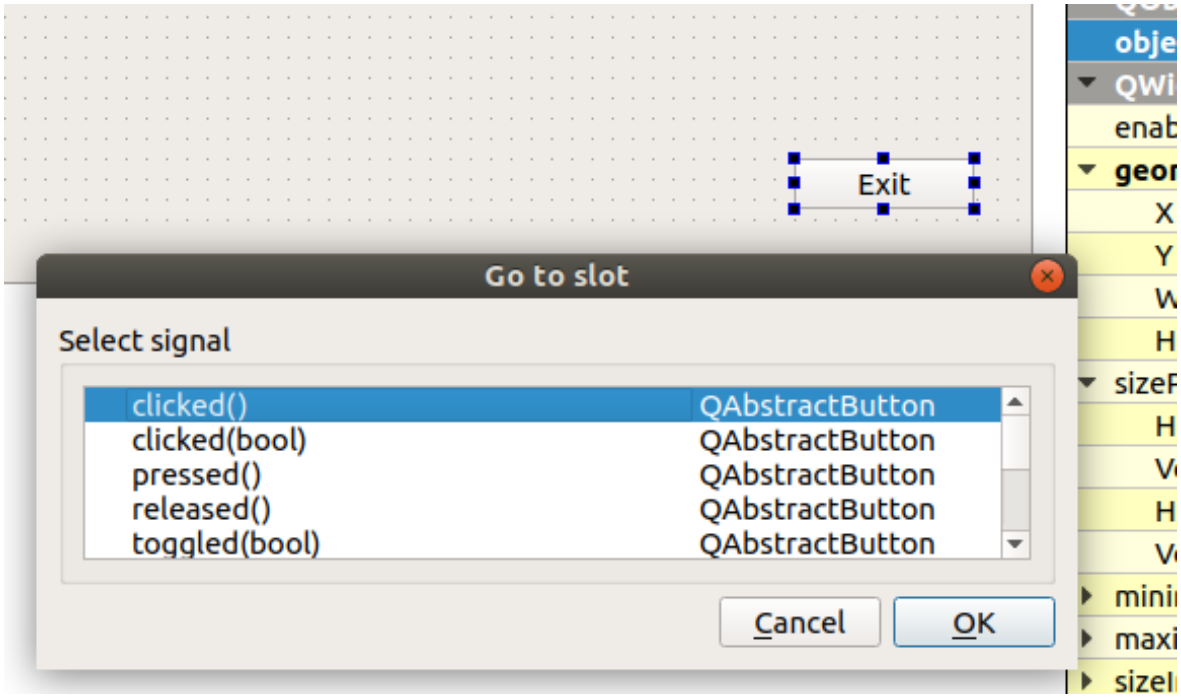
Editing the GUI #3

- You may change the properties of widgets on the bottom left side when the widget is selected
- You should give an appropriate name to each widget
- The text on widgets can be changed by double clicking them



Adding callbacks

- To add a callback, right click on the widget and select *Go to slot*
- Usually, you want callbacks to buttons being pressed, so select *clicked()*



Adding callbacks #2

- The previous will create a callback function and get it working
- It is like a regular member function

```
void MainWindow::on_exitButton_clicked()
{
    QApplication::quit();
}
```

Manipulating contents

- It generates a member variable `ui` that contains all widgets, use their methods to change contents or learn of their contents

```
void MainWindow::on_exitButton_clicked()
{
        ui->exitButton->setText("No, I won't!");
}
```

Exercises

1. Create a window with a button that changes its text on first click, then exits
2. Create a window that contains a few check boxes and text fields and a button that becomes an exit button if they are set properly

Advanced:

1. Create a simple calculator window

Reading files

- We should already be able to create a field where the user can write the name of a file, but that's not exactly a comfortable way of choosing files
- *QFileDialog::getOpenFileName* opens a open file window where you can select the file
- The second argument is the name of the window, the third argument is the folder it opens (empty means same folder as program) and the third argument is the file type it expects
- It returns `QString` that is quite annoying to convert to `std::string`

```
void MainWindow::on_openFile_clicked()
{
        std::string fileName = QFileDialog::getOpenFileName(
                this, tr("Open Data"), "",
                tr("data (*.csv)")).toUtf8().constData();
        std::ifstream file(fileName);
        std::string line;
        while (file >> line) {
                data_.push_back(std::stof(line));
        }
}
```

Saving files

- Files are saved almost identically to the way they are loaded

```
void MainWindow::on_saveFile_clicked()
{
        std::string fileName = QFileDialog::getSaveFileName(
                this, tr("Open Data"), "",
                tr("data (*.csv)")).toUtf8().constData();
        std::ofstream file(fileName);
        for (unsigned int i = 0; i < data_.size(); i++) {
                file << data_[i] << std::endl;
        }
}
```

Exercises

1. Create a window with a button that loads a file (with one column of numbers), multiplies all numbers in it by a user-set constant and saves it as the user demands

Advanced:

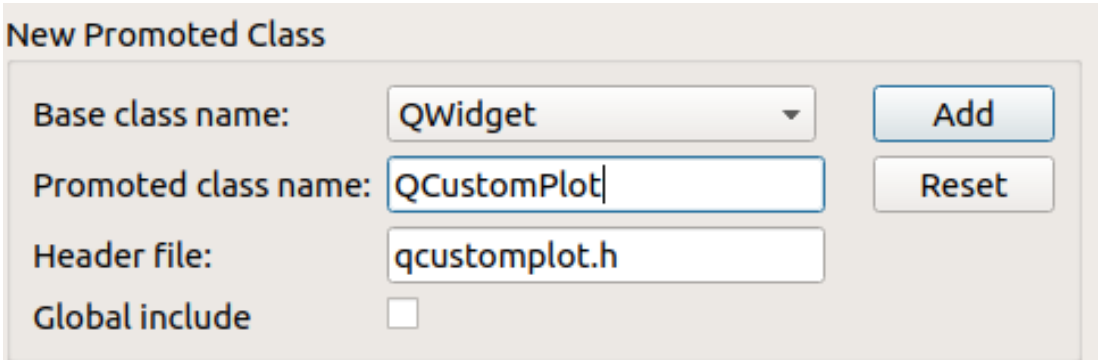
1. Create a calculator that perform operations on all numbers in the file, then saves it

QCustomPlot

- Qt has tools for vector graphics, but no chart widget and that sucks
- QCustomPlot is a small library that fixes the problem, adding a widget that does all kinds of graphs (including 3D graphs, heatmaps and other exotic plots)
- It consists of just two files, but they are huge (the source file is over 30000 lines long)

Adding to window

- QCustomPlot is expected to be used as a part of your program, not as an external library, it's made of one source file and one header file
- To use it, google it, download it, unpack it into the folder where your files are, then right click on your project in QtCreator, hit *Add Existing files...* and select these two files
- You also need to add `printsupport` to your `.pro` file, to a line containing `QT +=`
- Add the graph widget into your window by first adding a *Widget* (group *Containers*), then right clicking on it, selecting *Promote to...* adding `QCustomPlot` to promotions, selecting it and promoting the widget



Plotting

- The following code will clear old plots, name the axes, set ranges and add a graph line

```
void MainWindow::on_plotButton_clicked()
{
        ui->plot->clearGraphs();
        ui->plot->addGraph();
        ui->plot->xAxis->setRange(0, 6);
        ui->plot->yAxis->setRange(0, 70);
        ui->plot->xAxis->setLabel("line");
        ui->plot->yAxis->setLabel("value");
```

Plotting #2

- The following code will add some data to the plot and make the changes visible

```
        for (unsigned int i = 0; i < data_.size(); i++) {
                ui->plot->graph(0)->addData(i, data_[i]);
        }
        ui->plot->replot();
}
```

- QCustomPlot's website contains a lot of information about other tricks that can be done with QCustomPlot

Exercises

1. Create a program that is opened from the command line, reads the name of a file (as first argument) and plots it in a window (expected form are two columns, x and f(x))
2. Set minimum and maximum of axes accordingly to the data
3. Change the previous program to open and plot files with any number of columns (and thus plots)

Advanced:

1. Create a program that plots a user-defined function (supporting only a few operations)

A few suggestions

- Use *layouts* to position your widgets, it's more annoying, but makes them resize properly when the window is resized
- Don't neglect the command line, it allows the program to be controlled by different programs, which opens door to new applications of your program
- If the GUI program is not simple, give it a core that can be used both from the window and from the command line

Homework

- Create a program that is opened from the command line, reads the name of a file (as first argument) and plots it in a window (expected form are several columns, x and $f_1(x)$, $f_2(x)$, ...)
- Axes should rescale accordingly to the minima and maxima of values

Advanced homework:

- The same, but check out QCustomPlot's website to colour the lines differently and allow user interaction