|  |
| --- |
| Programming Labs Handbook |
| Essentials Edition |
| Version 5.10 |

You hear you forget.

You see you remember.

You do you learn.

TABLE OF CONTENTS

Lab 1: Signals and Slots 3

Lab 2: Event Handling 6

Lab 3: Item Containers 9

# Lab 1: Signals and Slots

Objectives

* Learn different ways to connect signals to signals or slots
* Use of overloaded signals/slots in function-based signals and slots
* Understand the parameter passing in case of direct and queued connections
* Delayed functionality: How to invoke a function, when the control returns to the event loop

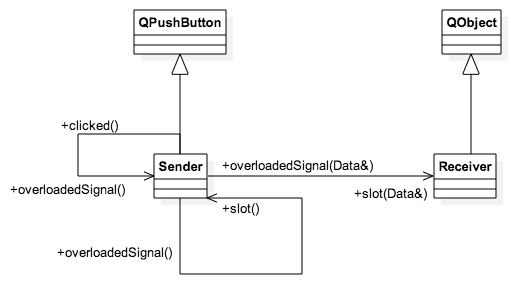
References

* labs/signals Lab starting point
* labs/solutions/signals One possible solution

Overview

You are provided with a program, in which you are asked to connect signals and slots between two objects: Sender and Receiver. Look at the diagram below. The idea is to test different kind of connections and how data may be passed using different connection types: automatic, queued, blocking queued, and of course direct. Use Qt debugging aids (QDebug) to see, when your slots are called and what kind of data they receive.

There are three parts in the lab. First, you are asked to connect a signal to a signal in Sender. In the second part, the same signal is connected to a slot in Sender, which emits an overloaded signal with Data argument. This signal is connected to slot() in Receiver, as shown below. Finally, the slot() function in Receiver should be called from Sender by using the meta object. These examples should include all typical use cases for signals and slots.



Practicalities

1. Open the project starting point into the QtCreator. You may run the program, but it does not do anything, until we connect some signals and slots. In some connections, we will pass a custom data type Data between the objects. Data has been defined in data.h/data.cpp. Sender is a simple QPushButton widget, while Receiver is a simple QObject sub-class, allowing other objects to connect to its slots.
   1. Instantiate Sender and Receiver objects in the main() function. Allocate Receiver in the heap and take care of proper de-allocation of Receiver object. There is not any other reason to allocate Receiver in the heap expect to test different ways to manage memory in Qt. Hint! Use QScopedPointer smart pointer.
   2. Define a signal and a private slot in Sender without any arguments. The signal corresponds overloadedSignal() and the slot slot() in the image above.
   3. Can you connect a signal to signal? Try to connect QPushButton::clicked() signal to the signal, you just added, and then your signal to your private slot in Sender. Add some debug statements to log slot function execution. Does this work?
   4. Now add an overloaded signal with Data & argument to Sender. Can you build the program now? The compiler should complain, it cannot resolve, which overloaded function pointer to use. Hint! You may define the overloaded version using a C++ syntax for a pointer to a member function.

void (Class:: \*pointerName)() = &Class::OverloadedFunction;

1. Let’s continue experimenting with signals and slots.
   1. Add a slot with Data & parameter in Receiver.
   2. In Sender slot function, instantiate Data object and emit a signal with the object argument.
   3. In the main() function, connect the overloaded Sender signal (with Data & parameter) to Receiver slot function.
   4. Change the parameter (Data) values in Receiver slot. Use Data::changeData() function.
   5. Print out the values in Sender slot after the overloaded signal has been emitted. Have the original values changed?
   6. What if you change the connection type to Qt::QueuedConnection? Does it work now? Why or why not? Hint! now it’s asynchronous
2. Finally, add missing functionality in Sender constructor, which results Receiver slot to be called without QPushButton to be clicked. You will need a queued connection to do this, but note that you have nothing to connect to, as Receiver object has not been created yet. QMetaObject will provide invokeMethod() function to do that. The function will queue an event, which is handled after the main thread returns to the event loop. Event handling will result the queued function to be called, which will results that Receiver slot will be called. This is because the connection between Sender and Receiver is created, before we enter the event loop. In general, the function is very useful to notify observer (call slots) in classes, not derived from QObject.

# Lab 2: Event Handling

Objectives

* Touch and gesture handling
* Key handling
* Event filters

References

* labs/slideShow Lab 2a starting point
* labs/solutions/slideShow One possible solution to 2a
* labs/eventFilter Lab 2b starting point
* labs/solutions/eventFilter One possible solution to 2b

Overview

There are two parts in this lab 2a and 2b. In the first part, you are provided with a Qt program, showing four images in a carousel. Your task is to implement both key handling and custom mouse-based gestures to rotate images in the carousel.

In the second part, you are requested to implement an application-wide (global) event filter to customize the standard mouse press event handling. You should implement an event filter, which moves all widgets to the right, when the right mouse button is pressed. Other mouse event and all other events it should be ignored.

Practicalities 2A

1. Open the project 2a starting point. You may run the program, but it just shows four images and nothing else. Your task is to add event handling to make the images rotate.



1. Implement keyPressEvent() function in SlideShowDialog. This function should allow the user to use left and right keys to scroll images left and right. The function has been provided to you, but the implementation is empty. Use signals scrollLeft() and scrollRight() to scroll the images, if left or right arrow key is pressed. In other words, if QKeyEvent::key() returns Qt::Key\_Left then you emit scrollLeft() signal and if it returns Qt::Key\_Right then you emit scrollRight() signal. Remember to call base class implementation QDialog::keyPressEvent(). Note that nothing happens yet, as event() function has not been implemented.
2. Implement the event() function in the same class. Check, if the event type is Gesture and in that case, call gestureEvent() function. Study the gestureEvent() function. How gestureDirection() is determined? What about gesture state? This is a good example about how to write custom gesture recognizers.
3. Now the first part is complete.

Practicalities 2B

1. The project starting point 2b provides you with a simple QMainWindow-based application. Open the project in Qt Creator, build and run. You will see three widgets on the screen; take a look at the MainWindow constructor to see how they are added to the window. Now we want to change the geometry of widgets using the event filter.
2. Click on the button and see what happens using both left and right mouse buttons. Notice the code in MainWindow that makes this happen. Clicking on the other widgets does nothing.
3. Your task is to implement the following functionality *without modifying the widgets on the screen (i.e. without sub-classing* QPushButton *or the other classes)*. This means you should use an event filter. Look at QObject::eventFilter() documentation.
   1. Check the event type using QEvent::type(). Call base implementation for all event types you are not interested in. In this lab, we are only interested in QEvent::MouseButtonPress.
   2. Cast the event object to the right sub-type. Look at QEvent and QInputEvent documentation to find the right sub-type.
   3. Use QEvent sub-type members to see, whether the button was right button. Qt class provides a lot of useful enumerations for you. Also the MouseButton enum.
   4. Cast the event receiver parameter to QWidget. Now you may use QWidget x() and y() members to read to current widget position and move() function to move the widget, e.g., 10 pixels to the right.
   5. Test your program. If it crashes, pay attention to the cast function, which you used. Use the debugger to see the crash reason.
4. In terms of performance, is this the most efficient way to implement custom event handling?

# Lab 3: Item Containers

Objectives

* Associative containers
* Non-mutable and mutable iterators
* C++11 ranged for loop
* Algorithms

References

* labs/itemContainers Lab starting point
* labs/solutions/itemContainers One possible solution

Overview

Your task is to store and manipulate QString type elements in the item container. Let’s assume we have 10,000 QString objects, which are stored in the container using a table index: x and y. Note that there is not a table container in Qt, so we have to use something else as a container class. To access a string element in the container, you are expected to use a QPoint as it provides x and y coordinates. The container type, you are expected to use, must keep the elements in the total order all the time without sorting.

Practicalities

1. Open the project itemContainers from labs/. The project contains nothing else but the main() function. Your task is to implement the rest.
2. Create a container in the main() function. As it is not a QObject, there is no need to allocate it in the heap.
3. Your container needs a less than operator for two QPoints (operator<(const QPoint& p1, const QPoint& p2)) to keep the items in the total order.
   1. Implement the operator globally outside the main() function so that it returns true either if the x value of point p1 is less than the x value of point p2 or the x values of two points are equal and the y value of p1 is less than the y value of p2. In other cases, return false.
4. Fill your container with data items using traditional for loop. For example, an item may contain some text and an integer index, because it is then easy to study the actual elements in the container.
   1. Use an iterator to print out your container elements. You may stream your output to QtCreator Application Output pane using qDebug()function. (Requires QDebug header).
5. Remove every third element in the container and print out the result. Use the same mutable iterator first for removing and later for printing the results. Hint! Use toFront() method.
6. Set the value of every fifth element in the container to “hello n”, where n is the nth instance of the hello string. Implement using the mutable iterator.
7. Print out the container elements using C++11 range-based for loop.
8. Use standard algorithms to count the number of “hello” sub-strings in the container.