

# Object Oriented Programming - Lecture 11

Diana Borza - [diana.borza@ubbcluj.ro](mailto:diana.borza@ubbcluj.ro)

April 15, 2024

- Callback functions
- Qt signal and slots
- Meta-information about QObjects
- Qt UI elements (cont'd)

- When dealing with graphical user interfaces we often want to be notified about the user interface elements that have been modified (by the user).
  - we want to perform an action when the user presses a button.
  - when a value is chosen in a combobox, a list should be populated with different values etc.
- In many toolkits this is achieved using callbacks.
- A **callback** is any executable code (function) that is passed as an argument to other code (function); that other code is expected to call back (i.e. execute) the argument function when appropriate.

# Callbacks II

- A callback is a function that is called by another function, when an event happens.
- In order to be notified by a processing function that an event occurred, we pass a pointer to another function (**the callback**) to the processing function.
- The processing function then calls the callback when appropriate (when the event occurs).
- We already used callbacks: for example, when sorting an array using the `qsort()` function, we passed a pointer to the comparison function to `qsort()`.

## Callback example

### Progress notification

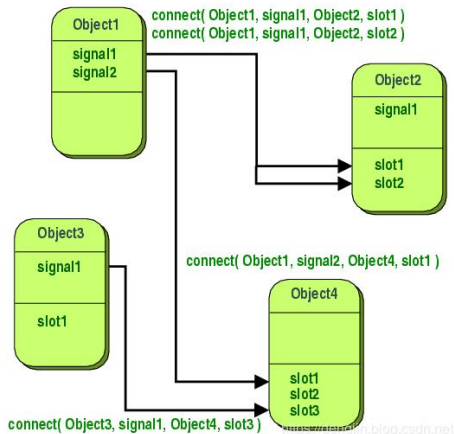
- Callback disadvantages:
  - if there are several notifications needed, we either need separate callback functions, or we could use generic parameters (`void*`), which cannot be verified at compile-time.
  - the processing function is coupled to the callback function (it needs to know its signature, its parameters).

# Signals and slots in Qt I

- Qt uses an alternative to callbacks: the **signals and slots mechanism**.
  - This is a central feature of Qt and probably the part that differs most from other frameworks.
- A **signal** is emitted when a particular **event occurs**.
  - Qt's widgets have many predefined signals, but signals can also be added to custom widgets and classes.
- A **slot** is a function that **is called in response to a particular signal**.
  - Qt's widgets have many predefined slots, but it is common practice to subclass widgets and add your own slots so that you can handle the signals that you are interested in.

# Signals and slots in Qt II

- The signals and slots mechanism is **type safe**: the signature of a signal must match the signature of the receiving slot.
- Only classes that inherit from `QObject` or one of its subclasses (e.g., `QWidget`) can contain signals and slots.
- Signals and slots are **loosely coupled**: A class which emits a signal neither knows nor cares which slots receive the signal.



# Signals and slots III

- Slots can be used for receiving signals, but they are also normal member functions.
- A slot does not know if it has any signals connected to it.
- Multiple signals can be connected to a single slot, and a signal can be connected to as many slots as needed.
- It is even possible to connect a signal directly to another signal. → the second signal is emitted immediately whenever the first is emitted.



- A signal is emitted when a particular event occurs.
- Signal can never have return types (i.e. they always return `void`).
- If several slots are connected to the same signal, the slots will be executed one after the other, in the order they have been connected, when the signal is emitted.
- When a signal is emitted, the slots connected to it are usually executed immediately (except for queued connections), just like a normal function call.
- Execution of the code following the emit statement will occur once all slots have returned.
- Signals are automatically generated by the *moc* and must not be implemented in the `.cpp` file.

- A slot is called when a signal connected to it is emitted.
- Slots are normal C++ functions and can be called normally; their only special feature is that signals can be connected to them.
- Using a signal-slot connection, slots can be invoked by any component.
  - A signal emitted from an instance of class A can cause a private slot to be invoked in an instance of class B, even if A and B are unrelated.

- Qt's meta-object system provides the signals and slots mechanism for inter-object communication, run-time type information, and the dynamic property system.
- The meta-object system is based on three things:
  - The `QObject` class: base class for objects that can take advantage of the meta-object system.
  - The `Q_OBJECT` macro inside the private section of the class declaration is used to enable meta-object features (signals, slots, runtime info.)
  - the Meta-Object Compiler (*moc*)
- *moc* is the program that handles Qt's C++ extensions (signals, slots, runtime info.).
- It is a code generator: it parses the header files and generates an additional C++ file that is compiled with the rest of the program.

# Meta Object Compiler II - moc

- The meta-object compiler takes all classes starting with the `Q_OBJECT` macro and generates a moc \*.cpp C++ source file.
- This file contains information about the class being "moc-ed" such as class name, inheritance tree and also the names and pointers to the signal and slot members.
- This means that emitting a signal is actually calling a function generated by the *moc*.
- Macros used by the *moc*:
  - `signals`
  - `emit`
  - `slots`
  - `SIGNAL`
  - `SLOT`

- The `QtMetaObject` class offers meta-information about Qt objects
  - **className()** returns the name of a class.
  - **superClass()** returns the superclass' meta-object.
  - **method()** and **methodCount()** provide information about a class' meta-methods (signals, slots and other invocable member functions).
  - **enumerator()** and **enumeratorCount()** and provide information about a class' enumerators.
  - **propertyCount()** and **property()** provide information about a class' properties.
  - **constructor()** and **constructorCount()** provide information about a class' meta-constructors.

## Demo

### Introspection example

- All classes that contain signals or slots must mention `Q_OBJECT` at the top of their (private) declaration.
- They must also derive (directly or indirectly) from `QObject`.

# Custom signals

- Custom signals can be defined using the signals macro.

```
signals:  
    void sizeChanged();  
    void elementRemoved(std::string id);
```

- Signals can be emitted by an object when its internal state has changed in some way that might be interesting to another object.
- The `emit` macro is used to emit signals.

```
emit sizeChanged();
```

- Signals are public access functions and can be emitted from anywhere, but it is recommended to only emit them from the class that defines the signal and its subclasses.

- Custom slots are declared using the `slots` keyword.
- `slots` is actually an empty macro needed by the *moc* tool to generate meta-information about the available slots.
- Slots are normal C++ functions and can be called normally
- Their only special feature is that signals can be connected to them.

slots:

```
void onSizeChanged();  
void onElementRemoved(std::string id);
```



# Connecting signals to slots

- There are two alternative ways of connecting signals to slots:

// first version: using the SIGNAL and SLOT macro

```
QLabel *label = new QLabel;
```

```
QLineEdit *lineEdit = new QLineEdit;
```

```
QObject::connect(lineEdit, SIGNAL(textChanged(QString)),  
                 label,   SLOT(setText(QString)));
```

// newer version

```
QLabel *label = new QLabel;
```

```
QLineEdit *lineEdit = new QLineEdit;
```

```
QObject::connect(lineEdit, &QLineEdit::textChanged,  
                 label,   &QLabel::setText);
```

- The signal and slot mechanism can also be used from QtDesigner:  
<https://doc.qt.io/qt-5/designer-connection-mode.html>.

# Signals and slots disadvantages

- The main advantages of signal and slots is that they provide great flexibility and simplicity.
- However, compared to callbacks, signals and slots are slightly slower, although the difference for real applications is insignificant.
- Usually emitting a signal that is connected to some slots, is about 10 times slower than calling the receivers directly, with non-virtual function calls.
- The meta object system needs to locate the connection object, to safely iterate over all connections, and to marshall any parameters in a generic fashion.
- However, this overhead is much less than for other operations, like `new` or `delete`.





- The `QDebug` class provides an output stream for debugging information.
- `QDebug` is used whenever the developer needs to write out debugging or tracing information to a *device*, *file*, *string* or *console*.
- In the most common usage, you should call the `QDebug()` function to obtain a default `QDebug` object to use for writing debugging information.
- In Visual Studio, the messages printed with `QDebug()` will be displayed in the *Output* window.
- It is defined in header `<QDebug>`.

- The `QString` class provides a Unicode character string.
- It is used in Qt to store strings.
- The class has an constructor that accepts a `const char*`:  
`QString(const char *str)`, so functions that have a `QString` as parameter will accept also a `const char*`.
- To convert between a `QString` and a `std::string` you can use the `toString()` and `QString::fromStdString(std::string)` methods;
- The `QString` class also have methods to convert between a `QString` and a number and viceversa: `toInt()` and `QString::number(int)` methods;

- Other useful methods from `QString`:
  - `contains()` check whether a `QString` contains a particular character or substring;
  - `startsWith()` or `endsWith()`: check if a `QString` starts or ends with a particular substring.
  - `count()`: determines how many times a particular character or substring occurs in the string.
  - `QString` can be compared using overloaded operators such as `operator<()`, `operator<=()`, `operator==()`, `operator>=()`, and so on.  
*Note that the comparison is based exclusively on the numeric Unicode values of the characters.*

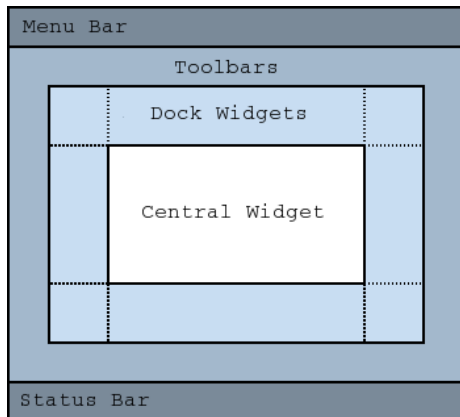
# QMessageBox

- A message box `QMessageBox` is a modal window that displays a text to inform user of a situation or to ask the user a question.
- It can display text, an item and standard buttons (*OK*, *Cancel*, *Open*, *Close*, *Save*, *Discard* etc.) for user response.
- `QMessageBox` supports four predefined message severity levels, or message types, which really only differ in the predefined icon they each show:

	Question	For asking a question during normal operations.
	Information	For reporting information about normal operations.
	Warning	For reporting non-critical errors.
	Critical	For reporting critical errors.

# QMainWindow I

- `QMainWindow` provides a main application window.
- `QMainWindow` has its own layout to which you can add:
  - a menu bar (at the top): `QMenuBar`
  - several tool bars: `QToolBar`
  - a status bar (at the bottom): `QStatusBar`
  - The layout has a center area that can be occupied by any kind of widget.



- QMainWindow provides the function `menuBar()`, which allows adding `QMenus` to the menu bar and adding `QActions` to the pop-up menus.
- `QAction` can be used for common commands can be invoked via menus, toolbar buttons, and keyboard shortcuts.



- `QToolBar` provides a movable panel that contains a set of controls.
- Toolbar buttons are added by adding actions, using the function `addAction`.

- The `QPainter` class performs low-level painting on widgets or other paint devices (classes that inherit from `QPaintDevice`).
- It can draw everything from simple lines to complex shapes like pies and chords. It can also draw aligned text and pixmaps.
- `QPainter` provides functions to draw most primitives: `drawPoint()`, `drawPoints()`, `drawLine()`, `drawRect()`, `drawRoundedRect()`, `drawEllipse()`, `drawArc()`, `drawPie()`, `drawChord()`, `drawPolyline()` etc.
- The `QPainterPath` is an object composed of building blocks such as rectangles, ellipses, lines.

- There are several settings that you can customize to make QPainter draw according to your preferences:
  - using a [QBrush](#) - you can define the color or pattern that is used for filling shapes.
  - using a [QPen](#) - you can define the color or stipple that is used for drawing lines or boundaries.
- The [paintEvent](#) method (of the [QWidget](#) class) is invoked when the [QWidget](#) needs to repaint all or part of the widget.
- !! When the paintdevice is a widget, [QPainter](#) can only be used inside a [paintEvent\(\)](#) function or in a function called by [paintEvent\(\)](#) !!

## Demo

### Painting example

# Summary

- Signals and slots are used for communication between objects.
- A signal is emitted when a particular event occurs.
- A slot is a function called in response to a particular signal.
- Signals and slots must be connected.