Object-Oriented Programming

Iuliana Bocicor
iuliana@cs.ubbcluj.ro

Babes-Bolyai University

2017

Overview

Callbacks I

- When we change one widget, we often want another widget to be notified. E.g.:
  - when the close button is pressed, the application shoud be closed;
  - when a value is chosen in a combobox, a list should be populated with different values.

- In other toolkits, this kind of communication is achieved using callbacks.

Callbacks II

- A callback is a pointer to a function. If you want a processing function to notify you about some event you pass a pointer to another function (the callback) to the processing function.

- The processing function then calls the callback when appropriate.

- E.g.
  - Comparison function passed to a sorting algorithm.
  - A progress bar object, which could be used by any client.

Callbacks III

Example - progress bar

- The prototype of the callback function is defined by the developer of the progress bar.

```
//callback function prototipe
typedef void(*ProgressListener)(int percent);

//function that notifies progress using callback
void someComputation(ProgressListener callback)
{
    for (int i = 0; i < 100; i++)
    {
        //do stuff
        callback(i); // notification after each
            step
    }
}
```

Callbacks IV

- The client code (which needs the notifications) will have to define a function with that specific prototype.

```
void onProgress(int percent)
{
    std::cout << "progress:" << percent;
}

int main()
{
    someComputation(onProgress);
}
```

Callbacks V

- Callbacks in c++ have two fundamental flaws:
    - 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 I

- Qt signals and slots are an alternative to the callback mechanism.
- They are used for communication between objects.
- The signals and slots mechanism is a central feature of Qt.

Signals and slots II

- A **signal** is emitted when a particular event occurs (e.g. a button is clicked).
- Qt widgets have many pre-defined signals which are emitted to indicate that a user action or a change of state has occurred.
- A **slot** is a function that is called in response to a particular signal.
- A signal can be connected to a function (called a slot), so that when the signal is emitted, the slot is automatically executed.

Signals and slots III

- The signature of a signal must match the signature of the receiving slot (in fact, a slot may have a shorter signature than the signal it receives because it can ignore extra arguments) $\Rightarrow$ **the signals and slots mechanism is type safe**.
- Slots can be used for receiving signals, but they are also normal member functions.
- Just as an object does not know if anything receives its signals, a slot does not know if it has any signals connected to it $\Rightarrow$ truly independent components can be created with Qt and there is a **loose coupling between signals and slots**.

Signals and slots IV

- You can connect as many signals as you want to a single slot, and a signal can be connected to as many slots as you need.
- Qt widgets have many predefined signals, but we can subclass widgets to add new signals to them.
- Qt widgets have many pre-defined slots.
- It is common practice to subclass widgets and add your user-defined slots that can handle the signals.

Connecting signals and slots I

- 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.

- Several slots can be connected to one signal, the slots will be executed one after the other, in the order they have been connected, when the signal is emitted.

Connecting signals and slots II

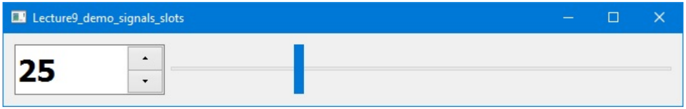
- To connect a signal and a slot we use the function QObject::connect and (optionally) the macros SIGNAL and SLOT.

```
QPushButton *closeButton = new QPushButton("&Close")
    ;
QObject::connect(this, &QPushButton::clicked,
    QApplication::instance(), QApplication::quit);
    // Qt5
QObject::connect(closeButton, SIGNAL(clicked()),
    QApplication::instance(), SLOT(quit())); //
    older Qt versions
```

DEMO

Simple example of connecting signals and slots (*Lecture9_demo_Qt_designer*).

Spin and slider example I



If the the user changes the value in the Spin Box:

- The spinbox will **emit the signal** valueChanged(int) with an int argument, the current value of the spinner.
- Because the spinner and slider are connected, the setValue(int) method of the slider will be invoked. The argument from the valueChanged method (the current value of the spinner) will be passed to the setValue method of the slider.

Spin and slider example II

- The slider updates itself to reflect the new value, and emits a valueChanged signal because its own value has changed.
- Because the slider is also connected with the spinner, the setValue slot of the spinner is invoked, in response to the slider's signal.
- The setValue of the spinner will not emit any signal because the current value is the same as the value provided in the setValue (prevents infinite loop).

DEMO

Simple example of connecting signals and slots (*Lecture10_demo_spinner_slider*).

Subclassing QWidget

- Create a separate class for the GUI that you are designing.
- This class must inherit from QWidget.
- Make it an independent, self-contained component, with its own signals and slots.

User defined signals and slots

- To define custom signals and slots, **use the Q_OBJECT macro**.
- This macro **must be at the beginning** of the class definition.

```
class GenesGUI: public QWidget
{
    Q_OBJECT
    //...
}
```

The meta-object system

- Qt introduces a mechanism called the **meta-object system** which provides two key services:
  - signals and slots;
  - introspection
    - the ability to introspect objects at runtime;
    - the the methods and properties of an object can be listed at runtime;
    - is necessary for implementing signals and slots;
    - allows application programmers to obtain "meta-information" about QObject subclasses at run-time, including the list of signals and slots supported by the object and its class name.

The meta-object compiler I

- The meta-object compiler (MOC) is the tool that provides introspection support.
- It is a code generator - parses the header files and generates an additional C++ file that is compiled with the rest of the program.
- The meta-object compiler takes all classes starting with the Q_OBJECT macro and generates a moc_*.cpp C++ source file.

The meta-object compiler II

- 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
  - slots
  - Q_OBJECT
  - emit
  - SIGNAL
  - SLOT

Custom slots I

- 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.
- A slot is just a regular method and the compiler will handle it as any other function.
- The only special feature of a slot is that signals can be connected to it.

Custom slots II

Declaration

```
public slots:
    // When an item in the list is clicked, the text
        boxes get filled with the item's
        information
    void listItemChanged();
    void populateGenesList();
    void addGene(const std::string& geneName, const
        std::string& organismName, const std::string
        & sequence);
```

- 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.

Custom signals I

- Custom signals can be defined using the signals macro.

```
signals:
    void genesUpdatedSignal();
    void addGeneSignal(const std::string&
        geneName, const std::string&
        organismName, const std::string&
        sequence);
```

- 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.

Custom signals II

```
void GenesGUI::deleteGene()
{
    // ...
    // emit the signal: the genes were updated
    emit genesUpdatedSignal();
}
```

- 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.
- When a signal is emitted, the slots connected to it are usually executed immediately, just like a normal function call.

Custom signals III

- Execution of the code following the emit statement will occur once all slots have returned.
- If several slots are connected to one signal, the slots will be executed one after the other, in the order they have been connected.
- Signals can never have return types (use void).

Example

DEMO

Gene manager (*Lecture10_demo_signals_and_slots*).

QMainWindow

- A main window provides a framework for building an application's user interface.

- QMainWindow has its own layout to which you can add:
  - A menu bar (on the top): QMenuBar;
  - Toolbars: QToolBar;
  - A central widget: this will typically be a QTextEdit or a QGraphicsView;
  - A status bar (on the bottom) : QStatusBar.

Using the menu bar

- 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.

```
QMenu* fileMenu = this->menuBar()->addMenu("&File");
QAction *openAction = new QAction("&Open", this);
fileMenu->addAction(openAction);
```

Using the tool bar

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

```
QToolBar* fileToolBar = addToolBar("&File");
fileToolBar->addAction(openAction);
fileToolBar->addAction(saveAction);
```

Notepad example

DEMO

Notepad example (*Lecture10_demo_Notepad*).

QPainter I

- QPainter is used for low-level painting on custom defined widgets.
- It can draw simple and complex shapes (from lines to pies).
- The QPen object, which defines how a painter should draw lines and outlines of shapes.
- The QBrush defines the fill pattern of shapes drawn by a QPainter.
- The QPainterPath is an object composed of building blocks such as rectangles, ellipses, lines.

QPainter II

- The paintEvent method (of the QWidget class) is invoked when the QWidget needs to repaint all or part of the widget.
- The keyPressEvent method is invoked when a key is pressed (**only if** the widget has the focus).
- The mousePressEvent method is invoked when a mouse key is pressed.

DEMO

Drawing example (*Lecture10_demo_Drawing*).

Useful Qt classes I

QString

- Unicode character string.
- The functions that have a QString as parameter will accept a const char*.
- There are methods to convert a QString in an std::string and viceversa.

```
QString s = "145";
std::string ss = s.toStdString();
QString s2 = QString::fromStdString(ss);
```

Useful Qt classes II

- There are methods to convert a QString in a number and viceversa.

```
QString s = "145";
int a = s.toInt();
double d = 3.8;
QString s3 = QString::number(d);
```

Useful Qt classes III

QVector

- This is a template class that provides a dynamic array.
- It stores its items in adjacent memory locations and provides fast index-based access.
- It offers operations similar to the STL vector.
- It can be converted to an STL vector and viceversa (function toStdVector, fromStdVector).

Useful Qt classes IV

QMessageBox

- Provides a modal dialog to show information.
- It can display text, an item and standard buttons (OK, Cancel, Open, Close, Save, Discard, etc.) for user response.
- There are 4 message types, which really only differ in the predefined icon they each show:
  - Question
  - Information
  - Warning
  - Critical

Useful Qt classes V

```
QMessageBox::information(this, "Info", "Selection
    changed");
// ...
int ret = QMessageBox::warning(this, "My Application
    ", "The document has been modified.Do you want
    to save your changes?", QMessageBox::Save |
    QMessageBox::Discard |QMessageBox::Cancel,
    QMessageBox::Save);
if (ret == QMessageBox::Save) {
 //do save
}
```

Useful Qt classes VI

QDebug

- Provides an output stream for debugging information.
- It is useful to call the qDebug() function to obtain a default QDebug object to use for writing debugging information.
- In VS, the message will be displayed in the "Output" window.

```
qDebug() << "Here is a message for debug.";
```