LABORATORY 12
Signals and slots in Qt

Objectives

The main objective of this laboratory is to present the signal and slots mechanism from the QT programming framework. Several code examples will be given to illustrate how actions can be linked to widgets using this mechanism.

Theoretical concepts

Signals and slots in Qt

When working with graphical user interfaces, an action/change on one element (widget) usually requires the notification of another element (widget). For example, when the minimize button of a window is pressed, the minimize method of the window should be called.
Many frameworks achieve this communication through callbacks (a pointer to another function). To receive notifications about an event, a widget or a processing function receives a callback and it is responsible for invoking the callback when required. While this mechanism is largely used, it can be counter-intuitive and may present difficulties in verifying that the arguments passed to the callback maintain the correct type.
The *Qt* framework introduces the signal and slot mechanism to enable communication between different components of a *Qt* application. It provides a flexible and efficient way to connect different parts of the application, allowing them to communicate and respond to events.
Signal: A signal is emitted when a particular event occurs. It acts as a notification that something has happened. For example, a button click, a value change, or a timer timeout can trigger signals.
Slot: A slot is a function that can be connected to a signal. When a signal is emitted, all connected slots are invoked. Slots are the functions that respond to signals. Slots can be class member functions or lambdas.
Connect: Connecting a signal to a slot means specifying that when a particular signal is emitted, a specific slot should be executed. This connection is established using the *connect* method defined in the *QObject* class. Using the *connect* function, signals can be connected to slots or even to other signals.
The basic syntax for the *connect* function is:

```
QObject::connect(sender, &SenderClass::signal, receiver,
&ReceiverClass::slot);
```

In the code snippet above:

- *sender* is the object emitting the signal;
- *&SenderClass::signal* is a pointer to the member function that represents the signal;
- *receiver* is the object containing the slot that should be called in response to the signal;
- *&ReceiverClass::slot* is a pointer to the member function that represents the slot, or it could be a lambda expression.

All classes that contain signals or slots use the Q_OBJECT macro at the top of their declaration and must be subclasses of *QObject*.

Example 1

The example below shows a simple Qt application that contains a single button, and each time the button is pressed the message "Button was clicked!" is displayed in the console.

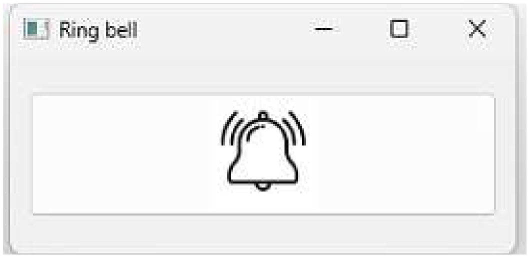
```
#include <QApplication>
#include <QPushButton>
#include <QDebug>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    QPushButton button("Click me!");

    // Connect the button's clicked signal to a lambda
function
    QObject::connect(&button, &QPushButton::clicked, [](){
        qDebug() << "Button was clicked!";
    });

    button.show();
    return app.exec();
}
```

A *QPushButton* named *button* is created with the text "Click me". The *QObject::connect* function call connects the sender's (the *button*) *click* function to the lambda function. In this case, the lambda `[](){qDebug() << "Button was clicked!";}` has no parameters and just displays the text "Button was clicked!" using the *qDebug()* function. The lambda function is invoked whenever the button is clicked.

Example 2

The example below shows a Qt application that creates a window with a single image button of type *QPushButton* that displays a bell image. The *QPushButton* class defines a signal called *clicked* and this signal is emitted each time the button is clicked. The slot connected to the button's *clicked* signal plays a bell sound. The application assumes that the image "bell_icon.jpg" and that the audio file "bell.wav" are stored in the same directory as the application's executable, and that the *Qt multimedia* module is already installed and enabled for the project.

```
#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QMediaPlayer>
#include <QVBoxLayout>
#include <QMediaContent>
#include <QPixmap>

class BellApp : public QWidget {
public:
    BellApp(QWidget *parent = nullptr) : QWidget(parent) {
        // Create a button with an icon
        QPushButton *bellButton = new QPushButton(this);
        QIcon bellIcon("bell_icon.jpg");
bellButton->setIcon(bellIcon);
        bellButton->setIconSize(QSize(128, 64));
        connect(bellButton, &QPushButton::clicked, this,
&BellApp::playBellSound);

        // Set up the layout
        QVBoxLayout *layout = new QVBoxLayout(this);
        layout->addWidget(bellButton);
        setLayout(layout);

        // Set up the media player
        mediaPlayer = new QMediaPlayer(this);
        QUrl bellSoundUrl = QUrl::fromLocalFile("bell.wav");
        mediaPlayer->setMedia(QMediaContent(bellSoundUrl));

        setWindowTitle("Ring bell");
    }

public slots:
    void playBellSound() {
        // Play the bell sound when the button is clicked
        mediaPlayer->play();
    }

private:
    QMediaPlayer *mediaPlayer;
};

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    BellApp bellApp;
    bellApp.setGeometry(100, 100, 300, 200);
    bellApp.show();

    return app.exec();
}
```



In the constructor of the *BellApp*, the *connect* method is used to connect the clicked signal of *bellButton* to the *playBellSound* slot within the *BellApp* class: `connect(bellButton, &QPushButton::clicked, this, &BellApp::playBellSound);`

- *bellButton* is the *QPush* button that will emit the signal when it is clicked;
- *&QPushButton::clicked* is a pointer to the signal from the button indicating it has been clicked;
- *this* refers to the current instance of *BellApp*, the receiver, specifying where the slot to be called resides;

- *&BellApp::playBellSound* is the slot, a member function of BellApp, designed to respond to the signal.

To play the media file "bell.wav" the code uses the *mediaPlayer* object of type *QMediaPlayer*. *QMediaPlayer* is part of the *Qt multimedia* module and is used for playing audio and video files.
The media file that *QMediaPlayer* will play is set using the *setMedia()* method in the constructor *BellApp*. This method takes a *QMediaContent* object, which is initialized here with a *QUrl* to specify the location of the media content. The *QUrl* is created from a local file named *bell.wav* located in the application's working directory The *playBellSound* slot is connected with the button's clicked signal to play the media file when the button is clicked. Inside this function, *mediaPlayer->play();* is called and therefore, the *QMediaPlayer* object will start playing the media content it has been configured with.

Example 3

The last example shows a simplified user interface for a book management application. The user interface consists of a simple form in which the user can input information about a book: *title*, *author*, *publication year*, *quantity* and *price*. The application also contains an *add* button which allows the user to add a new book based on the information inserted in the form. For the *year* field some basic user input validation is performed: the application checks whether the value inserted for the year is a positive integer. If not, a message box with a warning is displayed to the user.
The books are displayed in a tabular format (inside a *QTableWidget*).

mainwindow.h

```
#include <QMainWindow>
#include <QTableWidget>
#include <QLineEdit>
#include <QPushButton>
#include <QVBoxLayout>
#include <QFormLayout>
#include <QVector>
#include <QMessageBox>

class Book {

public:
    Book(QString t, QString a, int y, double p, int q):
title{t},
        author{a}, year{y}, price{p}, quantity{q}{};

    QString getTitle() const {return title;}
    QString getAuthor() const {return author;}
    int getYear() const {return year;}
    int getPrice() const {return price;}
    int getQuantity() const {return quantity;}

private:
    QString title;
    QString author;
    int year;
    double price;
    int quantity;


};

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void onAddButtonClicked();

private:
    QTableWidget *tableWidget;
    QLineEdit *titleEdit;
    QLineEdit *authorEdit;
    QLineEdit *yearEdit;
    QLineEdit *priceEdit;
    QLineEdit *quantityEdit;
    QPushButton *addButton;

    void setupUi();
    bool validateInput(); // Input validation function

    QVector<Book> books; // dynamic array of books

};
```

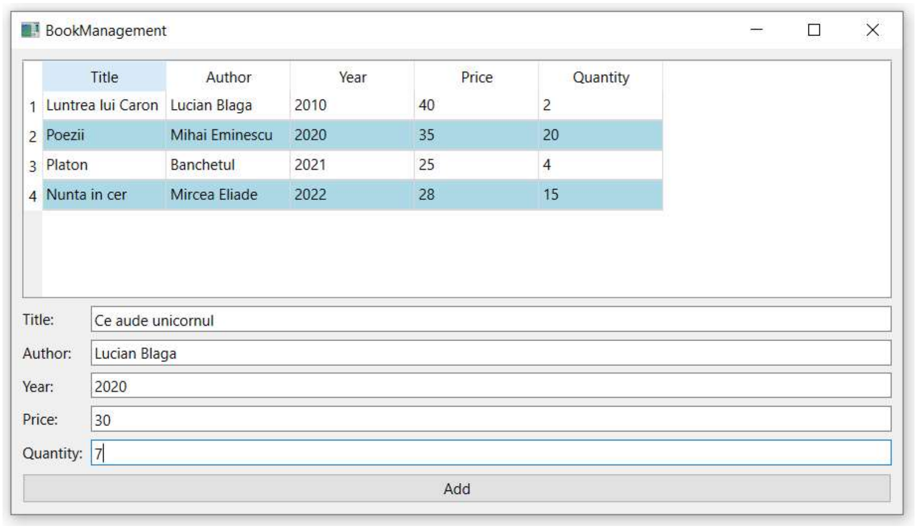
mainwindow.cpp

```
#include "mainwindow.h"
#include <QHeaderView>

MainWindow::MainWindow(QWidget *parent) :
QMainWindow(parent)
{
    setupUi();
}

MainWindow::~MainWindow()
{
}

void MainWindow::setupUi()
{

    QWidget *centralWidget = new QWidget(this);
    setCentralWidget(centralWidget);


    QVBoxLayout *mainLayout = new
QVBoxLayout(centralWidget);
    QStringList header{ "Title", "Author", "Year", "Price",
"Quantity"};
    // Table widget for displaying books
    tableWidget = new QTableWidget(0, 5, this); // 0 rows, 5
columns
    tableWidget->setHorizontalHeaderLabels(header);
    tableWidget->setAlternatingRowColors(true);
    tableWidget->setWordWrap(true);

tableWidget->verticalHeader()->setDefaultSectionSize(20);
    tableWidget->setStyleSheet("QTableWidget {
alternate-background-color: #ADD8E6; }");
    mainLayout->addWidget(tableWidget);

    // Form layout for book input
    QFormLayout *formLayout = new QFormLayout();
    titleEdit = new QLineEdit(this);
    authorEdit = new QLineEdit(this);
    yearEdit = new QLineEdit(this);
    priceEdit = new QLineEdit(this);
    quantityEdit = new QLineEdit(this);
    formLayout->addRow("Title:", titleEdit);
    formLayout->addRow("Author:", authorEdit);
    formLayout->addRow("Year:", yearEdit);
    formLayout->addRow("Price:", priceEdit);
    formLayout->addRow("Quantity:", quantityEdit);

    mainLayout->addLayout(formLayout);

    // Add button
    addButton = new QPushButton("Add", this);
    connect(addButton, &QPushButton::clicked, this,
&MainWindow::onAddButtonClicked);
    mainLayout->addWidget(addButton);
}

void MainWindow::onAddButtonClicked() {
    if (!validateInput()) {
        QMessageBox::warning(this, "Input Validation",
"Please enter a valid year.");
        return;
    }

    Book book{titleEdit->text(), authorEdit->text(),
yearEdit->text().toInt(),  priceEdit->text().toDouble(),
quantityEdit->text().toInt()};
    books.append(book); // Add the book to the array

    int currentRow = tableWidget->rowCount();
    tableWidget->insertRow(currentRow);

    tableWidget->setItem(currentRow, 0, new
QTableWidgetItem(book.getTitle()));
    tableWidget->setItem(currentRow, 1, new
QTableWidgetItem(book.getAuthor()));
    tableWidget->setItem(currentRow, 2, new
QTableWidgetItem(QString::number(book.getYear())));
    tableWidget->setItem(currentRow, 3, new
QTableWidgetItem(QString::number(book.getPrice())));
    tableWidget->setItem(currentRow, 4, new
QTableWidgetItem(QString::number(book.getQuantity())));

    // Optionally clear the input fields after adding
    titleEdit->clear();
    authorEdit->clear();
    yearEdit->clear();
    priceEdit->clear();
    quantityEdit->clear();

    tableWidget->resizeRowsToContents();

}

bool MainWindow::validateInput() {
    bool ok;
    int year = yearEdit->text().toInt(&ok);
    // Basic validation: ensure year is a positive integer
    return ok && year > 0;
}
```

main.cpp

```
#include "mainwindow.h"

#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

The result is the following:



	Title	Author	Year	Price	Quantity
1	Luntrea lui Caron	Lucian Blaga	2010	40	2
2	Poezii	Mihai Eminescu	2020	35	20
3	Platon	Banchetul	2021	25	4
4	Nunta in cer	Mircea Eliade	2022	28	15

Title:	Ce aude unicornul
Author:	Lucian Blaga
Year:	2020
Price:	30
Quantity:	7

The *MainWindow* class stores a vector of *Books* in the attribute *QVector<Book> books*. The method *setupUi* function programmatically builds user interface elements of the application (widget initialization and layout configuration). A *QTableWidget* is used to display book information; the table widget is configured with several properties: alternating row colors, word wrap, and custom row sizes. The function creates a form layout for inputting book details (title, author, year, price, and quantity) using *QLabel*s for the label and *QLineEdit*s for data entry. The *QPushButton is* used to add books based on the information the user inputted in the form. All these user interface elements (table widget, form layout and button) are arranged in a vertical box layout (*QVBoxLayout*).
The statement: *connect(addButton, &QPushButton::clicked, this, &MainWindow::onAddButtonClicked);* establishes the connection between the clicked signal of the add button and the *onAddButtonClicked* slot defined in the *MainWindow* class. When the button is clicked by the user, this signal is emitted, triggering the execution of the *onAddButtonClicked* method.
The *onAddButtonClicked* method first checks if the value entered by the user for the year is a positive integer using the *validateInput* function; if the validation fails, it shows a warning message. Otherwise, if validation passes, it constructs a *Book* object from the inputs, appends it to the *books QVector*, and updates the *QTableWidget* by adding a new row with the book's details. Finally, it clears the input fields for the next entry and resizes the table rows to fit the contents, ensuring that all text is visible and well-presented.

Proposed problems

1. Continue working on the playlist application from Laboratory 11. In the previous laboratory, just the user interface was created. Now, the actual functionality will be integrated into the application.
   - Create a class *Song* with the following attributes: title, artist, duration and mediaPath;
   - Create a class *SongController* that has at least two attributes: a *QVector* container to store all the songs and a *QVector* container to store the songs from the playlist;
   - The *MainWindow* class will store a pointer to the *SongController* object. The application logic (adding or removing songs) will be implemented in the *SongController* class;
   - Use signals and slots to perform the following operations:
     - When the user presses the *Add* button, a new song will be added in the container that stores all the songs; the information about this song is extracted from the form;
     - When the user presses the *Delete* button, the currently selected song will be removed front from the list from the all songs container and (if it is the case) from the playlist;
     - When the user presses the > button, the currently selected song from the list of all songs will be added at the end of the playlist.
2. [**Optional**] Use signals and slots such that when the user pressed the *Play* button the currently selected song from the playlist will be played using a *QMediaPlayer*.
3. [**Optional**] Use signals and slots so that when the user pressed the *Next* button the currently playing song (if any) will be stopped, and the next song from the playlist will be played. If the end of the playlist is reached, the first song will be played.