LABORATORY 11
Introduction to the Qt programming framework

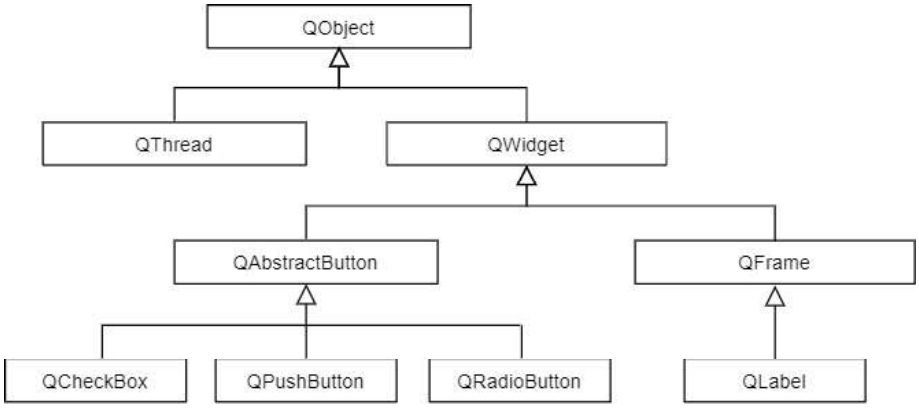
Objectives

The main object of this laboratory is to introduce the *Qt* programming framework. The laboratory will present the organization of the *Qt* framework, its main features and will present several examples on how to create basic graphical user interfaces.

Theoretical concepts

Qt framework

Qt (pronounced "cute") is a framework for creating graphical user interfaces and cross-platform applications that can be run on multiple software and hardware platforms (UNIX, Windows, macOS, Android or embedded systems) with little or no change to the code. The Qt framework organizes its classes in a modular fashion, providing tools for various purposes such as data containers and event management (*QtCore*), creating graphical user interfaces (*QtGui* and *QtWidgets* modules), managing network operations (*QtNetwork*), handling databases(*QtSQL*), and manipulating 3D objects (*Qt3D*), just to name a few.
Qt is based on the event-driven programming paradigm through its signal and slot mechanism. This paradigm simplifies event handling and communication between objects, enhancing the flexibility and maintainability of code.
The *QObject* class plays a central role as it is the base class for the classes defined in the Qt framework.



Class diagram depicting the main classes from the Qt framework

Qt introduces a powerful parenting system that manages the lifetime and organization of objects within a *Qt* application. Each *QObject* can have a parent, and when a *QObject* is assigned a parent, a hierarchical relationship is established. The parent-child relationship is established through the constructor of the child object, in which a pointer to the parent object is passed as an argument. *QObject* also provides the *setParent()* function for dynamic re-parenting.

The benefits of handling a hierarchy tree of objects are:

- memory management is simplified (providing automatic resource cleanup and preventing memory leaks), as child objects are automatically deleted when their parent is deleted;
- the *QObject* class provides the *findChild* and *findChildren* methods that can be used to access children of a given object;
- child widgets of a *QWidget* are automatically placed inside the parent widget.

Widgets in *Qt* are objects that represent user interface elements, such as buttons, text boxes or labels. The *QWidget* class is the base class for all graphical user interface objects in *Qt* (as illustrated in the figure above).

QApplication

QApplication is the that manages the graphical user interface control flow and settings. A *Qt* application must contain one and only one *QApplication* object, regardless of the number of windows the application has. This class manages the main event loop, where all events from the window system and other sources are processed and dispatched.

A *QApplication* object must be created in the *main()* function for any widget-based *Qt* application. Also, at the end of the *main()* function the *exec()* method of the *QApplication* object must be called to start the event loop. This loop keeps the application running, waiting for user interactions or system events, and dispatching them to the appropriate widgets. The *QApplication* class can also parse the applications common command-line arguments.

QString class

The *QString* class is used in *Qt* for managing strings and it implements several operations such as concatenation, slicing, and case changing. To convert a *QString* object to/from a *std::string* object the *toStdString* and *fromStdString* methods of the *QString* class must be used. The *toStdString()* member function is used for converting a *QString* into a *std::string*. On the other hand, converting a *std::string* into a *QString* is achieved with the *fromStdString()* static function of the *QString* class.
The example below shows how to convert a *std::string* to a *QString* and vice-versa. Also, the string stored in the *qtString* object is converted to uppercase by calling the function *toUpper* from the *QString* class.

```
#include <iostream>
#include <string>
#include <QString>
#include <QApplication>

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

     std::string stdString = "Hello, cute Qt!";

    // Use the static method fromStdString to convert
std::string to QString
    QString qtString = QString::fromStdString(stdString);

    // Convert qtString to all uppercase
    QString qtStringUpper = qtString.toUpper();

    // Use toStdString to convert the QString to std::string
    std::cout << "Converted QString to uppercase: " <<
qtStringUpper.toStdString() << std::endl;

    return 0; // No GUI shown, so exit immediately
```

```
}
```

```
Converted QString to uppercase: HELLO, CUTE QT!
```

To display messages to the standard output in *Qt*, the *qDebug()* function – declared in the <*QDebug*> header – should be used. The syntax of *qDebug()* is similar to that of std::cout in C++ (it also uses the stream operator (<<) to concatenate multiple items). The code snippet below displays: "*This is an integer: 10 ; this is a character: a*".

```
qDebug()<<"This is an integer: "<<10<<"; this is a character: "<<'a';
```

An advantage of *qDebug*() is its ability to handle various Qt types.

Basic graphical user elements

As mentioned before, widgets are the basic building blocks for creating graphical user interfaces.

QLabel is a widget suitable for displaying static plain text or simple images. *QLabel*'s buddy system allows it to be associated with another widget and to help the user quickly navigate to the other widget. When a *QLabel* is set as a buddy to another widget, activating the *QLabel*'s mnemonic key (indicated by an "&" sign before a character in the label's text) moves the focus to the associated widget.
QLineEdit represents a single-line text input field and it is used for user input of short, one-line text. This widget provides several editing functions such as undo and redo, cut and paste, and drag and drop. In addition, it supports different echo modes for displaying its contents, useful for sensitive inputs like passwords (the text can be displayed as asterisks or dots instead of the actual characters).
QPushButton represents a simple clickable button, capable of displaying both images and text.

The example below shows a simple login form that asks the user for the username and the password (the text of the password will be displayed as asterisks). The login form has a single button "Login".

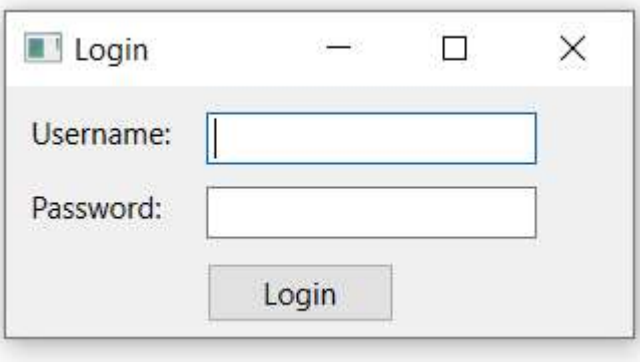
In the beginning of the main function, the *QApplication* object is initialized with *argc* and *argv* (the command-line arguments).
The following widgets are used to allow the user to specify the username:

- A *QLabel* called *usernameLabel* is created with the text "&*Username*:". The ampersand (&) indicates that the next character (U) is a mnemonic key, allowing keyboard shortcuts to focus on the associated input field when the user presses ALT+U. The *move* method is defined in the *QWidget* class and it moves the position of the widget (top left corner) to the specified location;
- A *QLineEdit* called *usernameEdit* is created for the user to enter the username. It is positioned to the right of the username label. The usernameLabel is associated with *usernameEdit* using the *setBuddy()*: `usernameLabel->setBuddy(usernameEdit);` function enabling keyboard navigation when the user presses the ALT+U key.

Similarly, a *QLabel* for the password (*passwordLabel*) and a *QLineEdit* for password input (*passwordEdit*) are created. The password label also uses an ampersand for mnemonic key setup. The echo mode of *passwordEdit* is set using the *setEchoMode*() function to *QLineEdit::Password*, ensuring that the entered password is obscured (dots are displayed instead of the text).
Then, a *QPushButton* called *sendButton* is created with the label "Login" and positioned below the password fields.
Finally, the *app.exec()* statement starts the application's event loop, allowing for user interaction.

```
#include <QApplication>
#include <QLabel>
#include <QLineEdit>
#include <QPushButton>

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

     // Main widget
     QWidget window;
     window.setWindowTitle("Login");
     window.resize(250, 100);

     // Username label and text edit
     QLabel *usernameLabel = new QLabel(&window);
     usernameLabel->setText("&Username:");
     usernameLabel->move(10, 10);
```

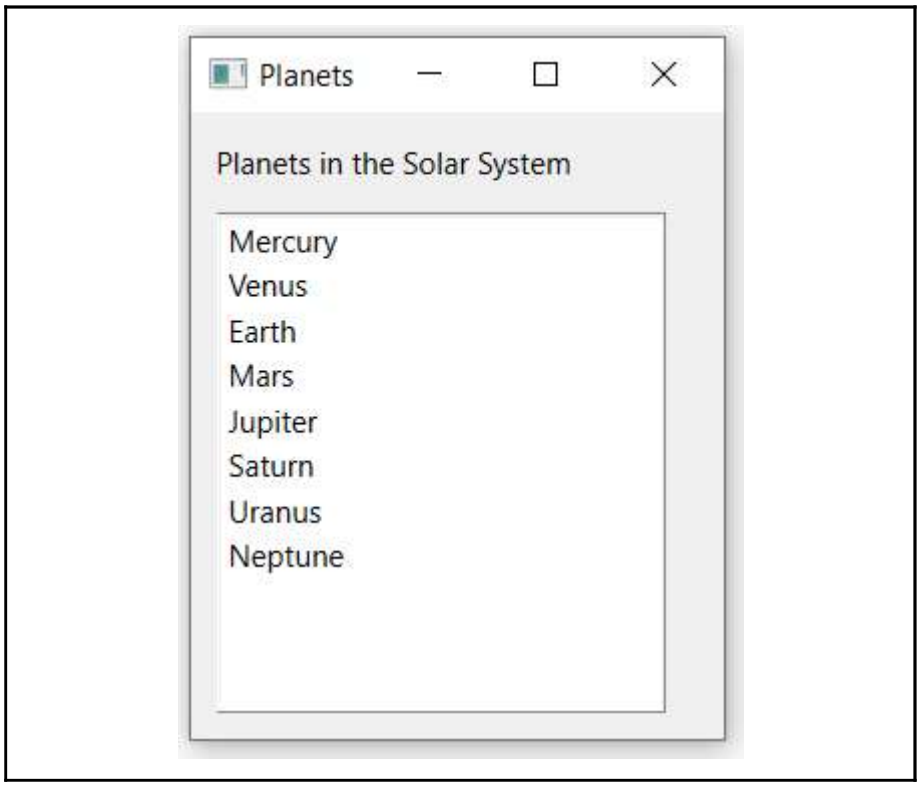
```
    QLineEdit *usernameEdit = new QLineEdit(&window);
    usernameEdit->move(80, 10);

    usernameLabel->setBuddy(usernameEdit);

    // Password label and text edit
    QLabel *passwordLabel = new QLabel(&window);
    passwordLabel->setText("&Password:");
    passwordLabel->move(10, 40);

    QLineEdit *passwordEdit = new QLineEdit(&window);
    passwordEdit->move(80, 40);
    passwordEdit->setEchoMode(QLineEdit::Password); // Set
echo mode for password

    passwordLabel->setBuddy(passwordEdit);

    // Send button
    QPushButton *sendButton = new QPushButton(&window);
    sendButton->setText("Login");
    sendButton->move(80, 70);

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



QListWidget is used to display a simple item-based list view. To add an item in the list view, the *void addItem(QString)* method should be used. The example below shows a simple list view that displays the planets in the solar system.

```
#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QListWidget>

int main(int argc, char *argv[]){
   QApplication app(argc, argv);
    QWidget window;
    window.setWindowTitle("Planets ");
    window.resize(200, 250);

    QLabel *label = new QLabel("Planets in the Solar
System", &window);
    label->move(10, 10);
    label->resize(180, 20);

    QListWidget *listWidget = new QListWidget(&window);
    listWidget->move(10, 40);
    listWidget->resize(180, 200);


    listWidget->addItem("Mercury");
    listWidget->addItem("Venus");
    listWidget->addItem("Earth");
    listWidget->addItem("Mars");
    listWidget->addItem("Jupiter");
    listWidget->addItem("Saturn");
    listWidget->addItem("Uranus");
    listWidget->addItem("Neptune");

    window.show();

    return app.exec();
}
```

Result



QTableWidget allows for the display and management of a 2D table. The class provides functions to create, access, and manage cells directly. The data stored by each cell is represented by an object of type *QTableWidgetItem*.

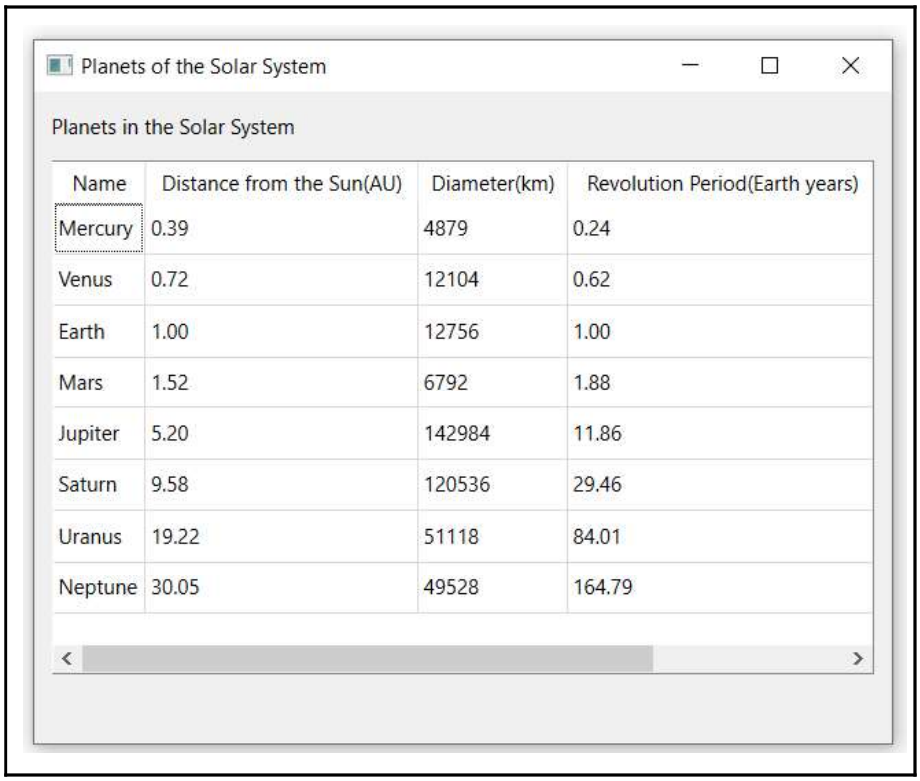
The example below shows a *QTableWidget* which displays information about the planets in the solar system. The columns of the table are the name of the planet, its distance from the sun, its diameter and its revolution period.
In the beginning, a *QTableWidget* is created as a child of a *window* and it is configured to have 8 rows and 4 columns to display information about the planets.
The *QStringList* class is a Qt container that stores a list of *QString* objects. The data for the planets of the Solar System is stored in four *QStringList* objects, each holding a specific type of information: planet names, their distances from the Sun (in astronomical units, AU), diameters (in kilometers), and revolution periods (in Earth years), with each list element corresponding to a planet in the order they're positioned from the Sun.
The for loop iterates through the lists of planet details and populates the *QTableWidget* with this data. The statement *tableWidget->setItem(i, 0, new*

QTableWidgetItem(planetNames.at(i))); creates a new table item (*QTableWidgetItem*) with the planet name from the list *planetNames* list at index *i*. This item is placed in row *i* and column 0 (the name column) of the table widget. Similarly, the other columns are populated with the distances, diameters and revolution periods.

```
#include <QApplication>
#include <QLabel>
#include <QTableWidget>
#include <QStringList>
#include <QHeaderView>
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget window;
    window.setWindowTitle("Planets of the Solar System");
    window.resize(500, 380);

    QLabel *label = new QLabel("Planets in the Solar
System", &window);
    label->move(10, 10);
    label->resize(480, 20);

    QTableWidget *tableWidget = new QTableWidget(&window);
    tableWidget->move(10, 40);
    tableWidget->resize(480, 300);
    tableWidget->setRowCount(8); // Number of planets
    tableWidget->setColumnCount(4); // Name, Distance,
Diameter, Revolution Period

    // Set the column headers
    QStringList headers = {"Name", "Distance from the Sun
(AU)", "Diameter (km)", "Revolution Period (Earth years)"};
    tableWidget->setHorizontalHeaderLabels(headers);

    // Add planet details
    QStringList planetNames = {"Mercury", "Venus", "Earth",
"Mars", "Jupiter", "Saturn", "Uranus", "Neptune"};
    QStringList distances = {"0.39", "0.72", "1.00", "1.52",
"5.20", "9.58", "19.22", "30.05"};
    QStringList diameters = {"4879", "12104", "12756",
"6792", "142984", "120536", "51118", "49528"};
    QStringList revolutionPeriods = {"0.24", "0.62", "1.00",
"1.88", "11.86", "29.46", "84.01", "164.79"};
```

```
    for(int i = 0; i < planetNames.size(); ++i) {
        tableWidget->setItem(i, 0, new
QTableWidgetItem(planetNames.at(i)));
        tableWidget->setItem(i, 1, new
QTableWidgetItem(distances.at(i)));
        tableWidget->setItem(i, 2, new
QTableWidgetItem(diameters.at(i)));
        tableWidget->setItem(i, 3, new
QTableWidgetItem(revolutionPeriods.at(i)));
    }

    // Make the table headers visible and adjust to fit
contents
    tableWidget->horizontalHeader()->setVisible(true);

tableWidget->horizontalHeader()->setSectionResizeMode(QHeade
rView::ResizeToContents);
    tableWidget->verticalHeader()->setVisible(false); //
Hide vertical header

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

Result



Layouts

An application usually requires the creation of several widgets. The straightforward way to arrange the widgets is to use absolute positioning. Absolute positioning involves specifying the exact coordinates (*x*, *y*) and size (*width*, *height*) of each widget relative to its parent widget. This can be achieved by setting the position and size of widgets using the *setGeometry()* method from the *QWidget* class or using the *move*() and *resize*() methods in combination.
While absolute positioning offers precise control over widget placement, it has several disadvantages:

- Lack of responsiveness: the widgets' positions and sizes are not automatically adjusted when the parent widget or window is resized;
- Maintenance difficulty: for complex graphical user interfaces managing the exact coordinates of each widget becomes cumbersome and error-prone;
- Localization issues: Different languages may require different amounts of space for text and absolute positioning makes it hard to accommodate these variations;

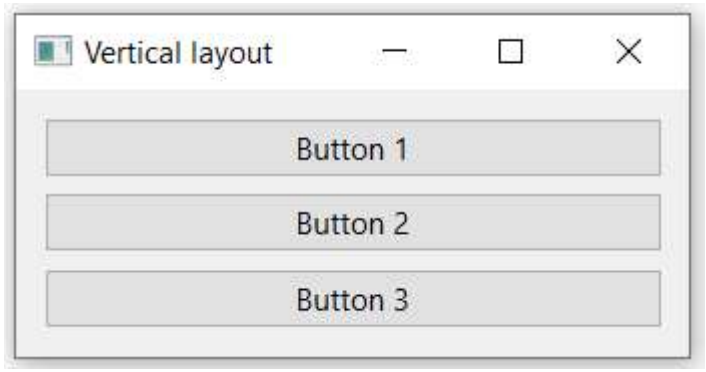
- Cross-platform inconsistencies: Absolute positioning does not account for differences in window decorations, font sizes, and other UI elements across platforms.

For these reasons, layouts managers are recommended, as they provide a more flexible, maintainable, and user-friendly approach to arranging widgets. Layouts are used to manage the placement and size of child widgets within a parent widget. They provide a way to create flexible and responsive user interfaces that can adapt to different window sizes and orientations. When placed inside a layout, child widgets are automatically positioned and resized based on certain rules, making it easier to create consistent and well-behaved user interfaces.
Qt provides several built-in layout classes, each serving different layout purposes.

QVBoxLayout (vertical layout) is used to stack widgets vertically, one on top of the other. In other words, this class can be used to create a column of widgets. To add a widget in a vertical layout the *void addWidget(QWidget* widget)* method of the *QVBoxLayout* is used.

```
#include <QApplication>
#include <QWidget>
#include <QVBoxLayout>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget *wnd = new QWidget{};
    QVBoxLayout *vLay = new QVBoxLayout{};
    QPushButton *btn1 = new QPushButton{ "Button 1" , wnd};
    QPushButton *btn2 = new QPushButton{ "Button 2" , wnd};
    QPushButton *btn3 = new QPushButton{ "Button 3" , wnd};
    vLay->addWidget(btn1); // the buttons will be aligned
vertically: i.e. btn1 is at the top of the widget, btn2 is
below btn1, btn3 is below btn2
    vLay->addWidget(btn2);
    vLay->addWidget(btn3);
    wnd->setLayout(vLay);
    wnd->show();
    wnd->setWindowTitle("Vertical layout");
    return a.exec();

    return a.exec();
}
```

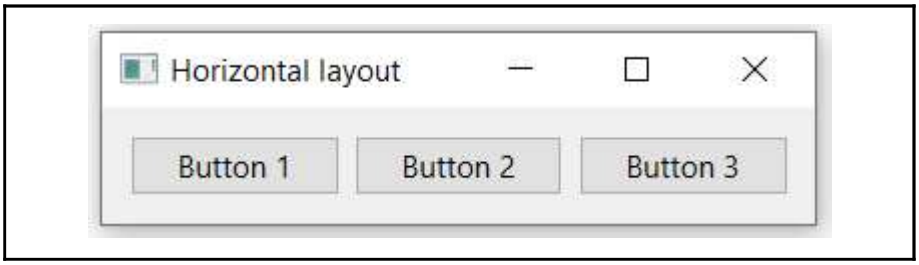
Result



QHBoxLayout (horizontal layout) is used to arrange widgets horizontally, side by side, i.e. to create a row of widgets. To add a widget in a horizontal layout the *void addWidget(QWidget* widget)* method of the *QHBoxLayout* is used.

```
#include <QApplication>
#include <QHBoxLayout>
#include <QPushButton>
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget wnd{};
    QHBoxLayout hLay{};
    QPushButton btn1{ "Button 1" };
    QPushButton btn2{ "Button 2" };
    QPushButton btn3{ "Button 3" };
    hLay.addWidget(&btn1); // the buttons will be aligned
horizontally, i.e. one next to the other: btn1 | btn2 | btn3
    hLay.addWidget(&btn2);
    hLay.addWidget(&btn3);
    wnd.setLayout(&hLay);
    wnd.setWindowTitle("Horizontal layout");
    wnd.show();

    return a.exec();
}
```

Result

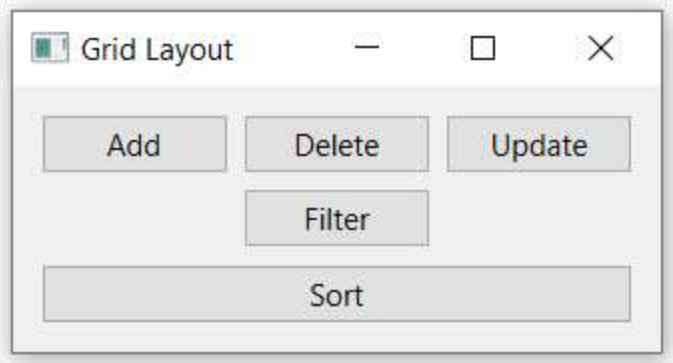


QGridLayout organizes widgets in rows and columns. The row and column of each widget are specified and the layout will automatically manage their positions. To add a widget to a grid layout the *void addWidget(QWidget* widget, int row, int column, int rowSpan = 1, int columnSpan = 1)* method should be used. The position of the widget inside the grid is specified in terms of row index (the *row* parameter) and column index (the *column* parameter). The *rowSpan* and *columnSpan* parameters are optional and allow the widget to span multiple rows or columns.

```
   QGridLayout *gridLayout = new QGridLayout{};
    QWidget *window = new QWidget{};
    // add some buttons
    // the add button is added at row 0, column 0
    gridLayout->addWidget(new QPushButton{"Add", window}, 0,
0);
    // the delete button is added at row 0, column 1
    gridLayout->addWidget(new QPushButton{"Delete", window},
0, 1);
    // the update button is added at row 0, column 2
    gridLayout->addWidget(new QPushButton{"Update", window},
0, 2);
    // the filter button is added at row 1, column 1
    gridLayout->addWidget(new QPushButton{"Filter", window},
1, 1);
    // it is also possible for a widget to occupy multiple
cells using the row and column spanning
    // row span = 1, column span = 3
    // the sort button is added at row 2, column 0, at it
spans 1 row and 3 columns
    gridLayout->addWidget(new QPushButton{"Sort", window},
2, 0, 1, 3);

    //  add     delete      update
    //          filter
    // -----------sort-------------
```

```
    window->setLayout(gridLayout);
    window->setWindowTitle("Grid Layout");
    window->show();
```

Result



QFormLayout is used to create form-based layouts which associate labels with input fields. It is often used for input forms. To add widgets to a form layout the *addRow()* method should be used. There are several overloads of this method for different use cases, such as adding a label and a field widget, or a single widget that spans both columns.

```
#include <QApplication>
#include <QFormLayout>
#include <QWidget>
#include <QLabel>
#include <QLineEdit>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget wnd{};
    QFormLayout *formLayout{new QFormLayout{&wnd}};

    // create labels and textboxes
    // name label and lineedit
    QLineEdit* nameLineEdit{new QLineEdit{&wnd}};
    QLabel* nameLabel{new QLabel{ "&Name:" }};
```

```
    nameLabel->setBuddy(nameLineEdit);

    // age label and lineedit
    QLineEdit* ageLineEdit{new QLineEdit{&wnd}};
    QLabel* ageLabel{new QLabel{ "&Age:" , &wnd}};
    ageLabel->setBuddy(ageLineEdit);

    // email label and lineedit
    QLineEdit* emailLineEdit{new QLineEdit{&wnd}};
    QLabel* emailLabel{new QLabel{ "&Email:" , &wnd}};
    emailLabel->setBuddy(emailLineEdit);

    // add the name, age and email rows to the form layout
    formLayout->addRow(nameLabel, nameLineEdit);
    formLayout->addRow(ageLabel, ageLineEdit);
    formLayout->addRow(emailLabel, emailLineEdit);



    // apply the form layout on the "main" widget
    wnd.setLayout(formLayout);
    wnd.setWindowTitle("Form layout");
    wnd.show();
    return app.exec();
}
```

Result:

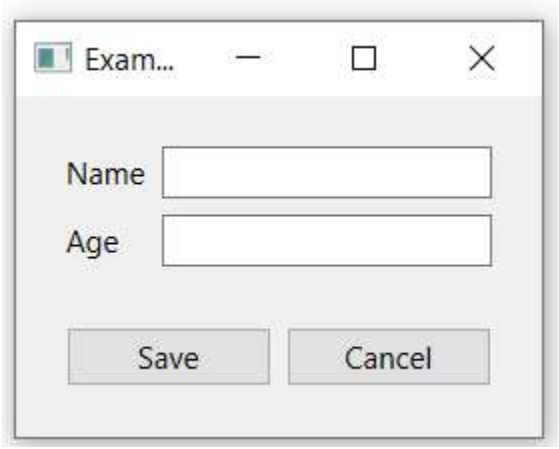
Of course, several layouts can be combined together. The example below shows how the three types of layouts are combined to create the resulting user interface. A form layout is used to pair labels and text edits fields horizontally (for the name and age respectively). To arrange the buttons *Save* and *Exit* a

horizontal layout is used. Finally, a vertical layout is used to stack the form layout and the horizontal layout on top of each other.

```
#include <QLabel>
#include <QLineEdit>
#include <QPushButton>
#include <QFormLayout>
#include <QVBoxLayout>
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QWidget window;
    QVBoxLayout *verticalLayout = new QVBoxLayout{&window};
    window.setLayout(verticalLayout); // all the elements
from the window will be aligned vertically
    window.setWindowTitle("Ex");

    // create an information widget, which uses a form
layout
    QWidget *detailsWidget = new QWidget{&window};
    QFormLayout *formLayout = new QFormLayout{};
    detailsWidget->setLayout(formLayout);

    // name row
    QLabel *labelName = new QLabel{ "&Name", &window};
    QLineEdit *nameTxt = new QLineEdit{&window};
    labelName->setBuddy(nameTxt);
    formLayout->addRow(labelName, nameTxt);

    // age row
    QLabel *labelAge = new QLabel{ "&Age" };
    QLineEdit *ageTxt = new QLineEdit{&window};
    labelAge->setBuddy(ageTxt);
    formLayout->addRow(labelAge, ageTxt);

    // add the information widget to the window
    verticalLayout->addWidget(detailsWidget);

    // create a widget with two buttons for two possible
actions: Save or Cancel
    // this widget will have a QHBoxLayout
    QWidget* actions = new QWidget{&window};
    QHBoxLayout* horizontalLayout = new QHBoxLayout{};
    QPushButton *cancelBtn = new QPushButton{ "&Cancel",
```

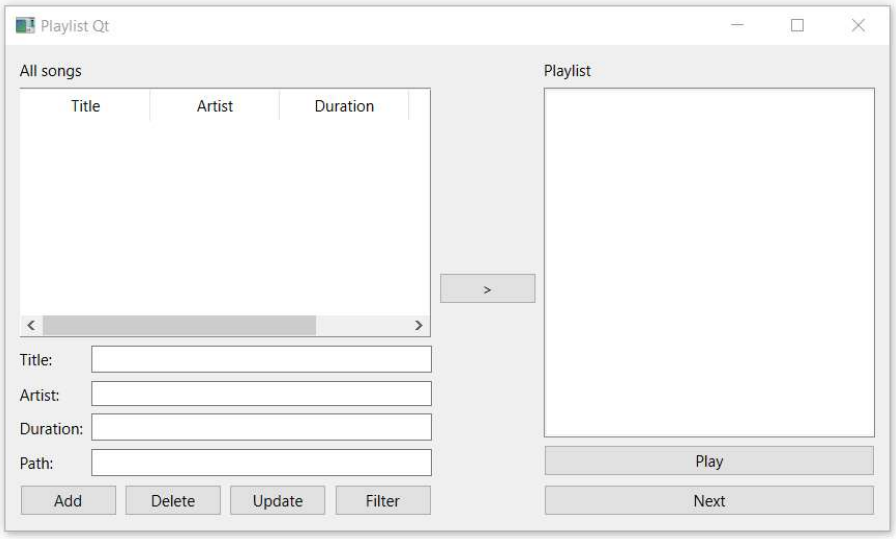
```
&window };
    QPushButton *saveBtn = new QPushButton{ "&Save", &window
};
    horizontalLayout->addWidget(saveBtn);
    horizontalLayout->addWidget(cancelBtn);
    actions->setLayout(horizontalLayout);

    // add the actions widget to the window
    verticalLayout->addWidget(actions);

    //show window
    window.show();

    return a.exec();
}
```

Result:



Proposed problems

1. Programmatically implement the following graphical user interface in Qt. First, identify the widgets that should be used for each element, then think about what layout types are appropriate to arrange the widgets.
   Create a class *MainWindow* which inherits from *QWindow* and create the use interface inside this class.


Playlist Qt
All songs
Title
Artist
Duration
Title:
Artist:
Duration:
Path:
Add
Delete
Update
Filter
>
Playlist
Play
Next