

# Слоты, Сигналы и События в Qt5

## Модель событий в программах Qt5

Механизм сигналов и слотов является расширением [языка программирования C++](#), который используется для установления связи между объектами. Если происходит какое-либо определённое **событие**, то при этом может генерироваться **сигнал**. Данный сигнал попадает в связанный с ним слот. В свою очередь, **слот** — это обычный [метод](#) в C++, который присоединяется к сигналу; он вызывается тогда, когда генерируется связанный с ним сигнал. Как видите, ничего сложного здесь нет.

Все [графические приложения](#) управляются событиями: всё, что происходит в приложении является результатом обработки тех или иных событий. Они являются важной частью любой графической программы. В большинстве случаев события генерируются пользователем приложения, но они также могут быть сгенерированы и другими средствами, например, подключением к интернету, оконным менеджером или таймером. При разработке программ в Qt5, задумываться о событиях приходится довольно редко, поскольку виджеты Qt5 генерируют *сигналы* в основном, когда происходит нечто значительное. Сами же события приобретают значение в том случае, когда необходимо создать, например, новый виджет или расширить функционал существующего.

В модели событий есть три участника:

- ➔ **источник события** — это объект, состояние которого изменяется;
- ➔ **объект события** — это отслеживаемый параметр источника события (например, нажатие клавиши на клавиатуре или изменение размеров виджета);
- ➔ **цель события** — это объект, который должен быть уведомлен о произошедшем событии.

Не нужно путать сигналы с событиями. Сигналы необходимы для организации взаимодействия между виджетами, тогда как события необходимы для организации взаимодействия между виджетом и системой.

## Щелчок мыши

В следующем примере мы рассмотрим простой способ обработки событий. У нас есть одна кнопка, по щелчку мышкой на которую мы завершаем работу приложения.

[Заголовочный файл](#) — click.h:

```
1 #pragma once
2
3 #include <QWidget>
4
5 class Click : public QWidget {
6
7     public:
8         Click(QWidget *parent = 0);
9 };
```

Файл реализации — click.cpp:

```
1 #include <QPushButton>
```

```

2 #include <QApplication>
3 #include <QHBoxLayout>
4 #include "click.h"
5
6 Click::Click(QWidget *parent)
7     : QWidget(parent) {
8
9     QHBoxLayout *hbox = new QHBoxLayout(this);
10    hbox->setSpacing(5);
11
12    QPushButton *quitBtn = new QPushButton("Quit", this);
13    hbox->addWidget(quitBtn, 0, Qt::AlignLeft | Qt::AlignTop);
14
15    connect(quitBtn, &QPushButton::clicked, qApp, &QApplication::quit);
16 }

```

Метод connect() соединяет сигнал со слотом. Когда мы нажимаем на кнопку Quit, генерируется сигнал щелчка кнопки мыши. qApp — это глобальный [указатель](#) на объект нашего приложения. Он определяется в заголовочном файле QApplication. Метод quit() вызывается при появлении сигнала щелчка мышкой:

```

1 connect(quitBtn, &QPushButton::clicked, qApp, &QApplication::quit);

```

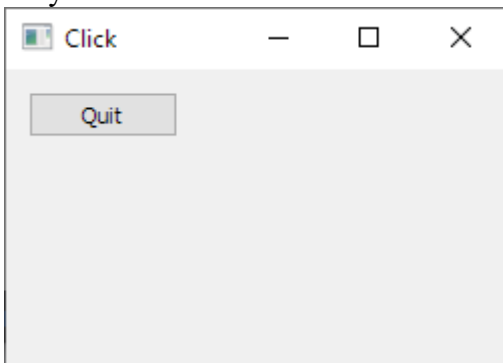
Главный файл программы — main.cpp:

```

1 #include <QApplication>
2 #include "click.h"
3
4 int main(int argc, char *argv[]) {
5
6     QApplication app(argc, argv);
7
8     Click window;
9
10    window.resize(250, 150);
11    window.setWindowTitle("Click");
12    window.show();
13
14    return app.exec();
15 }

```

Результат:



## Нажатие кнопки клавиатуры

В следующем примере мы рассмотрим способ реагирования на нажатие кнопки клавиатуры. Приложение завершит своё выполнение, если мы нажмём на клавишу Esc.

Заголовочный файл — keypress.h:

```
1 #pragma once
2
3 #include <QWidget>
4
5 class KeyPress : public QWidget {
6
7     public:
8         KeyPress(QWidget *parent = 0);
9
10    protected:
11        void keyPressEvent(QKeyEvent * e);
12};
```

Файл реализации — keypress.cpp:

```
1 #include <QApplication>
2 #include <QKeyEvent>
3 #include "keypress.h"
4
5 KeyPress::KeyPress(QWidget *parent)
6     : QWidget(parent)
7 { }
8
9 void KeyPress::keyPressEvent(QKeyEvent *event) {
10
11     if (event->key() == Qt::Key_Escape) {
12         qApp->quit();
13     }
14 }
```

Одним из способов работы с событиями в Qt5 является переопределение обработчика событий. `QKeyEvent` — это класс, который содержит информацию о произошедшем событии. В нашем случае, мы используем объект данного класса, чтобы определить, какая именно клавиша была нажата:

```
1 void KeyPress::keyPressEvent(QKeyEvent *e) {
2
3     if (e->key() == Qt::Key_Escape) {
4         qApp->quit();
5     }
6 }
```

Главный файл программы — main.cpp:

```
1 #include <QApplication>
2 #include "keypress.h"
3
4 int main(int argc, char *argv[]) {
5
6     QApplication app(argc, argv);
```

```

7
8     KeyPress window;
9
10    window.resize(250, 150);
11    window.setWindowTitle("Key press");
12    window.show();
13
14    return app.exec();
15 }

```

## Класс QMoveEvent

Класс **QMoveEvent** содержит параметры событий, возникающих при перемещении виджета. В следующем примере мы реагируем на событие перемещения, затем определяем текущие координаты *x* и *y* верхнего левого угла клиентской области окна и устанавливаем эти значения в заголовок окна.

Заголовочный файл — `move.h`:

```

1  #pragma once
2
3  #include <QMainWindow>
4
5  class Move : public QWidget {
6
7      Q_OBJECT
8
9      public:
10         Move(QWidget *parent = 0);
11
12     protected:
13         void moveEvent(QMoveEvent *e);
14 };

```

Файл реализации — `move.cpp`:

```

1  #include <QMoveEvent>
2  #include "move.h"
3
4  Move::Move(QWidget *parent)
5      : QWidget(parent)
6  { }
7
8  void Move::moveEvent(QMoveEvent *e) {
9
10     int x = e->pos().x();
11     int y = e->pos().y();
12
13     QString text = QString::number(x) + "," + QString::number(y);
14
15     setTitle(text);
16 }

```

Мы используем объект класса `QMoveEvent` для определения значений `x` и `y`:

```
1 int x = e->pos().x();
2 int y = e->pos().y();
```

Затем мы конвертируем целочисленные значения в [строки](#):

```
1 QString text = QString::number(x) + "," + QString::number(y);
```

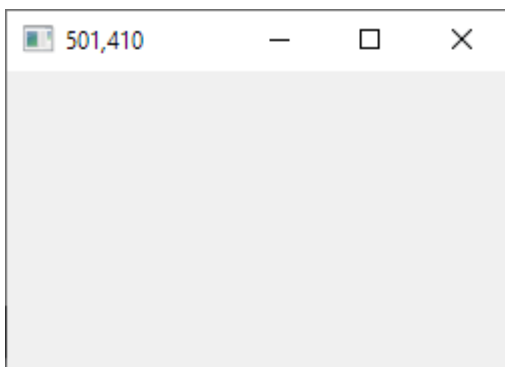
И с помощью метода `setWindowTitle()` устанавливаем текст в заголовок окна:

```
1 setWindowTitle(text);
```

Главный файл программы — `main.cpp`:

```
1 #include <QApplication>
2 #include "move.h"
3
4 int main(int argc, char *argv[]) {
5
6     QApplication app(argc, argv);
7
8     Move window;
9
10    window.resize(250, 150);
11    window.setWindowTitle("Move");
12    window.show();
13
14    return app.exec();
15 }
```

Результат:



## Отключение сигналов

Сигнал может быть отключен от слота. Следующий пример показывает, как мы можем это сделать.

В заголовочном файле мы объявили два слота. Следует отметить, что `slot` не является [ключевым словом](#) в C++, а лишь расширение Qt5. Подобные расширения обрабатываются препроцессором фреймворка до выполнения компиляции кода. Когда в наших классах мы используем сигналы и слоты, то обязательно должны предоставить макрос `Q_OBJECT` в начало определения класса. В противном случае препроцессор будет выдавать сообщения об ошибках.

В следующем примере у нас есть кнопка и флажок. Флажок подключает и отключает слот от сигнала нажатия кнопок.

Заголовочный файл — disconnect.h:

```
1  #pragma once
2
3  #include <QWidget>
4  #include <QPushButton>
5
6  class Disconnect : public QWidget {
7
8      Q_OBJECT
9
10     public:
11         Disconnect(QWidget *parent = 0);
12
13     private slots:
14         void onClick();
15         void onCheck(int);
16
17     private:
18         QPushButton *clickBtn;
19 };
```

Файл реализации — disconnect.cpp:

```
1  #include <QTextStream>
2  #include <QCheckBox>
3  #include <QHBoxLayout>
4  #include "disconnect.h"
5
6  Disconnect::Disconnect(QWidget *parent)
7      : QWidget(parent) {
8
9      QHBoxLayout *hbox = new QHBoxLayout(this);
10     hbox->setSpacing(5);
11
12     clickBtn = new QPushButton("Click", this);
13     hbox->addWidget(clickBtn, 0, Qt::AlignLeft | Qt::AlignTop);
14
15     QCheckBox *cb = new QCheckBox("Connect", this);
16     cb->setCheckState(Qt::Checked);
17     hbox->addWidget(cb, 0, Qt::AlignLeft | Qt::AlignTop);
18
19     connect(clickBtn, &QPushButton::clicked, this, &Disconnect::onClick);
20     connect(cb, &QCheckBox::stateChanged, this, &Disconnect::onCheck);
21 }
22
23 void Disconnect::onClick() {
24
25     QTextStream out(stdout);
26     out << "Button clicked" << endl;
```

```

27 }
28
29 void Disconnect::onCheck(int state) {
30
31     if (state == Qt::Checked) {
32         connect(clickBtn, &QPushButton::clicked, this, &Disconnect::onClick);
33     } else {
34         disconnect(clickBtn, &QPushButton::clicked, this,
35                     &Disconnect::onClick);
36     }
37 }

```

Подключаем сигналы к нашим пользовательским слотам:

```

1 connect(clickBtn, &QPushButton::clicked, this, &Disconnect::onClick);
2 connect(cb, &QCheckBox::stateChanged, this, &Disconnect::onCheck);

```

Если мы делаем щелчок мышкой, то в окно терминала будет отправляться текст `Button clicked`:

```

1 void Disconnect::onClick() {
2
3     QTextStream out(stdout);
4     out << "Button clicked" << endl;
5 }

```

Внутри слота `onCheck()` мы подключаем или отключаем слот `onClick()` от кнопки, в зависимости от полученного параметра состояния:

```

1 void Disconnect::onCheck(int state) {
2
3     if (state == Qt::Checked) {
4         connect(clickBtn, &QPushButton::clicked, this, &Disconnect::onClick);
5     } else {
6         disconnect(clickBtn, &QPushButton::clicked, this, &Disconnect::onClick);
7     }
8 }

```

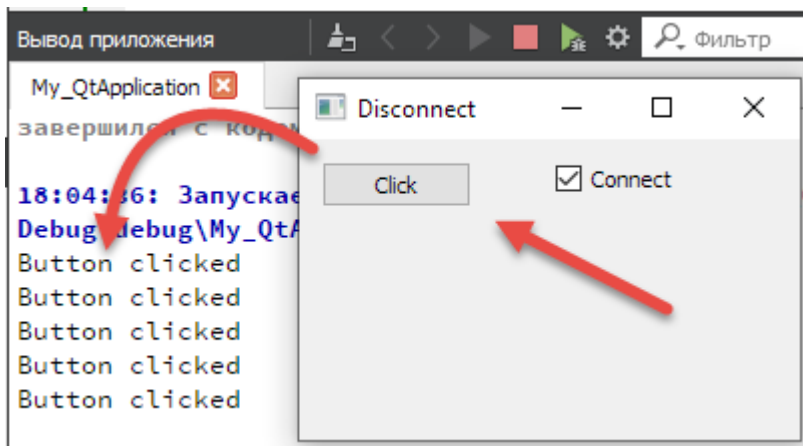
Главный файл программы — `main.cpp`:

```

1 #include <QApplication>
2 #include "disconnect.h"
3
4 int main(int argc, char *argv[]) {
5
6     QApplication app(argc, argv);
7
8     Disconnect window;
9
10    window.resize(250, 150);
11    window.setWindowTitle("Disconnect");
12    window.show();
13
14    return app.exec();
15 }

```

Результат:



## Таймер

Таймер используется для реализации одиночного действия или же повторяющихся задач. Хорошим примером, где мы можем задействовать таймер, являются часы; каждую секунду мы должны обновлять нашу метку, отображающую текущее время.

В следующем примере мы попробуем отобразить в окне текущее местное время.

Заголовочный файл — timer.h:

```
1  #pragma once
2
3  #include <QWidget>
4  #include <QLabel>
5
6  class Timer : public QWidget {
7
8  public:
9      Timer(QWidget *parent = 0);
10
11  protected:
12      void timerEvent(QTimerEvent *e);
13
14  private:
15      QLabel *label;
16  };
```

Файл реализации — timer.cpp:

```
1  #include "timer.h"
2  #include <QHBoxLayout>
3  #include <QTime>
4
5  Timer::Timer(QWidget *parent)
6      : QWidget(parent) {
7
```



```

8   QHBoxLayout *hbox = new QHBoxLayout(this);
9   hbox->setSpacing(5);
10
11   label = new QLabel("", this);
12   hbox->addWidget(label, 0, Qt::AlignLeft | Qt::AlignTop);
13
14   QTime qtime = QTime::currentTime();
15   QString stime = qtime.toString();
16   label->setText(stime);
17
18   startTimer(1000);
19 }
20
21 void Timer::timerEvent(QTimerEvent *e) {
22
23     Q_UNUSED(e);
24
25     QTime qtime = QTime::currentTime();
26     QString stime = qtime.toString();
27     label->setText(stime);
28 }

```

Для отображения времени мы используем виджет-метку:

```

1 label = new QLabel("", this);

```

Затем мы определяем текущее местное время и устанавливаем его в виджет-метку:

```

1 QTime qtime = QTime::currentTime();
2 QString stime = qtime.toString();
3 label->setText(stime);

```

Запускаем таймер (при этом каждые 1000 мс генерируется событие таймера):

```

1 startTimer(1000);

```

Для работы с событиями таймера необходимо переопределить метод timerEvent():

```

1 void Timer::timerEvent(QTimerEvent *e) {
2
3     Q_UNUSED(e);
4
5     QTime qtime = QTime::currentTime();
6     QString stime = qtime.toString();
7     label->setText(stime);
8 }

```

Главный файл программы — main.cpp:

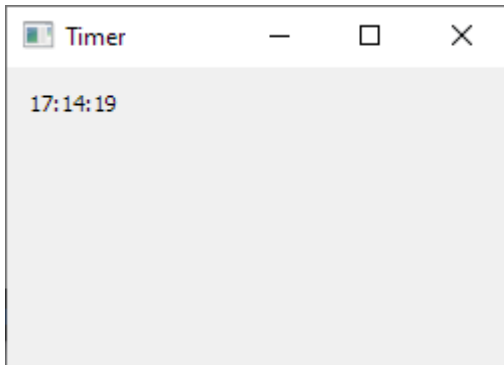
```

1 #include <QApplication>
2 #include "timer.h"
3
4 int main(int argc, char *argv[]) {
5

```

```
6   QApplication app(argc, argv);
7
8   Timer window;
9
10  window.resize(250, 150);
11  window.setWindowTitle("Timer");
12  window.show();
13
14  return app.exec();
15 }
```

Результат:



# Виджеты в Qt5

## Виджет QLabel

Виджет **QLabel** используется для отображения текста или изображения. При этом стоит заметить, что у него не предусмотрено взаимодействие с пользователем. В примере ниже мы задействуем данный виджет для отображения в окне текстов песен.

Заголовочный файл — label.h:

```
1  #pragma once
2
3  #include <QWidget>
4  #include <QLabel>
5
6  class Label : public QWidget {
7
8  public:
9      Label(QWidget *parent = 0);
10
11  private:
12      QLabel *label;
13  };
```

Файл реализации — label.cpp:

```
1  #include <QVBoxLayout>
2  #include <QFont>
3  #include "label.h"
4
5  Label::Label(QWidget *parent)
6      : QWidget(parent) {
7
8      QString lyrics = "Who doesn't long for someone to hold\n\
9  Who knows how to love you without being told\n\
10 Somebody tell me why I'm on my own\n\
11 If there's a soulmate for everyone\n\
12 \n\
13 Here we are again, circles never end\n\
14 How do I find the perfect fit\n\
15 There's enough for everyone\n\
16 But I'm still waiting in line\n\
17 \n\
18 Who doesn't long for someone to hold\n\
19 Who knows how to love you without being told\n\
20 Somebody tell me why I'm on my own\n\
21 If there's a soulmate for everyone";
22
23     label = new QLabel(lyrics, this);
24     label->setFont(QFont("Purisa", 10));
25 }
```

```

26 QVBoxLayout *vbox = new QVBoxLayout();
27 vbox->addWidget(label);
28 setLayout(vbox);
29 }

```

Создаём виджет метки и устанавливаем для него определённый шрифт:

```

1 label = new QLabel(lyrics, this);
2 label->setFont(QFont("Purisa", 10));

```

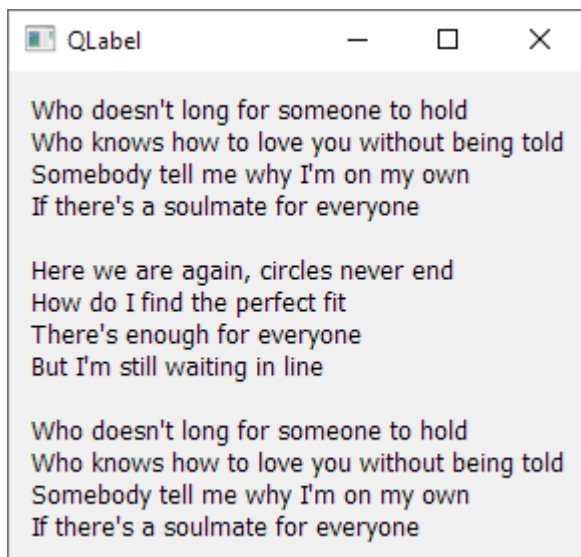
Главный файл программы — main.cpp:

```

1 #include <QApplication>
2 #include <QTextStream>
3 #include "label.h"
4
5 int main(int argc, char *argv[]) {
6
7     QApplication app(argc, argv);
8
9     Label window;
10
11     window.setWindowTitle("QLabel");
12     window.show();
13
14     return app.exec();
15 }

```

Результат:



## Виджет QSlider

**Виджет QSlider** представляет собой вертикальный или горизонтальный ползунок (слайдер). Он позволяет пользователю перемещать маркер ползунка вдоль горизонтальной или вертикальной линии и переводит положение ползунка в целочисленное значение в пределах допустимого диапазона.

В следующем примере у нас есть два виджета: слайдер и метка. Ползунок будет управлять числом, отображаемым в метке.

Заголовочный файл — slider.h:

```
1  #pragma once
2
3  #include <QWidget>
4  #include <QSlider>
5  #include <QLabel>
6
7  class Slider : public QWidget {
8
9      Q_OBJECT
10
11  public:
12      Slider(QWidget *parent = 0);
13
14  private:
15      QSlider *slider;
16      QLabel *label;
17  };
```

Файл реализации — slider.cpp:

```
1  #include <QHBoxLayout>
2  #include "slider.h"
3
4  Slider::Slider(QWidget *parent)
5      : QWidget(parent) {
6
7      QHBoxLayout *hbox = new QHBoxLayout(this);
8
9      slider = new QSlider(Qt::Horizontal, this);
10     hbox->addWidget(slider);
11
12     label = new QLabel("0", this);
13     hbox->addWidget(label);
14
15     connect(slider, &QSlider::valueChanged, label,
16             static_cast<void (QLabel::*)(int)>(&QLabel::setNum));
17 }
```

Создаётся горизонтальный QSlider:

```
1  slider = new QSlider(Qt::Horizontal, this);
```

Здесь мы подключаем [сигнал](#) valueChanged() к встроенному в метку слоту setNum(). Поскольку метод setNum() перегружен, мы используем [оператор static\\_cast](#) для выбора корректного метода:

```
1  connect(slider, &QSlider::valueChanged, label,
2  static_cast<void (QLabel::*)(int)>(&QLabel::setNum));
```

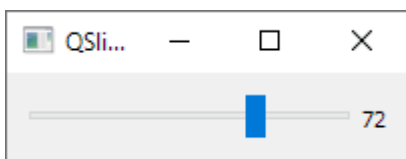
Основной файл программы — main.cpp:

```

1 #include <QApplication>
2 #include "slider.h"
3
4 int main(int argc, char *argv[]) {
5
6     QApplication app(argc, argv);
7
8     Slider window;
9
10    window.setWindowTitle("QSlider");
11    window.show();
12
13    return app.exec();
14 }

```

Результат:



## Виджет QComboBox

**QComboBox** — это виджет выбора, который отображает текущий элемент и может отображать выпадающий список выбираемых элементов. Список при этом может быть редактируемым, позволяя пользователю изменять каждый элемент в данном списке.

В следующем примере выбранный элемент из **QComboBox** будет отображаться в метке.

Заголовочный файл — `combobox.h`:

```

1 #pragma once
2
3 #include <QWidget>
4 #include <QComboBox>
5 #include <QLabel>
6
7 class ComboBoxEx : public QWidget {
8
9     Q_OBJECT
10
11 public:
12     ComboBoxEx(QWidget *parent = 0);
13
14 private:
15     QComboBox *combo;
16     QLabel *label;
17 };

```

Файл реализации — `combobox.cpp`:

```

1 #include <QHBoxLayout>

```

```

2  #include "combobox.h"
3
4  ComboBoxEx::ComboBoxEx(QWidget *parent)
5      : QWidget(parent) {
6
7      QStringList distros = {"Arch", "Xubuntu", "Redhat", "Debian",
8          "Mandriva"};
9
10     QHBoxLayout *hbox = new QHBoxLayout(this);
11
12     combo = new QComboBox();
13     combo->addItem(distros);
14
15     hbox->addWidget(combo);
16     hbox->addSpacing(15);
17
18     label = new QLabel("Arch", this);
19     hbox->addWidget(label);
20
21     connect(combo, static_cast<void(QComboBox::*)(const QString &)>(&QComboBox::activated),
22         label, &QLabel::setText);
23 }

```

В QStringList хранятся данные QComboBox, а именно — список дистрибутивов Linux:

```

1  QStringList distros = {"Arch", "Xubuntu", "Redhat", "Debian",
2      "Mandriva"};

```

Создается QComboBox, и затем с помощью метода addItem() в него добавляются элементы:

```

1  combo = new QComboBox();
2  combo->addItem(distros);

```

Сигнал activated() нашего QComboBox подключается к слоту setText() метки. Поскольку сигнал перегружен, мы делаем статическое преобразование данных при помощи static\_cast:

```

1  connect(combo, static_cast<void(QComboBox::*)(const QString &)>(&QComboBox::activated),
2      label, &QLabel::setText);

```

Главный файл программы — main.cpp:

```

1  #include <QApplication>
2  #include "combobox.h"
3
4  int main(int argc, char *argv[]) {
5
6      QApplication app(argc, argv);
7
8      ComboBoxEx window;
9
10     window.resize(300, 150);
11     window.setWindowTitle("QComboBox");
12     window.show();

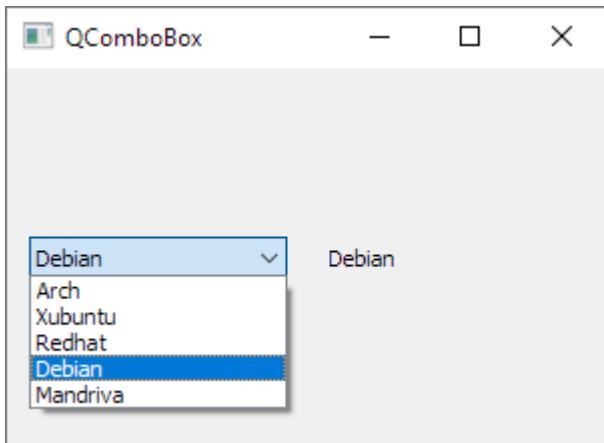
```

```

13
14     return app.exec();
15 }

```

Результат:



## Виджет QSpinBox

**QSpinBox** — это виджет, который используется для обработки целых чисел и дискретных наборов значений. Данный виджет позволяет пользователю указать нужное значение либо вручную, либо щёлкая мышью по кнопкам вверх/вниз или нажимая клавиши клавиатуры вверх/вниз для увеличения/уменьшения значения, отображаемого в текущий момент.

В программе ниже с помощью виджета **QSpinBox** мы можем выбирать число от 0 до 99. Выбранное в текущий момент значение отображается в метке.

Заголовочный файл — `spinbox.h`:

```

1  #pragma once
2
3  #include <QWidget>
4  #include <QSpinBox>
5
6  class SpinBox : public QWidget {
7
8      Q_OBJECT
9
10     public:
11         SpinBox(QWidget *parent = 0);
12
13     private:
14         QSpinBox *spinbox;
15 };

```

Файл реализации — `spinbox.cpp`:

```

1  #include <QHBoxLayout>
2  #include <QLabel>
3  #include "spinbox.h"
4

```



```

5 SpinBox::SpinBox(QWidget *parent)
6     : QWidget(parent) {
7
8     QHBoxLayout *hbox = new QHBoxLayout(this);
9     hbox->setSpacing(15);
10
11     spinbox = new QSpinBox(this);
12     QLabel *lbl = new QLabel("0", this);
13
14     hbox->addWidget(spinbox);
15     hbox->addWidget(lbl);
16
17     connect(spinbox, static_cast<void (QSpinBox::*)(int)>(&QSpinBox::valueChanged),
18            lbl, static_cast<void (QLabel::*)(int)>(&QLabel::setNum));
19 }

```

Нам нужно применить преобразование `static_cast` дважды, потому что и сигнал, и слот перегружены:

```

1 connect(spinbox, static_cast<void (QSpinBox::*)(int)>(&QSpinBox::valueChanged),
2     lbl, static_cast<void (QLabel::*)(int)>(&QLabel::setNum));

```

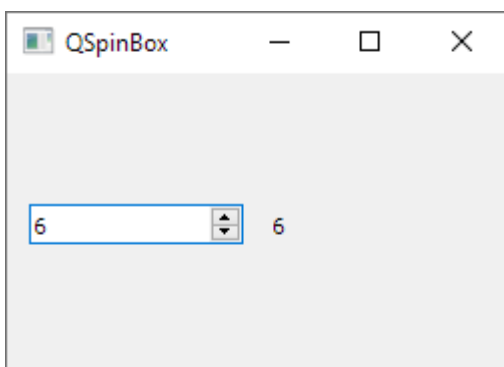
Главный файл программы — `main.cpp`:

```

1 #include <QApplication>
2 #include "spinbox.h"
3
4 int main(int argc, char *argv[]) {
5
6     QApplication app(argc, argv);
7
8     SpinBox window;
9
10    window.resize(250, 150);
11    window.setWindowTitle("QSpinBox");
12    window.show();
13
14    return app.exec();
15 }

```

Результат:



## Виджет QLineEdit

**Виджет QLineEdit** представляет собой редактор однострочного текста. Редактор строки позволяет пользователю вводить одну строку обычного текста и при этом использовать такие функции редактирования, как: отмена, повтор, вырезание, вставка, а также перетаскивание с помощью механизма drag-and-drop.

В следующем примере мы выведем три метки и три строки для редактирования, которые скомпонованы с помощью [менеджера компоновки QGridLayout](#).

Заголовочный файл — ledit.h:

```
1 #pragma once
2
3 #include <QWidget>
4
5 class Ledit : public QWidget {
6
7     public:
8         Ledit(QWidget *parent = 0);
9 };
```

Файл реализации — ledit.cpp:

```
1 #include <QGridLayout>
2 #include <QLabel>
3 #include <QLineEdit>
4 #include "ledit.h"
5
6 Ledit::Ledit(QWidget *parent)
7     : QWidget(parent) {
8
9     QLabel *name = new QLabel("Name:", this);
10    name->setAlignment(Qt::AlignRight | Qt::AlignVCenter);
11    QLabel *age = new QLabel("Age:", this);
12    age->setAlignment(Qt::AlignRight | Qt::AlignVCenter);
13    QLabel *occupation = new QLabel("Occupation:", this);
14    occupation->setAlignment(Qt::AlignRight | Qt::AlignVCenter);
15
16    QLineEdit *le1 = new QLineEdit(this);
17    QLineEdit *le2 = new QLineEdit(this);
18    QLineEdit *le3 = new QLineEdit(this);
19
20    QGridLayout *grid = new QGridLayout();
21
22    grid->addWidget(name, 0, 0);
23    grid->addWidget(le1, 0, 1);
24    grid->addWidget(age, 1, 0);
25    grid->addWidget(le2, 1, 1);
26    grid->addWidget(occupation, 2, 0);
27    grid->addWidget(le3, 2, 1);
28
29    setLayout(grid);
30 }
```

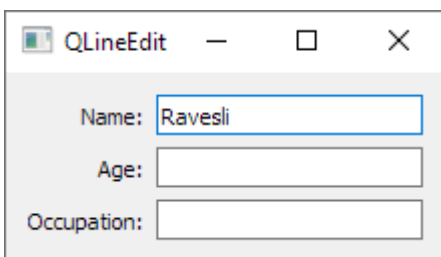
Главный файл программы — main.cpp:

```

1  #include "ledit.h"
2  #include <QApplication>
3
4  int main(int argc, char *argv[]) {
5
6      QApplication app(argc, argv);
7
8      Ledit window;
9
10     window.setWindowTitle("QLineEdit");
11     window.show();
12
13     return app.exec();
14 }

```

Результат:



## Строка состояния

**Строка состояния** (англ. «**statusbar**») — это панель, которая используется для отображения информации о состоянии приложения. Виджет **Statusbar** является частью виджета **QMainWindow**.

В следующем примере у нас есть две кнопки и одна строка состояния. При нажатии на кнопку будет отображаться соответствующее сообщение.

Заголовочный файл — `statusbar.h`:

```

1  #pragma once
2
3  #include <QMainWindow>
4  #include <QPushButton>
5
6  class Statusbar : public QMainWindow {
7
8      Q_OBJECT
9
10     public:
11         Statusbar(QWidget *parent = 0);
12
13     private slots:
14         void OnOkPressed();
15         void OnApplyPressed();
16
17     private:

```

```
18     QPushButton *okBtn;  
19     QPushButton *aplBtn;  
20 };
```

Файл реализации — statusbar.cpp:

```
1  #include <QLabel>  
2  #include <QFrame>  
3  #include <QStatusBar>  
4  #include <QHBoxLayout>  
5  #include "statusbar.h"  
6  
7  Statusbar::Statusbar(QWidget *parent)  
8      : QMainWindow(parent) {  
9  
10     QFrame *frame = new QFrame(this);  
11     setCentralWidget(frame);  
12  
13     QHBoxLayout *hbox = new QHBoxLayout(frame);  
14  
15     okBtn = new QPushButton("OK", frame);  
16     hbox->addWidget(okBtn, 0, Qt::AlignLeft | Qt::AlignTop);  
17  
18     aplBtn = new QPushButton("Apply", frame);  
19     hbox->addWidget(aplBtn, 1, Qt::AlignLeft | Qt::AlignTop);  
20  
21     statusBar();  
22  
23     connect(okBtn, &QPushButton::clicked, this, &Statusbar::OnOkPressed);  
24     connect(aplBtn, &QPushButton::clicked, this, &Statusbar::OnApplyPressed);  
25 }  
26  
27 void Statusbar::OnOkPressed() {  
28     statusBar()->showMessage("OK button pressed", 2000);  
29 }  
30  
31 void Statusbar::OnApplyPressed() {  
32     statusBar()->showMessage("Apply button pressed", 2000);  
33 }
```

Виджет `QFrame` помещается в центральную область виджета `QMainWindow`. Заметим, что центральную область может занимать только один виджет:

```
1  QFrame *frame = new QFrame(this);  
2  setCentralWidget(frame);
```

Мы создаём два виджета `QPushButton` и компоуем их вдоль горизонтальной линии. Родительским элементом кнопок является виджет `frame`:

```
1  okBtn = new QPushButton("OK", frame);  
2  hbox->addWidget(okBtn, 0, Qt::AlignLeft | Qt::AlignTop);  
3  
4  aplBtn = new QPushButton("Apply", frame);
```

```
5 hbox->addWidget(aPlBtn, 1, Qt::AlignLeft | Qt::AlignTop);
```

Для отображения строки состояния мы вызываем метод `statusBar()` виджета `QMainWindow`:

```
1 statusBar();
```

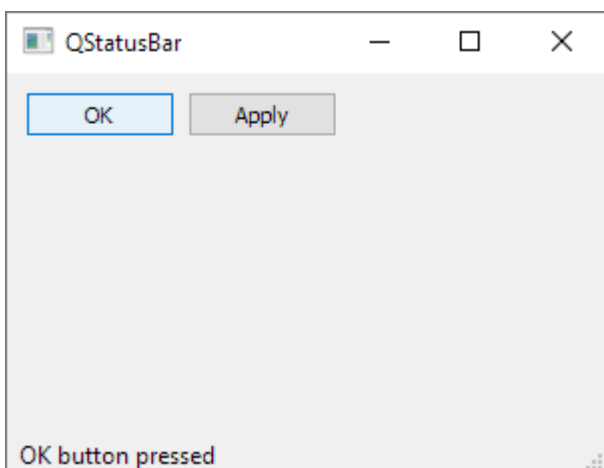
Метод `showMessage()` отображает сообщение в строке состояния. Последний параметр указывает количество миллисекунд, в течение которых сообщение отображается в строке состояния:

```
1 void StatusBar::OnOkPressed() {  
2     statusBar()->showMessage("OK button pressed", 2000);  
3 }
```

Главный файл программы — `main.cpp`:

```
1 #include <QApplication>  
2 #include "statusbar.h"  
3  
4 int main(int argc, char *argv[]) {  
5  
6     QApplication app(argc, argv);  
7  
8     StatusBar window;  
9  
10    window.resize(300, 200);  
11    window.setWindowTitle("QStatusBar");  
12    window.show();  
13  
14    return app.exec();  
15 }
```

Результат:



## Виджеты в Qt5 — Часть №2

### Виджет QCheckBox

**QCheckBox** — это виджет чекбокса (англ. «checkbox»), состоящий из ячейки и подписи к ней. QCheckBox имеет 2 состояния: включено или выключено. При включенном состоянии, внутри ячейки отображается флажок (галочка или крестик), при выключенном — ничего.

В следующем примере мы выведем в окне виджет чекбокса. Если у чекбокса установлен флажок, то будет выводиться заголовок окна, в противном случае заголовок окна будет скрыт.

Заголовочный файл — checkbox.h:

```
1  #pragma once
2
3  #include <QWidget>
4
5  class CheckBox : public QWidget {
6
7      Q_OBJECT
8
9      public:
10         CheckBox(QWidget *parent = 0);
11
12     private slots:
13         void showTitle(int);
14 };
```

Выводим чекбокс в окне и подключаем его к слоту showTitle().

Файл реализации — checkbox.cpp:

```
1  #include <QCheckBox>
2  #include <QHBoxLayout>
3  #include "checkbox.h"
4
5  CheckBox::CheckBox(QWidget *parent)
6      : QWidget(parent) {
7
8      QHBoxLayout *hbox = new QHBoxLayout(this);
9
10     QCheckBox *cb = new QCheckBox("Show Title", this);
11     cb->setCheckState(Qt::Checked);
12     hbox->addWidget(cb, 0, Qt::AlignLeft | Qt::AlignTop);
13
14     connect(cb, &QCheckBox::stateChanged, this, &CheckBox::showTitle);
15 }
16
17 void CheckBox::showTitle(int state) {
18
19     if (state == Qt::Checked) {
20         setWindowTitle("QCheckBox");
```

```

21     } else {
22         setWindowTitle(" ");
23     }
24 }

```

Флажок устанавливается при запуске примера:

```

1 cb->setCheckState(Qt::Checked);

```

Определяем состояние флажка и вызываем метод `setWindowTitle()`:

```

1 void CheckBox::showTitle(int state) {
2
3     if (state == Qt::Checked) {
4         setWindowTitle("QCheckBox");
5     } else {
6         setWindowTitle(" ");
7     }
8 }

```

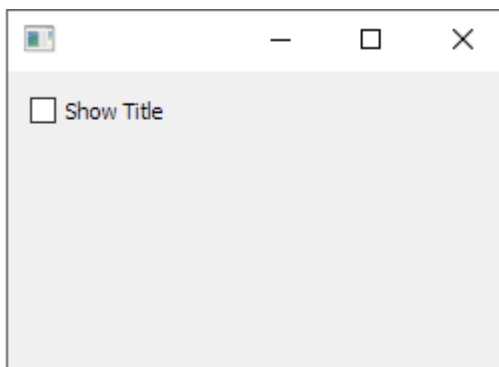
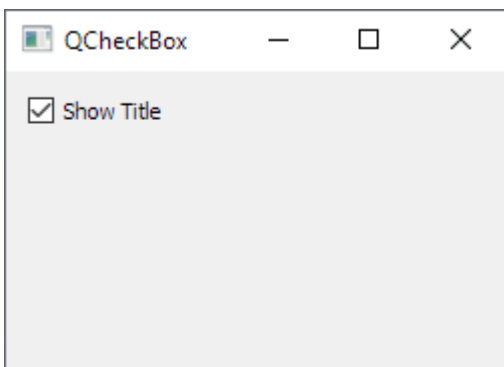
Главный файл программы — `main.cpp`:

```

1 #include <QApplication>
2 #include "checkbox.h"
3
4 int main(int argc, char *argv[]) {
5
6     QApplication app(argc, argv);
7
8     CheckBox window;
9
10    window.resize(250, 150);
11    window.setWindowTitle("QCheckBox");
12    window.show();
13
14    return app.exec();
15 }

```

Результат:



## Виджет QListWidget

**QListWidget** — это виджет, который используется для отображения списка элементов в Qt5. В нашем примере мы покажем, как добавлять, переименовывать и удалять элементы из данного виджета.

В примере ниже присутствуют виджет списка и четыре кнопки. Данные кнопки используются для добавления, переименования и удаления элементов из виджета списка.

Заголовочный файл — listwidget.h:

```
1  #pragma once
2
3  #include <QWidget>
4  #include <QPushButton>
5  #include <QListWidget>
6
7  class ListWidget : public QWidget {
8
9      Q_OBJECT
10
11     public:
12         ListWidget(QWidget *parent = 0);
13
14     private slots:
15         void addItem();
16         void renameItem();
17         void removeItem();
18         void clearItems();
19
20     private:
21         QListWidget *lw;
22         QPushButton *add;
23         QPushButton *rename;
24         QPushButton *remove;
25         QPushButton *removeAll;
26 };
```

Файл реализации — listwidget.cpp:

```
1  #include "listwidget.h"
2  #include <QVBoxLayout>
3  #include <QInputDialog>
4
5  ListWidget::ListWidget(QWidget *parent)
6      : QWidget(parent) {
7
8      QVBoxLayout *vbox = new QVBoxLayout();
9      vbox->setSpacing(10);
10
11      QHBoxLayout *hbox = new QHBoxLayout(this);
12
13      lw = new QListWidget(this);
14      lw->addItem("The Omen");
15      lw->addItem("The Exorcist");
16      lw->addItem("Notes on a scandal");
```



```

17 lw->addItem("Fargo");
18 lw->addItem("Capote");
19
20 add = new QPushButton("Add", this);
21 rename = new QPushButton("Rename", this);
22 remove = new QPushButton("Remove", this);
23 removeAll = new QPushButton("Remove All", this);
24
25 vbox->setSpacing(3);
26 vbox->addStretch(1);
27 vbox->addWidget(add);
28 vbox->addWidget(rename);
29 vbox->addWidget(remove);
30 vbox->addWidget(removeAll);
31 vbox->addStretch(1);
32
33 hbox->addWidget(lw);
34 hbox->addSpacing(15);
35 hbox->addLayout(vbox);
36
37 connect(add, &QPushButton::clicked, this, &ListWidget::addItem);
38 connect(rename, &QPushButton::clicked, this, &ListWidget::renameItem);
39 connect(remove, &QPushButton::clicked, this, &ListWidget::removeItem);
40 connect(removeAll, &QPushButton::clicked, this, &ListWidget::clearItems);
41
42 setLayout(hbox);
43 }
44
45 void ListWidget::addItem() {
46
47     QString c_text = QInputDialog::getText(this, "Item", "Enter new item");
48     QString s_text = c_text.simplified();
49
50     if (!s_text.isEmpty()) {
51
52         lw->addItem(s_text);
53         int r = lw->count() - 1;
54         lw->setCurrentRow(r);
55     }
56 }
57
58 void ListWidget::renameItem() {
59
60     QListWidgetItem *curitem = lw->currentItem();
61
62     int r = lw->row(curitem);
63     QString c_text = curitem->text();
64     QString r_text = QInputDialog::getText(this, "Item",
65         "Enter new item", QLineEdit::Normal, c_text);
66
67     QString s_text = r_text.simplified();
68

```

```

69     if (!s_text.isEmpty()) {
70
71         QListWidgetItem *item = lw->takeItem(r);
72         delete item;
73         lw->insertItem(r, s_text);
74         lw->setCurrentRow(r);
75     }
76 }
77
78 void ListWidget::removeItem() {
79
80     int r = lw->currentRow();
81
82     if (r != -1) {
83
84         QListWidgetItem *item = lw->takeItem(r);
85         delete item;
86     }
87 }
88
89 void ListWidget::clearItems(){
90
91     if (lw->count() != 0) {
92         lw->clear();
93     }
94 }

```

Создаётся QListWidget и заполняется пятью элементами:

```

1  lw = new QListWidget(this);
2  lw->addItem("The Omen");
3  lw->addItem("The Exorcist");
4  lw->addItem("Notes on a scandal");
5  lw->addItem(" Fargo");
6  lw->addItem("Capote");

```

**Добавление нового элемента в виджет списка выполняется с помощью метода addItem():**

- ➔ Данный метод открывает диалоговое окно ввода, которое возвращает строковое значение.
- ➔ Затем мы удаляем возможные пробельные символы из строки с помощью метода simplified().
- ➔ Метод QString::simplified() возвращает строку, в которой пробельные символы удалены в начале и в конце, а все пробельные символы, находящиеся внутри строки, заменены одиночным пробелом.
- ➔ Если возвращаемая строка не пуста, то мы добавляем её в конец виджета списка.
- ➔ Наконец, выделяем текущий вставленный элемент с помощью метода setCurrentRow().

Код:

```

1  void ListWidget::addItem() {
2
3      QString c_text = QInputDialog::getText(this, "Item", "Enter new item");

```

```

4   QString s_text = c_text.simplified();
5
6   if (!s_text.isEmpty()) {
7
8       lw->addItem(s_text);
9       int r = lw->count() - 1;
10      lw->setCurrentRow(r);
11  }
12 }

```

**Переименование элемента** состоит из нескольких шагов:

- ➔ Во-первых, мы получаем текущий элемент списка и номер строки, в которой он находится, с помощью метода `currentItem()`.
- ➔ Текст элемента отображается в диалоговом окне `QInputDialog`. Строка, возвращаемая из диалогового окна, для удаления потенциальных пробельных символов, обрабатывается методом `simplified()`.
- ➔ Затем мы извлекаем старый элемент с помощью метода `takeItem()` и заменяем его на другой элемент с помощью метода `insertItem()`.
- ➔ Затем удаляем элемент, извлеченный методом `takeItem()` (поскольку извлеченные элементы больше не управляются Qt, то это нужно сделать ручками).
- ➔ Наконец, при помощи метода `setCurrentRow()` задаётся новый элемент.

Код:

```

1  void ListWidget::renameItem() {
2
3      QListWidgetItem *curitem = lw->currentItem();
4
5      int r = lw->row(curitem);
6      QString c_text = curitem->text();
7      QString r_text = QInputDialog::getText(this, "Item",
8          "Enter new item", QLineEdit::Normal, c_text);
9
10     QString s_text = r_text.simplified();
11
12     if (!s_text.isEmpty()) {
13
14         QListWidgetItem *item = lw->takeItem(r);
15         delete item;
16         lw->insertItem(r, s_text);
17         lw->setCurrentRow(r);
18     }
19 }

```

**Удаление определённого элемента из списка осуществляется с помощью метода `removeItem()`.**

Сначала мы получаем текущую строку с помощью метода `currentRow()` (он возвращает -1, если строк больше не осталось), затем текущий выбранный элемент извлекается с помощью метода `takeItem()`:

```

1  void ListWidget::removeItem() {
2

```

```

3   int r = lw->currentRow();
4
5   if (r != -1) {
6
7       QListWidgetItem *item = lw->takeItem(r);
8       delete item;
9   }
10 }

```

Далее метод `clear()` удаляет все элементы из виджета списка:

```

1 void ListWidget::clearItems(){
2
3     if (lw->count() != 0) {
4         lw->clear();
5     }
6 }

```

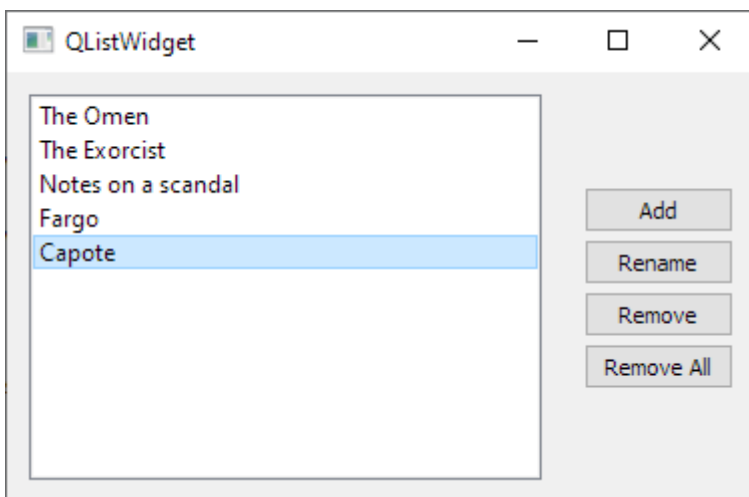
Главный файл программы — `main.cpp`:

```

1 #include <QApplication>
2 #include "listwidget.h"
3
4 int main(int argc, char *argv[]) {
5
6     QApplication app(argc, argv);
7
8     ListWidget window;
9
10    window.setWindowTitle("QListWidget");
11    window.show();
12
13    return app.exec();
14 }

```

Результат:



## Виджет `QProgressBar`

**QProgressBar** — это индикатор процесса (или ещё «индикатор выполнения»), который используется для визуального представления пользователю ход выполнения определённого процесса/операции.

В следующем примере у нас присутствуют **QProgressBar** и две кнопки. Одна из этих кнопок запускает таймер, который, в свою очередь, обновляет индикатор выполнения. Другая кнопка останавливает таймер.

Заголовочный файл — `progressbar.h`:

```
1  #pragma once
2
3  #include <QWidget>
4  #include <QProgressBar>
5  #include <QPushButton>
6
7  class ProgressBarEx : public QWidget {
8
9      Q_OBJECT
10
11     public:
12         ProgressBarEx(QWidget *parent = 0);
13
14     private:
15         int progress;
16         QTimer *timer;
17         QProgressBar *pbar;
18         QPushButton *startBtn;
19         QPushButton *stopBtn;
20         static const int DELAY = 200;
21         static const int MAX_VALUE = 100;
22
23         void updateBar();
24         void startMyTimer();
25         void stopMyTimer();
26 };
```

Файл реализации — `progressbar.cpp`:

```
1  #include <QProgressBar>
2  #include <QTimer>
3  #include <QGridLayout>
4  #include "progressbar.h"
5
6  ProgressBarEx::ProgressBarEx(QWidget *parent)
7      : QWidget(parent) {
8
9      progress = 0;
10     timer = new QTimer(this);
11     connect(timer, &QTimer::timeout, this, &ProgressBarEx::updateBar);
12
13     QGridLayout *grid = new QGridLayout(this);
14     grid->setColumnStretch(2, 1);
15 }
```

```
16 pbar = new QProgressBar();
17 grid->addWidget(pbar, 0, 0, 1, 3);
18
19 startBtn = new QPushButton("Start", this);
20 connect(startBtn, &QPushButton::clicked, this, &ProgressBarEx::startMyTimer);
21 grid->addWidget(startBtn, 1, 0, 1, 1);
22
23 stopBtn = new QPushButton("Stop", this);
24 connect(stopBtn, &QPushButton::clicked, this, &ProgressBarEx::stopMyTimer);
25 grid->addWidget(stopBtn, 1, 1);
26 }
27
28 void ProgressBarEx::startMyTimer() {
29
30     if (progress >= MAX_VALUE) {
31
32         progress = 0;
33         pbar->setValue(0);
34     }
35
36     if (!timer->isActive()) {
37
38         startBtn->setEnabled(false);
39         stopBtn->setEnabled(true);
40         timer->start(Delay);
41     }
42 }
43
44 void ProgressBarEx::stopMyTimer() {
45
46     if (timer->isActive()) {
47
48         startBtn->setEnabled(true);
49         stopBtn->setEnabled(false);
50         timer->stop();
51     }
52 }
53
54 void ProgressBarEx::updateBar() {
55
56     progress++;
57
58     if (progress <= MAX_VALUE) {
59
60         pbar->setValue(progress);
61     } else {
62
63         timer->stop();
64         startBtn->setEnabled(true);
65         stopBtn->setEnabled(false);
66     }
67 }
```

QTimer используется для управления виджетом QProgressBar:

```
1 timer = new QTimer(this);
2 connect(timer, &QTimer::timeout, this, &ProgressBarEx::updateBar);
```

Создаётся экземпляр QProgressBar. Минимальные и максимальные значения по умолчанию указываются в диапазоне от 0 до 100:

```
1 pbar = new QProgressBar();
```

В зависимости от состояния индикатора выполнения, кнопки могут быть включены или выключены. Эта возможность реализуется с помощью метода setEnabled():

```
1 if (!timer->isActive()) {
2
3     startBtn->setEnabled(false);
4     stopBtn->setEnabled(true);
5     timer->start(DELAY);
6 }
```

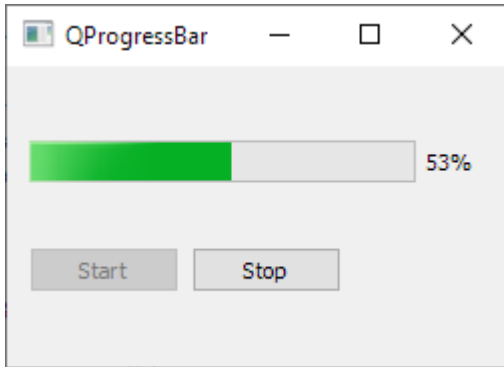
Ход выполнения операции сохраняется в переменной progress. Метод setValue() обновляет текущее значение индикатора выполнения:

```
1 void ProgressBarEx::updateBar() {
2
3     progress++;
4
5     if (progress <= MAX_VALUE) {
6
7         pbar->setValue(progress);
8     } else {
9
10        timer->stop();
11        startBtn->setEnabled(true);
12        stopBtn->setEnabled(false);
13    }
14 }
```

Главный файл программы — main.cpp:

```
1 #include <QApplication>
2 #include "progressbar.h"
3
4 int main(int argc, char *argv[]) {
5
6     QApplication app(argc, argv);
7
8     ProgressBarEx window;
9
10    window.resize(250, 150);
11    window.setWindowTitle("QProgressBar");
12    window.show();
13
14    return app.exec();
}
```

Результат:



## Виджет QPixmap

**Виджет QPixmap** используется для работы с изображениями. Он оптимизирован для показа изображений на экране.

В следующем примере мы выведем на экран изображение знаменитого Бойницкого замка, расположенного в центральной части Словакии.

Заголовочный файл — pixmap.h:

```
1 #pragma once
2
3 #include <QWidget>
4
5 class QPixmap : public QWidget {
6
7     public:
8         QPixmap(QWidget *parent = 0);
9 };
```

Файл реализации — pixmap.cpp:

```
1 #include <QPixmap>
2 #include <QLabel>
3 #include <QHBoxLayout>
4 #include "pixmap.h"
5
6 QPixmap::Pixmap(QWidget *parent)
7     : QWidget(parent) {
8
9     QHBoxLayout *hbox = new QHBoxLayout(this);
10
11     QPixmap pixmap("bojnice.jpg");
12
13     QLabel *label = new QLabel(this);
14     label->setPixmap(pixmap);
15 }
```



```
16 hbox->addWidget(label, 0, Qt::AlignTop);
17 }
```

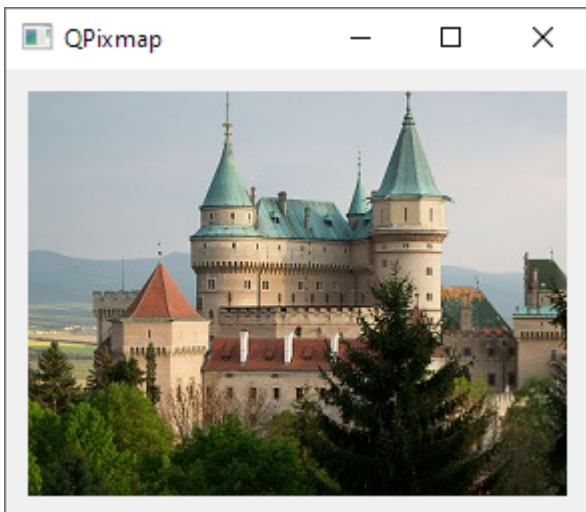
Мы создаём `pixmap` — пространство, в которое поместим наше растровое изображение. Затем созданный `pixmap` мы помещаем в виджет метки:

```
1 QPixmap pixmap("bojnice.jpg");
2
3 QLabel *label = new QLabel(this);
4 label->setPixmap(pixmap);
```

Главный файл программы — `main.cpp`:

```
1 #include <QApplication>
2 #include "pixmap.h"
3
4 int main(int argc, char *argv[]) {
5
6     QApplication app(argc, argv);
7
8     QPixmap window;
9
10    window.setWindowTitle("QPixmap");
11    window.show();
12
13    return app.exec();
14 }
```

Результат:



## Виджет QSplitter

**Виджет QSplitter** позволяет пользователю управлять размером дочерних виджетов, перетаскивая границу между дочерними элементами.

В следующем примере мы рассмотрим три виджета `QFrame`, организованных с помощью двух **сплиттеров** (или ещё «разделителей»).

Заголовочный файл — `splitter.h`:

```

1 #pragma once
2
3 #include <QWidget>
4
5 class Splitter : public QWidget {
6
7     public:
8         Splitter(QWidget *parent = 0);
9 };

```

Файл реализации — splitter.cpp:

```

1 #include <QFrame>
2 #include <QSplitter>
3 #include <QHBoxLayout>
4 #include "splitter.h"
5
6 Splitter::Splitter(QWidget *parent)
7     : QWidget(parent) {
8
9     QHBoxLayout *hbox = new QHBoxLayout(this);
10
11     QFrame *topleft = new QFrame(this);
12     topleft->setFrameShape(QFrame::StyledPanel);
13
14     QFrame *topright = new QFrame(this);
15     topright->setFrameShape(QFrame::StyledPanel);
16
17     QSplitter *splitter1 = new QSplitter(Qt::Horizontal, this);
18     splitter1->addWidget(topleft);
19     splitter1->addWidget(topright);
20
21     QFrame *bottom = new QFrame(this);
22     bottom->setFrameShape(QFrame::StyledPanel);
23
24     QSplitter *splitter2 = new QSplitter(Qt::Vertical, this);
25     splitter2->addWidget(splitter1);
26     splitter2->addWidget(bottom);
27
28     QList<int> sizes({50, 100});
29     splitter2->setSizes(sizes);
30
31     hbox->addWidget(splitter2);
32 }

```

Создаём виджет `splitter1` и добавляем в него два виджета `frame`:

```

1 QSplitter *splitter1 = new QSplitter(Qt::Horizontal, this);
2 splitter1->addWidget(topleft);
3 splitter1->addWidget(topright);

```

Мы также можем добавить один сплиттер к другому сплиттеру:

```
1 QSplitter *splitter2 = new QSplitter(Qt::Vertical, this);
2 splitter2->addWidget(splitter1);
```

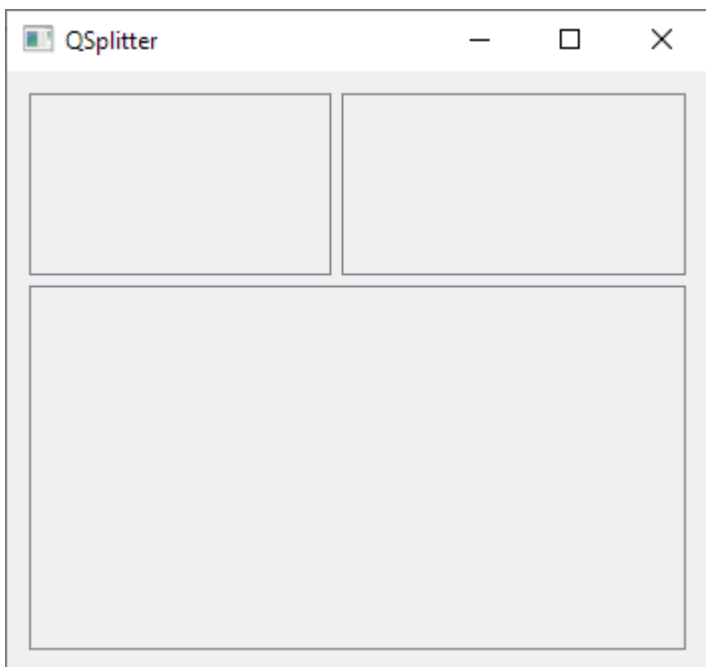
С помощью метода `setSizes()` устанавливаем размер дочерних виджетов сплиттера:

```
1 QList<int> sizes({50, 100});
2 splitter2->setSizes(sizes);
```

Главный файл программы — `main.cpp`:

```
1 #include <QDesktopWidget>
2 #include <QApplication>
3 #include "splitter.h"
4
5 int main(int argc, char *argv[]) {
6
7     QApplication app(argc, argv);
8
9     Splitter window;
10
11     window.resize(350, 300);
12     window.setWindowTitle("QSplitter");
13     window.show();
14
15     return app.exec();
16 }
```

Результат:



*Примечание:* В некоторых темах рабочего стола сплиттер может быть не очень хорошо виден.

## Виджет `QTableWidget`

**QTableWidget** — это уникальный виджет, используемый в приложениях для работы с электронными таблицами (его ещё называют «**виджет сетки**»). Это один из самых сложных виджетов.

В следующем примере мы создадим таблицу с помощью виджета **QTableWidget** и выведем её на экран.

Заголовочный файл — `table.h`:

```
1 #pragma once
2
3 #include <QWidget>
4
5 class Table : public QWidget {
6
7     public:
8         Table(QWidget *parent = 0);
9 };
```

Файл реализации — `table.cpp`:

```
1 #include <QHBoxLayout>
2 #include <QTableWidget>
3 #include "table.h"
4
5 Table::Table(QWidget *parent)
6     : QWidget(parent) {
7
8     QHBoxLayout *hbox = new QHBoxLayout(this);
9
10    QTableWidget *table = new QTableWidget(25, 25, this);
11
12    hbox->addWidget(table);
13 }
```

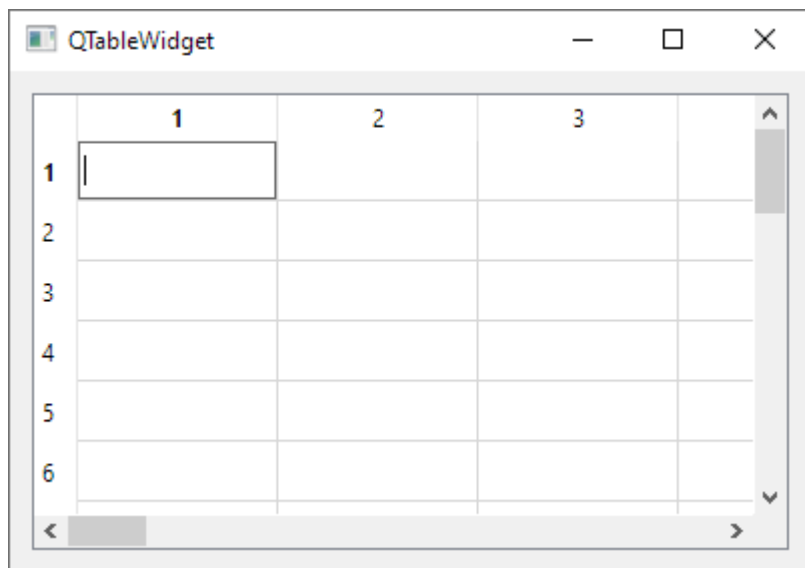
Создаём виджет таблицы с 25 строками и 25 столбцами:

```
1 QTableWidget *table = new QTableWidget(25, 25, this);
```

Главный файл программы — `main.cpp`:

```
1 #include <QApplication>
2 #include "table.h"
3
4 int main(int argc, char *argv[]) {
5
6     QApplication app(argc, argv);
7
8     Table window;
9
10    window.resize(400, 250);
11    window.setWindowTitle("QTableWidget");
12    window.show();
13
14    return app.exec();
15 }
```

Результат:



The image shows a Qt TableWidget window with a 6x4 grid. The columns are labeled 1, 2, 3, and an unlabeled fourth column. The rows are labeled 1 through 6. The first cell (row 1, column 1) is selected and contains a vertical cursor. The window has a title bar with 'QTableWidget' and standard minimize, maximize, and close buttons. Scroll bars are visible on the right and bottom.

	1	2	3	
1				
2				
3				
4				
5				
6				