

С++: разработка программ с графическим интерфейсом на Qt

Разработка графического интерфейса в Qt

Знакомство с классами QWindow и QWidget. Подключение сторонних библиотек

[QWidget. Виды виджетов. Иерархия виджетов](#)

[Класс QWidget](#)

[Подключение OpenGL и других сторонних библиотек вывода](#)

[Разработка форм в Qt Designer. Использование форм в проектах](#)

[Компоновка виджетов. Политика изменения размера](#)

[Соединение сигналов со слотами. Фокус ввода](#)

[Таблицы стилей. Цветовая палитра](#)

[Таблицы стилей](#)

[Цветовая палитра](#)

[Создание собственных виджетов](#)

[Практическое задание](#)

[Дополнительные материалы](#)

[Используемая литература](#)

QWidget. Виды виджетов. Иерархия виджетов

Класс QWidget

Класс `QWidget` — базовый класс всех объектов пользовательского интерфейса. Виджет — основа пользовательского интерфейса. Виджет не только отображает компоненты, но и обрабатывает события мыши и клавиатуры. Все виджеты прямоугольные, сортируются в Z-порядке (чем позже создан виджет, тем выше приоритет обработки событий относительно более ранних виджетов при совпадении координат обработки) и содержат основные свойства объекта, такие как события, политику размеров, настройку стилей, показатель активности/неактивности элемента, методы компоновки элементов по рабочей области окна. Виджет без родительского виджета всегда является независимым окном (виджетом верхнего уровня). Для таких виджетов методы `setWindowTitle()` и `setWindowIcon()` устанавливают заголовок окна и иконку. Если виджет используется как контейнер, объединяющий дочерние виджеты, он называется комбинированным виджетом. Такой виджет может быть создан на основе виджета, который имеет определенные визуальные свойства — как, например, `QFrame`, — а его дочерние виджеты можно разместить в нем с помощью компоновщика или `QMainWindow`, который является контейнером для всех виджетов графического интерфейса.

Виджеты получают события, которые порождаются типичными действиями пользователя. Qt посылает события виджету через вызовы специальных функций обработчиков событий с аргументом в виде подкласса от `QEvent`, содержащего информацию о событии.

Если виджет является контейнером для дочерних виджетов, то, скорее всего, не понадобится дополнительно реализовывать никаких обработчиков событий. Если нужно отловить щелчок мыши в дочернем виджете, вызовите его функцию `underMouse()` внутри функции `mousePressEvent()` виджета.

Подключение OpenGL и других сторонних библиотек вывода

Фреймворк обладает весьма богатым набором классов, облегчающим разработку программного продукта, но порой приходится решать специфические задачи и использовать для этого сторонние библиотеки, аналогов которых нет в Qt, например `OpenCL`, `OpenGL`, `OpenSSL`. Подключение любой библиотеки происходит по одному шаблону:

- указываем компилятору путь к заголовкам подключаемой библиотеки (для некоторых библиотек утилита `qmake` автоматически определяет путь к заголовкам);
- указываем линковщику путь к статическим библиотекам;

- указываем линковщику, какие именно статические библиотеки нужно добавить в сборку проекта.

Создадим новый проект без формы. Отредактируем файл **qmake** (расширение **.pro**):

```
#-----
#
# Project created by QtCreator 2019-06-23T15:47:34
#
#-----

QT      += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = OpenGLTest
TEMPLATE = app

DEFINES += QT_DEPRECATED_WARNINGS

CONFIG += c++11

SOURCES += \
    main.cpp \
    mainwindow.cpp

HEADERS += \
    mainwindow.h

LIBS += -lOpenGL32

qnx: target.path = /tmp/${TARGET}/bin
else: unix:!android: target.path = /opt/${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

Для работы с функциями **OpenGL** нам нужно подключить статическую библиотеку: для ОС Windows — **OpenGL32**, для Linux — **libGL**. Чтобы подключить библиотеку, нужно в файле проекта добавить ее к переменной **LIBS** с параметром **-l**. Следующим шагом отредактируем класс главного окна. Поменяем родительский класс на **Window**, а также наследуем методы от класса **QOpenGLFunctions**:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QWindow>
#include <QOpenGLFunctions>

class MainWindow : public QWindow, protected QOpenGLFunctions
{
    Q_OBJECT
```

```

public:
    MainWindow(QWindow *parent = 0);
    ~MainWindow();
    virtual void render(QPainter *painter);
    virtual void render();

    virtual void initialize();

public slots:
    void renderLater();
    void renderNow();

protected:
    bool event(QEvent *event) override;

    void exposeEvent(QExposeEvent *event) override;

private:

    QOpenGLContext *m_context;
    QOpenGLPaintDevice *m_device;
};

#endif // MAINWINDOW_H

```

Также нужно получить контекст клиентской области окна (**m_context**). Отрисовка с аппаратным ускорением нуждается в перерисовке при каждом обновлении окна, поэтому нам нужно перехватить соответствующие событие:

```

#include "mainwindow.h"
#include <QOpenGLPaintDevice>
#include <QPainter>

MainWindow::MainWindow(QWindow *parent)
    : QWindow(parent)
    , m_context(0)
    , m_device(0)
{
    setSurfaceType(QWindow::OpenGLSurface);
}

MainWindow::~MainWindow()
{
}

void MainWindow::render(QPainter *painter)
{

```

```

    Q_UNUSED(painter);
}

void MainWindow::initialize()
{
}

void MainWindow::render()
{
    if (!m_device)
        m_device = new QOpenGLPaintDevice;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

    m_device->setSize(size() * devicePixelRatio());
    m_device->setDevicePixelRatio(devicePixelRatio());

    QPainter painter(m_device);
    render(&painter);
}

void MainWindow::renderLater()
{
    requestUpdate();
}

bool MainWindow::event(QEvent *event)
{
    switch (event->type()) {
        case QEvent::UpdateRequest:
            renderNow();
            return true;
        default:
            return QWindow::event(event);
    }
}

void MainWindow::exposeEvent(QExposeEvent *event)
{
    Q_UNUSED(event);

    if (isExposed())
        renderNow();
}

void MainWindow::renderNow()
{
    if (!isExposed())
        return;

    bool needsInitialize = false;

    if (!m_context) {
        m_context = new QOpenGLContext(this);
    }
}

```

```

        m_context->setFormat(requestedFormat());
        m_context->create();

        needsInitialize = true;
    }

    m_context->makeCurrent(this);

    if (needsInitialize) {
        initializeOpenGLFunctions();
        initialize();
    }

    render();

    m_context->swapBuffers(this);
}

```

В конструкторе окна для отрисовки устанавливаем формат отображения пикселей, нужный для работы **OpenGL**. Осталось отредактировать код в файле **main.cpp**:

```

#include "mainwindow.h"
#include <QApplication>
#include <QScreen>
#include <QtGui/QOpenGLShaderProgram>

class DrawPoligon : public MainWindow
{
public:
    DrawPoligon();
    ~DrawPoligon();
    void initialize() override;
    void render() override;

private:
    QOpenGLShaderProgram *m_program;
    GLuint buff;
    GLint pos, wsize;
};

DrawPoligon::DrawPoligon()
    : m_program(0)
    , buff(0)
{
}

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

```

```

QGuiApplication app(argc, argv);

QSurfaceFormat format;
format.setSamples(16);

DrawPoligon window;
window.setFormat(format);
window.resize(640, 480);
window.show();

return app.exec();
}

static const char *vertexShaderSource =
    "attribute vec2 input;"
    "void main(){\"
    \"gl_Position = vec4(input, 0., 1.0);\"
    \"}\";

static const char *fragmentShaderSource =
    \"uniform int sizes[2];\"
    \"void main(){\"
    \"vec2 coord = gl_FragCoord.xy;\"
    \"gl_FragColor = vec4(coord.x / float(sizes[0]), 0.5, coord.y /\"
    float(sizes[1]), 1.0);\"
    \"}\";

void DrawPoligon::initialize()
{
    m_program = new QOpenGLShaderProgram(this);
    m_program->addShaderFromSourceCode(QOpenGLShader::Vertex,
vertexShaderSource);
    m_program->addShaderFromSourceCode(QOpenGLShader::Fragment,
fragmentShaderSource);
    bool rez = m_program->link();
    if (!rez) qApp->quit();
    glGenBuffers(1, &buff);
    if (!buff) qApp->quit();
    pos = m_program->attributeLocation(\"input\");
    wsizes = m_program->uniformLocation(\"sizes\");

    float vert[] = {-1.0f, -1.0f,
                    1.0f, -1.0f,
                    1.0f, 1.0f,
                    -1.0f, 1.0f,
                    };

    glBindBuffer(GL_ARRAY_BUFFER, buff);
    glBufferData(GL_ARRAY_BUFFER, sizeof vert, vert, GL_STATIC_DRAW);
    glVertexAttribPointer(pos, 2, GL_FLOAT, GL_FALSE, 0, 0);
    glEnableVertexAttribArray(pos);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
}

```

```

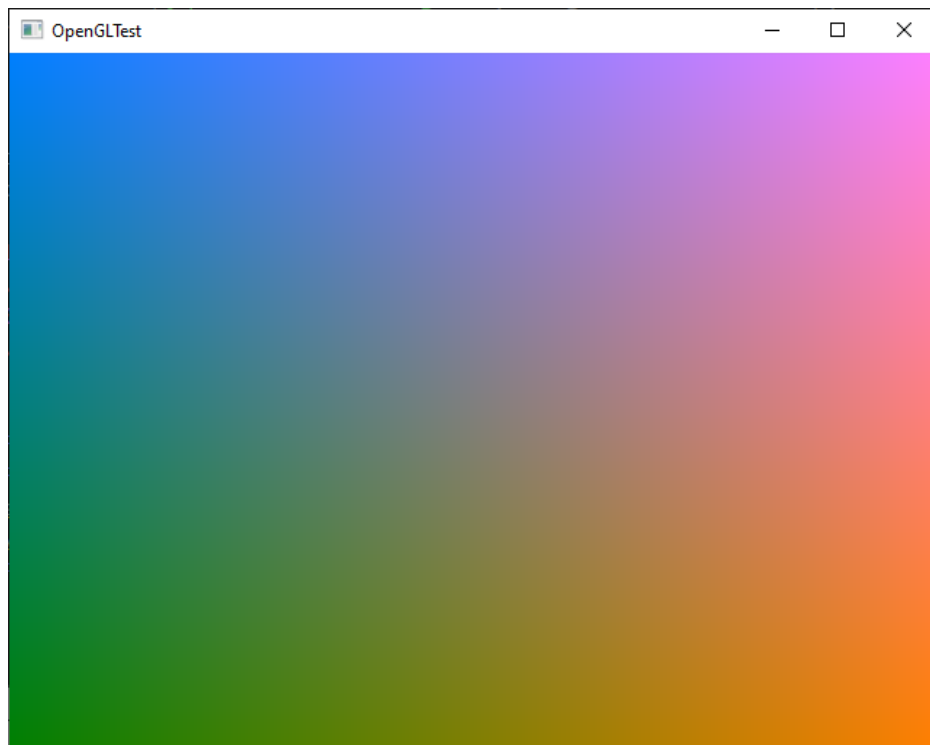
void DrawPoligon::render()
{
    const qreal retinaScale = devicePixelRatio();
    int sizes[] = {int(width() * retinaScale), int(height() * retinaScale)};

    glViewport(0, 0, sizes[0], sizes[1]);

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    m_program->bind();
    m_program->setUniformValueArray(wsizes, sizes, 2);
    glBindBuffer(GL_ARRAY_BUFFER, buff);
    glDrawArrays(GL_QUADS, 0, 4);
    glBindBuffer(GL_ARRAY_BUFFER, 0);
    m_program->release();
}
DrawPoligon::~DrawPoligon()
{
    if (buff) glDeleteBuffers(1, &buff);
}

```

На базе созданного класса создаем новый, который и будет использоваться для отрисовки **OpenGL**. Для удобства создания шейдеров в Qt есть класс **QOpenGLShaderProgram**, который упрощает компиляцию шейдеров и линковку программы для выполнения на GPU. Окно самой программы будет выглядеть так:

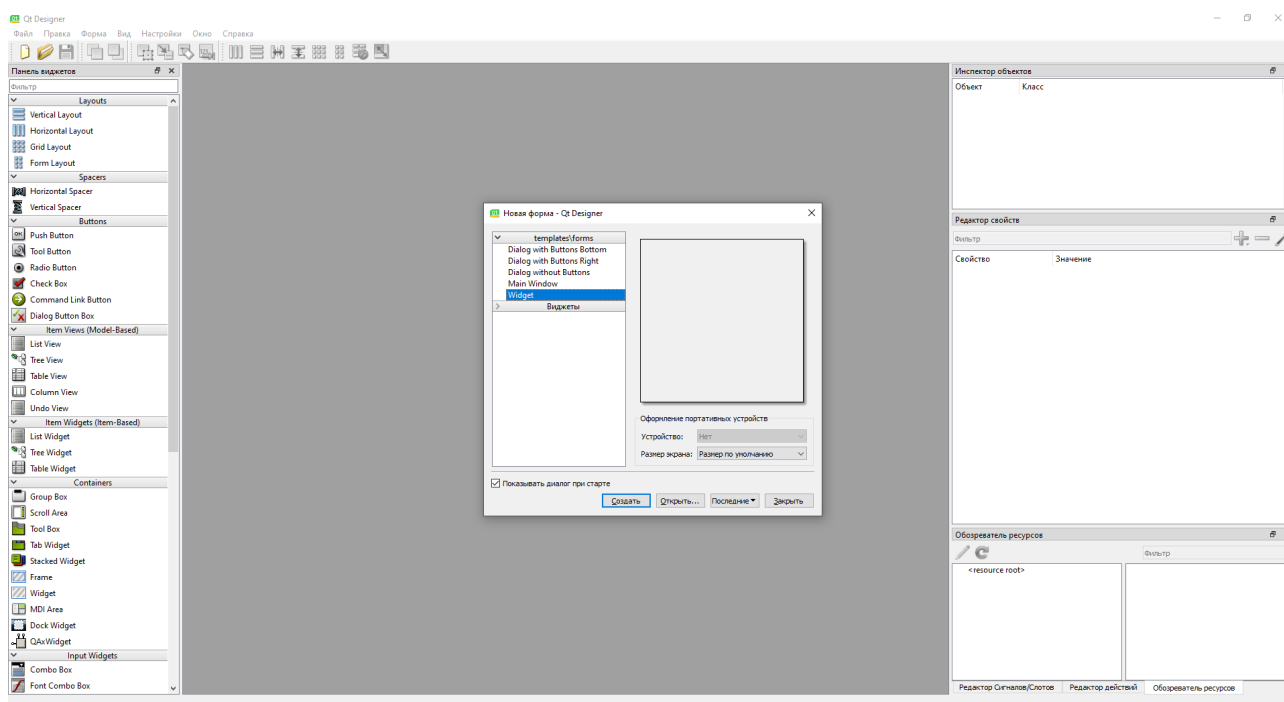


Использовать **OpenGL** на Qt сложнее, чем на специальных кроссплатформенных библиотеках, то есть использовать Qt в качестве основы для компьютерных игр — не лучшее решение.

Разработка форм в Qt Designer.

Использование форм в проектах

Любой графический интерфейс можно создать с помощью кода: добавить переменные виджетов, слоев. Такой подход разработки интерфейса медленный: придется вручную запускать методы по установке размеров виджетов, надписей. При этом код класса формы будет громоздким и сложным в поддержке. Запустим программу Qt Designer, один из компонентов, поставляемых вместе с фреймворком Qt. Он позволяет проектировать и создавать виджеты и диалоги, используя экранные формы с теми же виджетами, что будут использоваться в вашем приложении. Компоненты, созданные с помощью Qt Designer, могут также получить преимущество сигналов и слотов Qt, их можно предварительно просмотреть и убедиться, что они будут выглядеть и вести себя так, как планировалось.



При запуске предлагается выбрать базовую форму. Помимо стандартных **QMainWindow** и **QWidget** есть шаблоны для создания диалогового окна. Сама программа имеет знакомый интерфейс, который можно увидеть в Qt Creator. В левой части набор объектов для размещения на форме элементов управления и ввода и вывода.

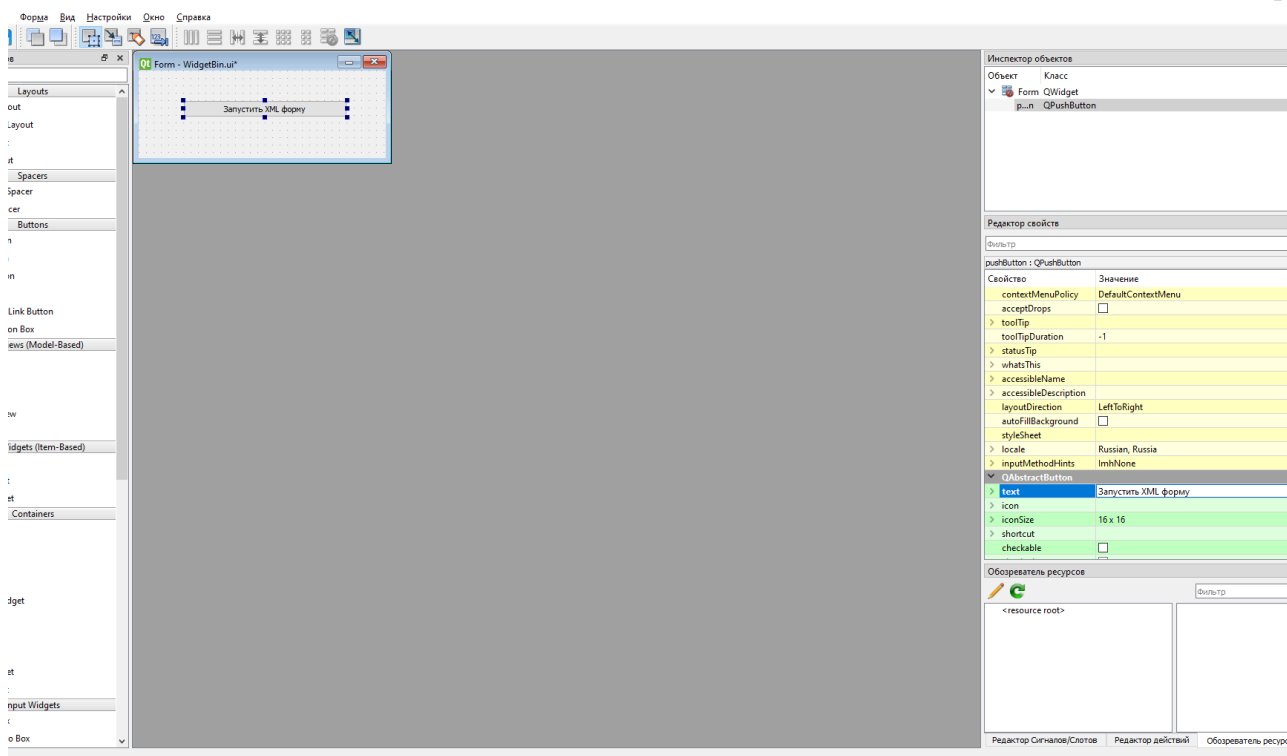
Созданную форму можно подключить к проекту двумя способами:

- во время компиляции (формы будут преобразованы в код C++, который может быть скомпилирован);

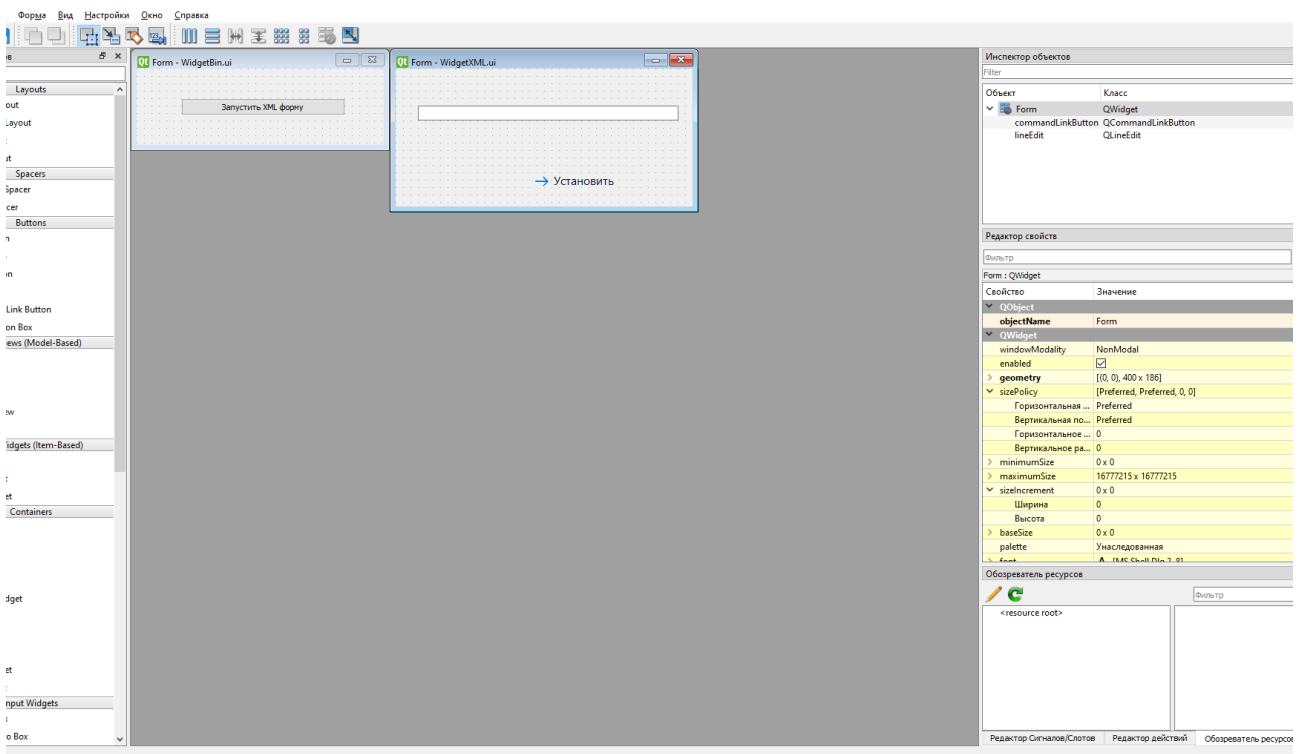
- во время выполнения (формы обрабатываются классом **QUiLoader**, который динамически создает дерево виджетов при анализе XML-файла).

Второй вариант позволяет менять расположение компонентов формы без повторной компиляции.

Создадим новый проект. В качестве базового класса выберем **QWidget** без создания формы. Создадим в Qt Designer форму нового виджета. Сохраним виджет в папку с созданным проектом под именем **WidgetBin**. Разместим на форме кнопку, а в свойствах **text** укажем: «Запустить XML форму».



При нажатии на кнопку запустим вторую форму, расположение компонентов которой будет считываться из файла.



Теперь подключим первую форму к проекту, на ее базе утилита **uic** сгенерирует код построения элементов на C++ и сохранит под именем **ui_Widget.h**:

```
QT += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = demo
TEMPLATE = app
FORMS += WidgetBin.ui //наш виджет
DEFINES += QT_DEPRECATED_WARNINGS
CONFIG += c++11

SOURCES += \
    main.cpp \
    widget.cpp

HEADERS += \
    widget.h

# Default rules for deployment.
qnx: target.path = /tmp/${TARGET}/bin
else: unix:!android: target.path = /opt/${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

Теперь утилита **qmake** автоматически запустит **uic**. Посмотрим на объявление класса **Widget**, который был создан вместе с проектом:

```

#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();
};

#endif // WIDGET_H

```

Подключим заголовок от формы к проекту. Название созданного класса должно соответствовать названию формы в Qt Designer и иметь ключевой префикс **Ui_**. Первая форма называется **Form**, то есть класс будет называться **Ui_Form**. Заранее создадим слот для запуска второй формы:

```

#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include "ui_WidgetBin.h"

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = 0);
    ~Widget();
private:
    Ui_Form *uiForm;
private slots:
    void openXMLForm();
};

#endif // WIDGET_H

```

Осталось создать новый экземпляр класса созданной формы и установить родительский виджет, для которого создавалась форма:

```

#include "widget.h"

Widget::Widget(QWidget *parent)

```

```

    : QWidget(parent)
{
    uiForm = new Ui_Form();
    uiForm->setupUi(this);
}

Widget::~Widget()
{
    delete uiForm;    // не забываем освободить память в деструкторе либо
                     // используем защищенный (умный) указатель QSharedPointer
                     // при объявлении класса
}

void Widget::openXMLForm()
{
}

```

Запускаем программу. Видим, что форма та же, которая была создана в Qt Designer.

Второй вариант — загрузка формы из файла без компиляции. Для загрузки формы нам понадобится загрузить файл и на его основе построить форму. Файл загрузим при помощи класса **QFile**. Теперь по его содержимому построим новое окно. Для этого нам понадобится класс **QtLoader**. Получаем следующий код:

```

#include "widget.h"
#include <QFile>
#include <QFileDialog>
#include <QtUiTools/QtUiLoader>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    uiForm = new Ui_Form();
    uiForm->setupUi(this);
    connect(uiForm->pushButton, SIGNAL(clicked()), this, SLOT(openXMLForm()));
}

Widget::~Widget()
{
    delete uiForm;
}

void Widget::openXMLForm()
{
    QFile file("./WidgetXML.ui");
    if (file.open(QIODevice::ReadOnly))
    {

```

```

        QUiLoader loader;
        QWidget *w = loader.load(&file);
        w->show();
        file.close();
    }
}

```

В этом коде мы видим, что создается объект класса **QUiLoader**. Методом **load()** создается новый виджет, который соответствует созданной форме. Для компиляции необходимо подключить модуль **uitools**.

Осталось получить доступ к виджетам созданного окна. Делается это с помощью метода **findChild**. Обратите внимание, что для использования виджетов должны быть подключены соответствующие заголовки. В этом примере используются виджеты **QLineEdit** и **QCommandLinkButton**.

```

#include "widget.h"
#include <QFile>
#include <QFileDialog>
#include <QtUiTools/QUiLoader>
#include <QLineEdit>
#include <QCommandLinkButton>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    uiForm = new Ui_Form();
    uiForm->setupUi(this);
    connect(uiForm->pushButton, SIGNAL(clicked()), this, SLOT(openXMLForm()));
}

Widget::~~Widget()
{
    delete uiForm;
}

void Widget::openXMLForm()
{
    QFile file("./WidgetXML.ui");
    if (file.open(QIODevice::ReadOnly))
    {
        QUiLoader loader;
        QWidget *w = loader.load(&file);
        w->show();
        file.close();
        QLineEdit *le = w->findChild<QLineEdit*>("lineEdit");
        QCommandLinkButton *clb =
w->findChild<QCommandLinkButton*>("commandLinkButton");
    }
}

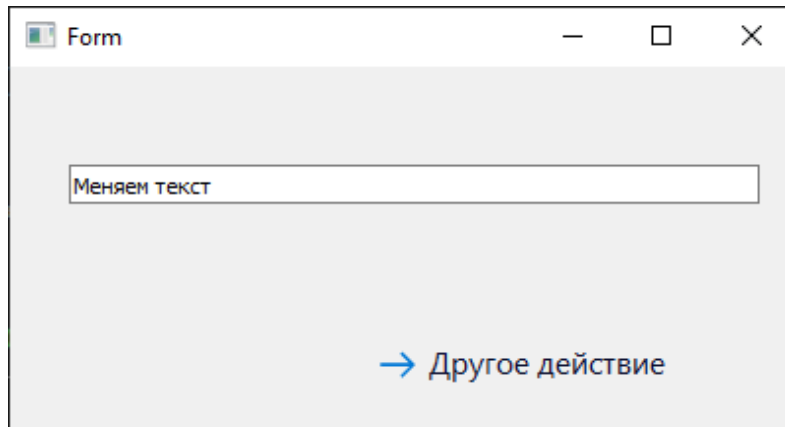
```

```
le->setText("Меняем текст");
clb->setText("Другое действие");

}

}
```

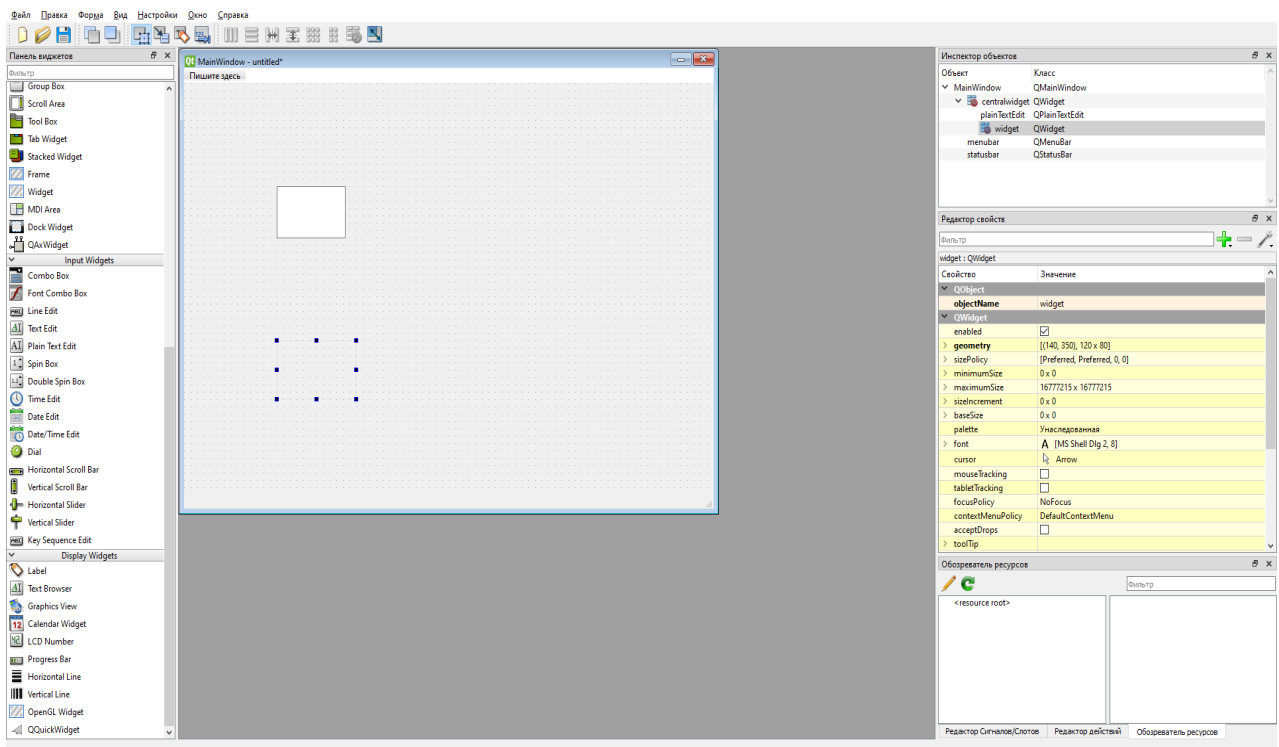
Запустим приложение и нажмем на кнопку «Запустить XML форму». Мы увидим форму с измененными надписями.



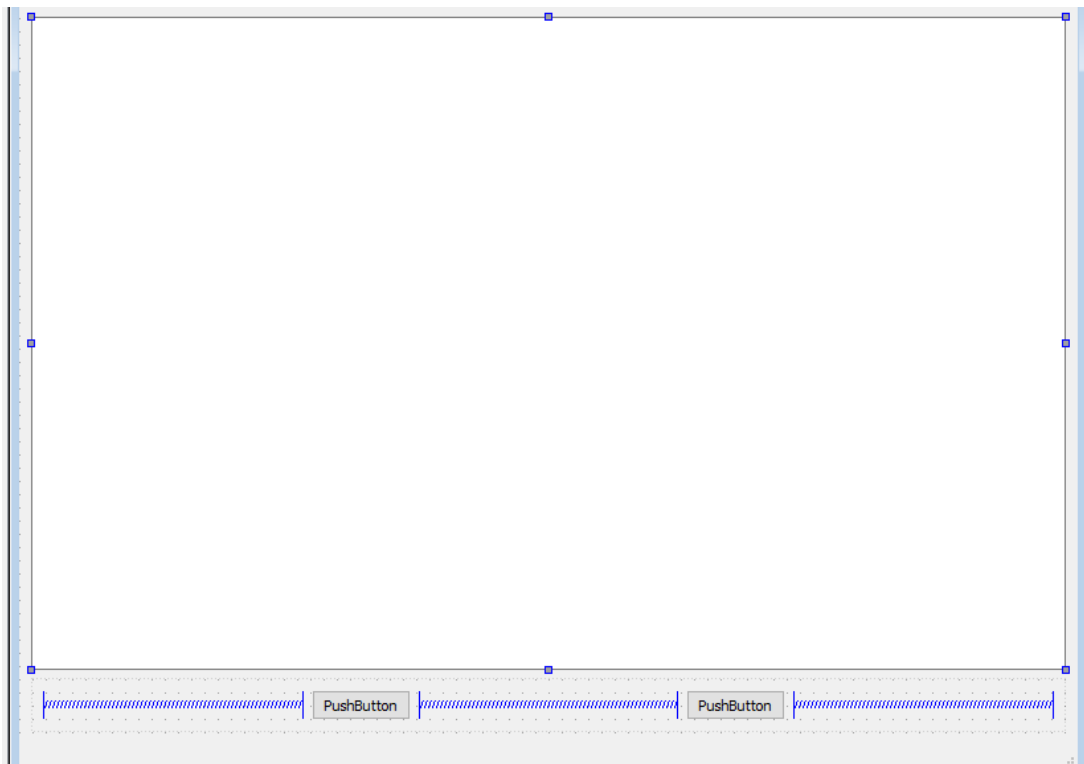
Так придется загружать каждый объект. Получив ссылку на виджет, можно привязать сигнал от его виджета к слоту и использовать созданный виджет в своем ПО.

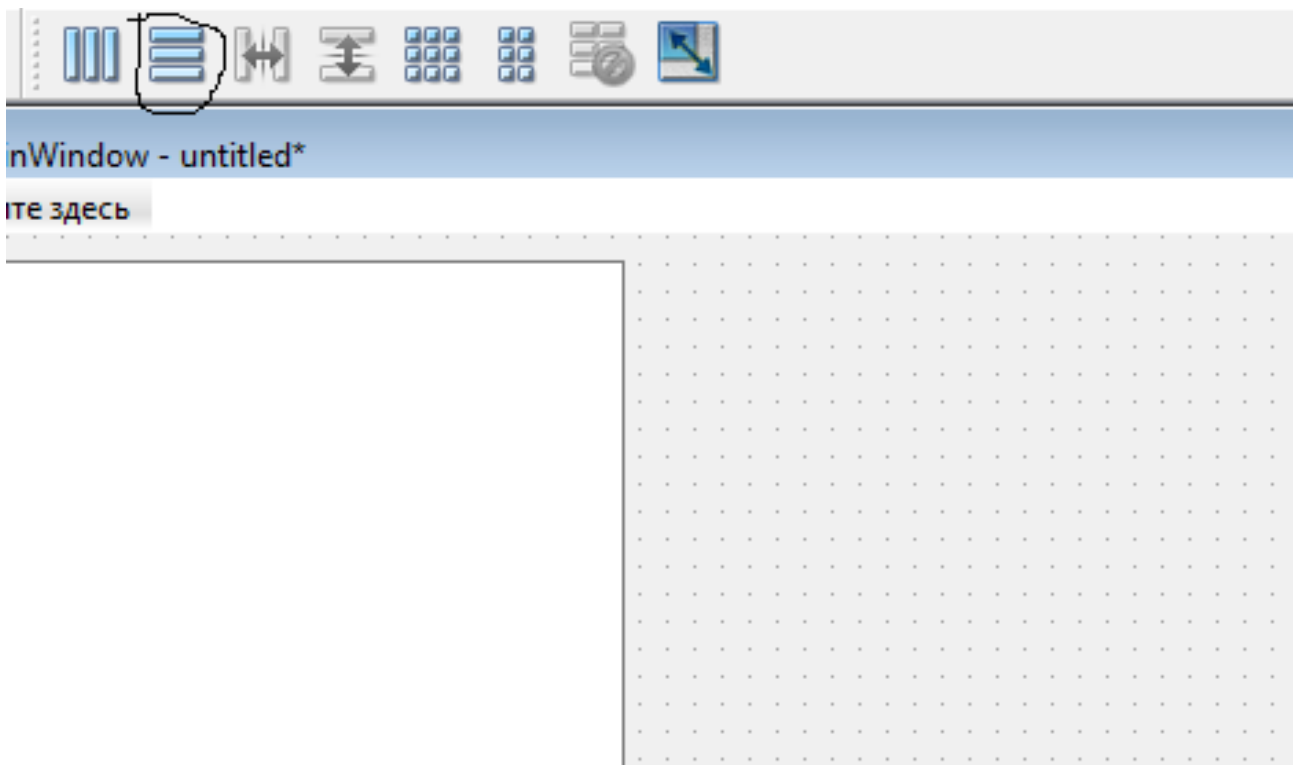
Компоновка виджетов. Политика изменения размера

Слои — это контейнеры для виджетов. Они используются для компоновки виджетов на форме или других виджетах. Есть три основных вида слоев: для вертикальной компоновки, горизонтальной компоновки и компоновки по сетке. Нужно это для динамической переконфигурации виджетов при масштабировании окна. Запустим Qt Designer. Разместим на нем виджеты **QPlainTextEdit** и **QWidget**:

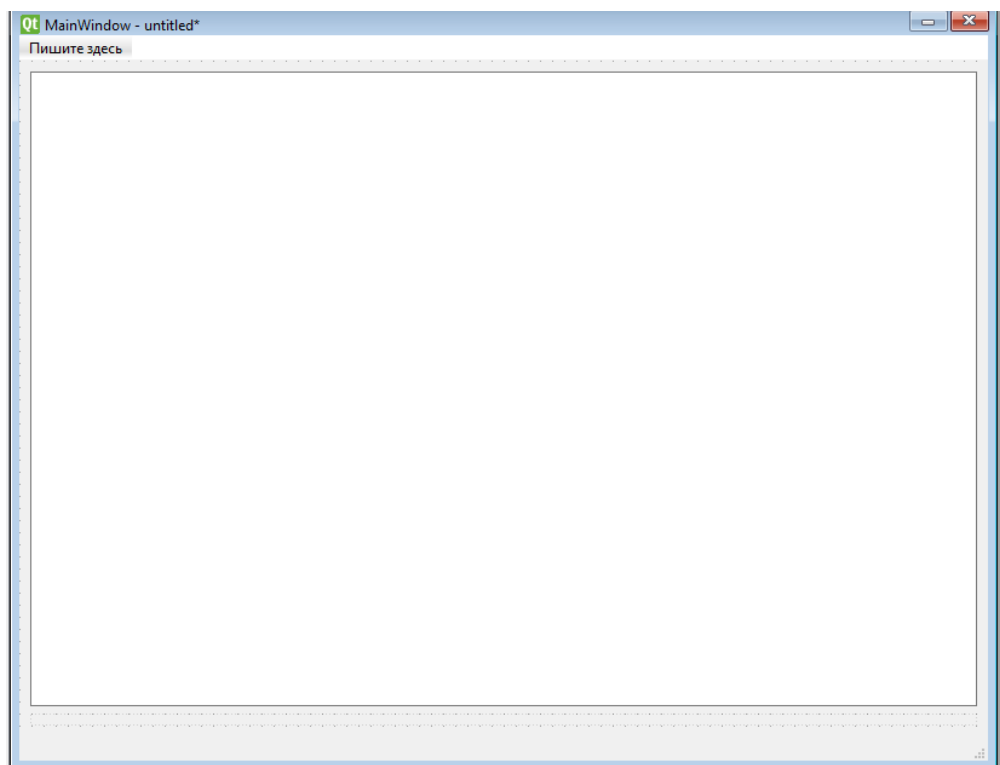


В качестве базового класса был выбран **QMainWindow**. Из предыдущих уроков мы помним, что **QMainWindow** содержит объекты меню, строки состояния и центральный виджет. Выделим главную форму. В верхней части активируются кнопки элементов управления:






Нажмем на обведенную кнопку. Виджеты расположились по вертикали.

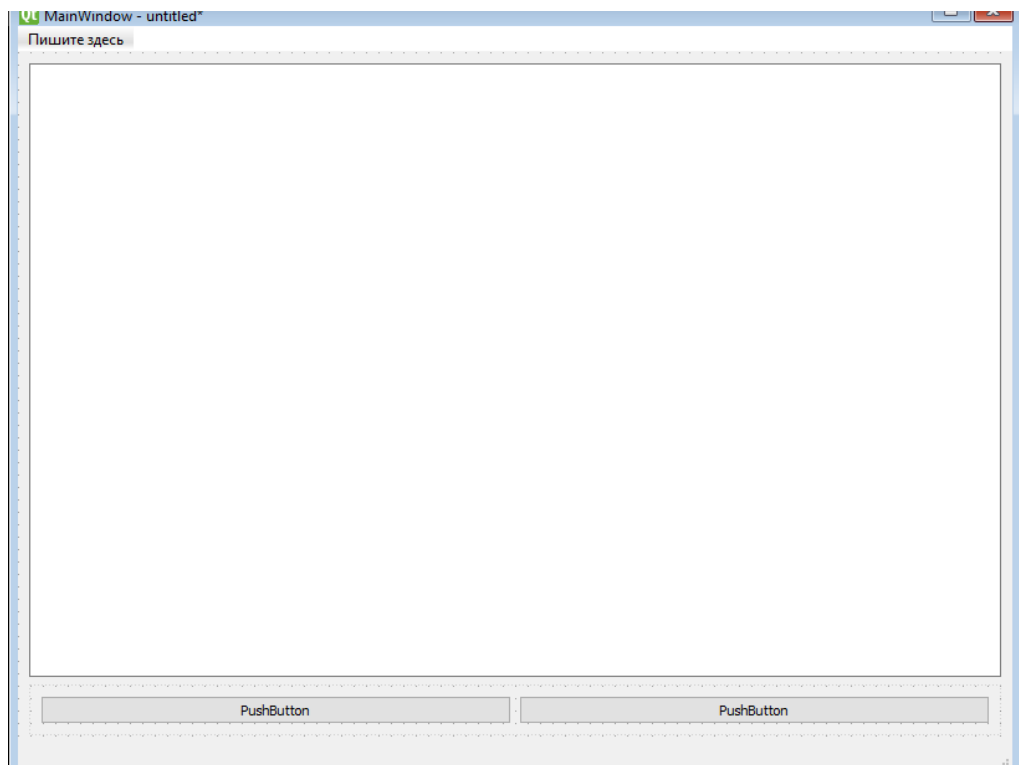


Теперь при масштабировании размеры виджетов будут изменяться пропорционально изменению размера формы. Установим объекту минимальную высоту: так при изменении размера формы компонент не станет меньше, чем указано, а кроме того, компоненты расположатся в одной строке

или столбце слоя.

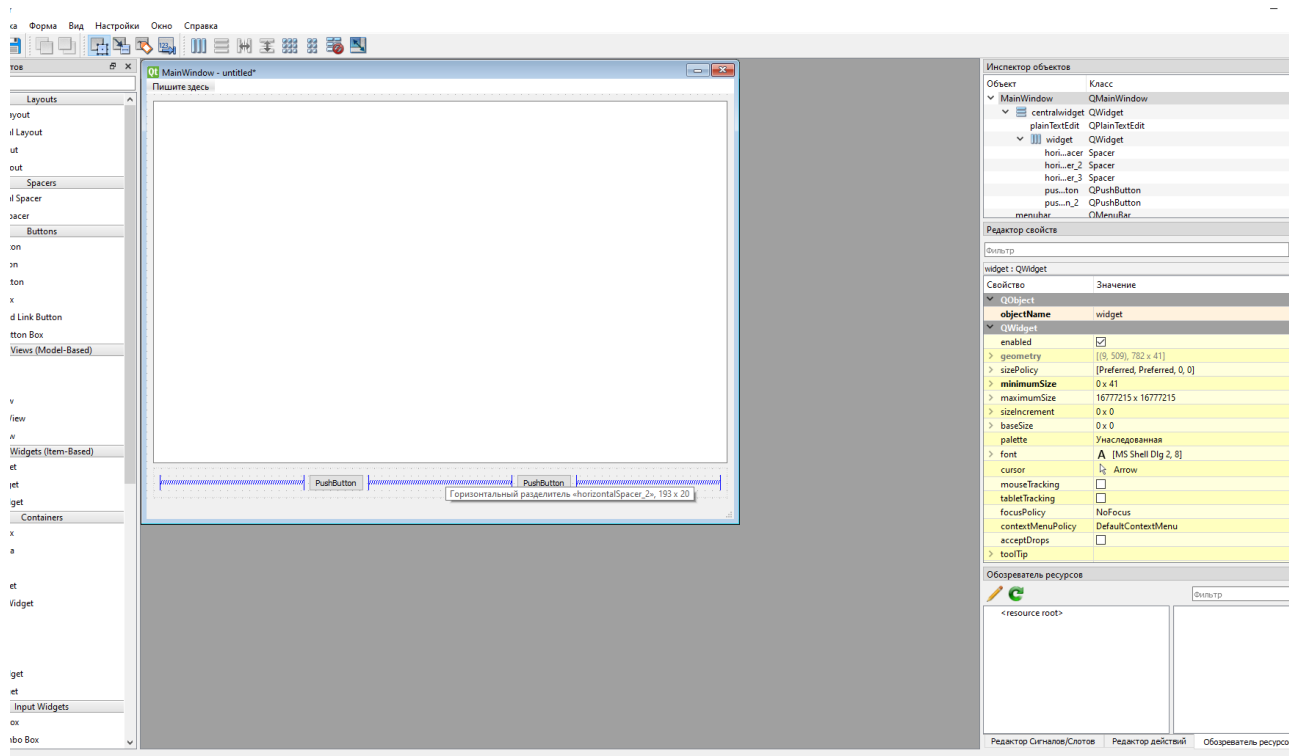
> geometry	[(9, 509), 782 x 41]
> sizePolicy	[Preferred, Preferred, 0, 0]
> minimumSize	0 x 41
> maximumSize	16777215 x 16777215
> sizeIncrement	0 x 0
> baseSize	0 x 0
palette	Унаследованная
> font	[MS Shell Dlg 2, 8]
cursor	 Arrow
mouseTracking	<input type="checkbox"/>
tabletTracking	<input type="checkbox"/>
focusPolicy	NoFocus
contextMenuPolicy	DefaultContextMenu
acceptDrops	<input type="checkbox"/>
> toolTip	
toolTipDuration	-1

Разместим на виджете две кнопки. Применим к виджету горизонтальную компоновку (слева от вертикальной). Получим следующий результат:



Для разделения виджетов используются компоненты **QHorizontalSpacer** и **QVericalSpacer**. Разместим

QHorizontalSpacer перед первой кнопкой, между кнопками и после второй кнопки:



Теперь сделаем что-нибудь посложнее. Расположим на форме четыре виджета **QPlainTextEdit** и скомпонуем по сетке. Выделим верхний левый виджет и в свойствах найдем свойство **sizePolicy**. Оно позволяет менять пропорции виджета на занимаемой части рабочей области окна.

Соединение сигналов со слотами. Фокус ввода

В предыдущих уроках мы уже рассматривали подключение сигналов со слотами. Есть два способа подключения слота к сигналу виджета:

1. Объявление слота в классе объекта, к слоту которого нужно подключить сигнал. Слот создается по шаблону: `void on_widgetName_name_signal(аргументы сигнала);`
2. Подключение сигнала от произвольного виджета к слоту произвольного виджета или объекта через вызов метода `QObject::connect(<Виджет — источник сигнала>, SIGNAL(<сигнал с аргументами>), <виджет или объект, к слоту которого нужно привязать сигнал>, SLOT(<имя слота с аргументами>), <способ подключения, по умолчанию AutoConnect>).`

Второй способ предполагает динамическое подключение и (или) отключение слота объекта от сигнала (**QObject::disconnect**). Если же в ходе работы для данного виджета не предполагается изменения связей слота и сигналов, то предпочтительней использовать первый способ подключения.

Таблицы стилей. Цветовая палитра

Таблицы стилей

В Qt есть мощный механизм настройки внешнего вида виджетов — таблица стилей, похожая на CSS-разметку в веб-программировании. Таблицы стилей — текстовые спецификации, которые могут быть установлены для всего приложения с помощью **QApplication::setStyleSheet()** или для определенного виджета (и его потомков) посредством **QWidget::setStyleSheet()**. Если на различных уровнях установлено несколько таблиц стилей, то Qt сводит их в одну. Это называется каскадированием. Например, следующая таблица стилей определяет, что все объекты **QLineEdit** должны использовать в качестве фона зеленый цвет, а все объекты **QCheckBox** должны использовать красный как цвет текста:

```
QLineEdit { background: green }
QCheckBox { color: red }
```

Установка таблицы стилей выполняется методом **setStyleSheet(QString)**:

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

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.resize(800, 600);
    w.setStyleSheet("QPushButton {font: bold 14 px; background-color: red;}
QMainWindow{background-color:grey}");
    w.show();

    return a.exec();
}
```

В данном случае таблица стилей будет применена только к виджетам, расположенным на данном виджете (в данном случае на виджете **w: MainWindow**). Чтобы стили применялись ко всем объектам во всех компонентах, нужно вызывать метод из объекта приложения **qApp**:

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

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.resize(800, 600);
    qApp->setStyleSheet("QPushButton {font: bold 14 px; background-color: red;}");
}
```

```

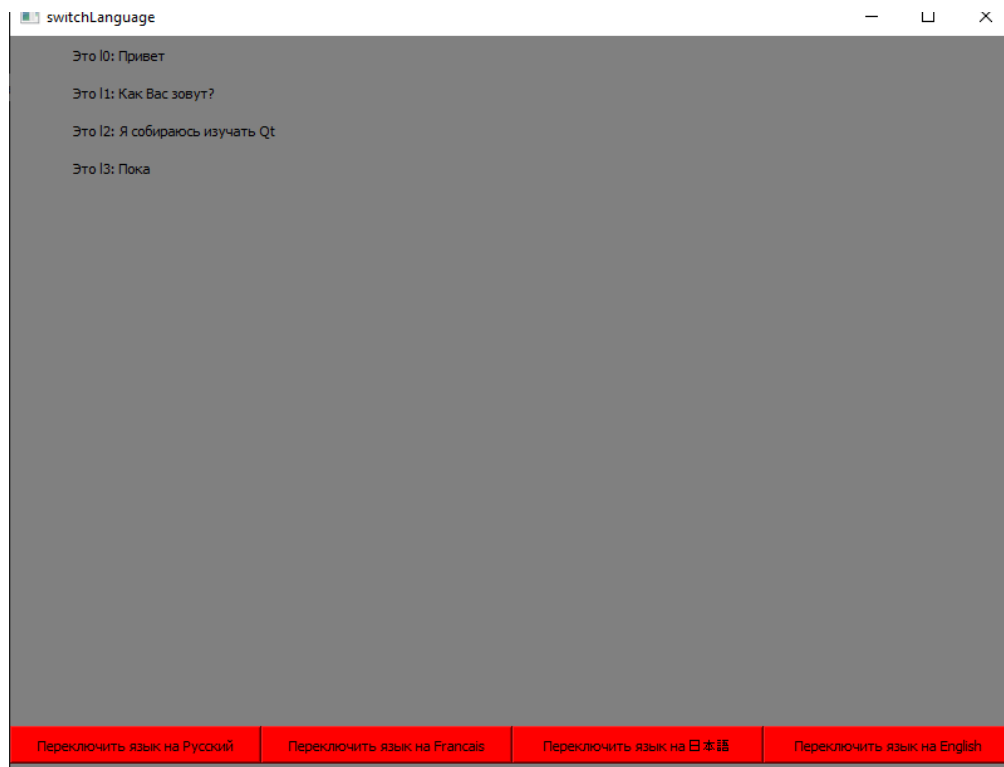
QMainWindow{background-color:grey}");

    w.show();

    return a.exec();
}

```

Для окна `w` мы в обоих случаях получим одинаковый результат:



Для наглядности рассмотрим два примера.

Код первого примера:

```

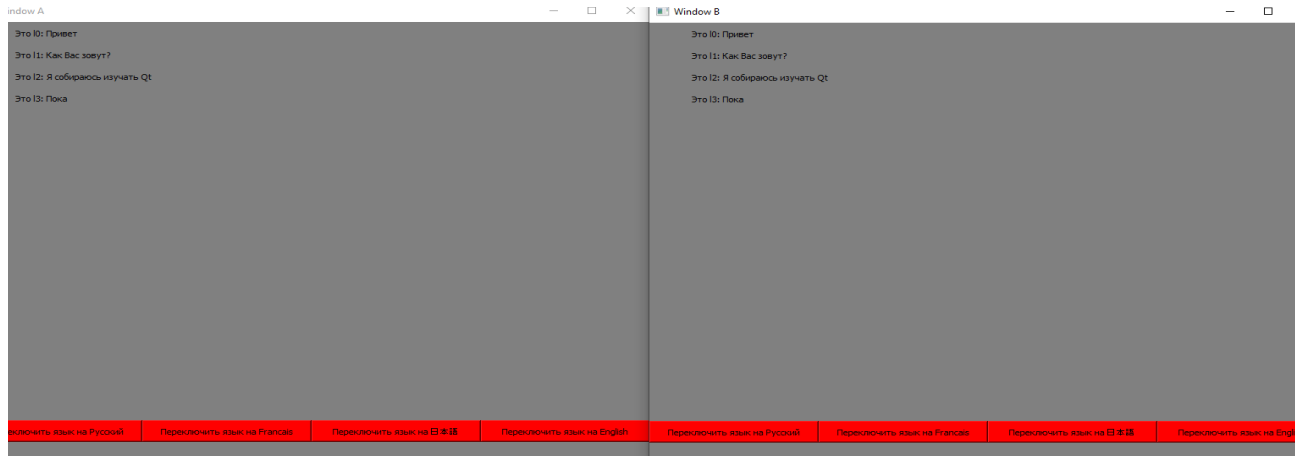
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.setWindowTitle("Window A");
    w.resize(800, 600);
    MainWindow w1;
    w1.setWindowTitle("Window B");
    w1.resize(800, 600);
    qApp->setStyleSheet("QPushButton {font: bold 14 px; background-color: red;}");
    QMainWindow{background-color:grey}");
    w.show();
}

```

```
w1.show();
return a.exec();
}
```

В данном примере было создано два окна (А и В), а таблица стилей применена к объекту приложения.



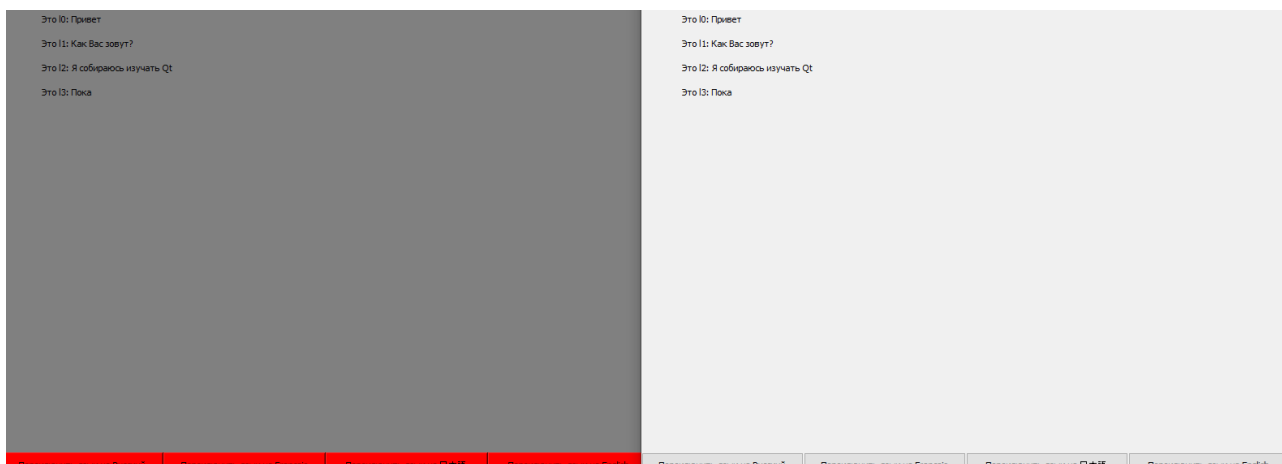
Получилось два одинаковых по стилю окна.

И второй пример:

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

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.setWindowTitle("Window A");
    w.resize(800, 600);
    MainWindow w1;
    w1.setWindowTitle("Window B");
    w1.resize(800, 600);
    w.setStyleSheet("QPushButton {font: bold 14 px; background-color: red;}
QMainWindow{background-color:grey}");
    w.show();
    w1.show();
    return a.exec();
}
```

Сейчас таблица стилей применена к окну А. В результате получаем:



Изменение стиля произошло только для окна A.

Цветовая палитра

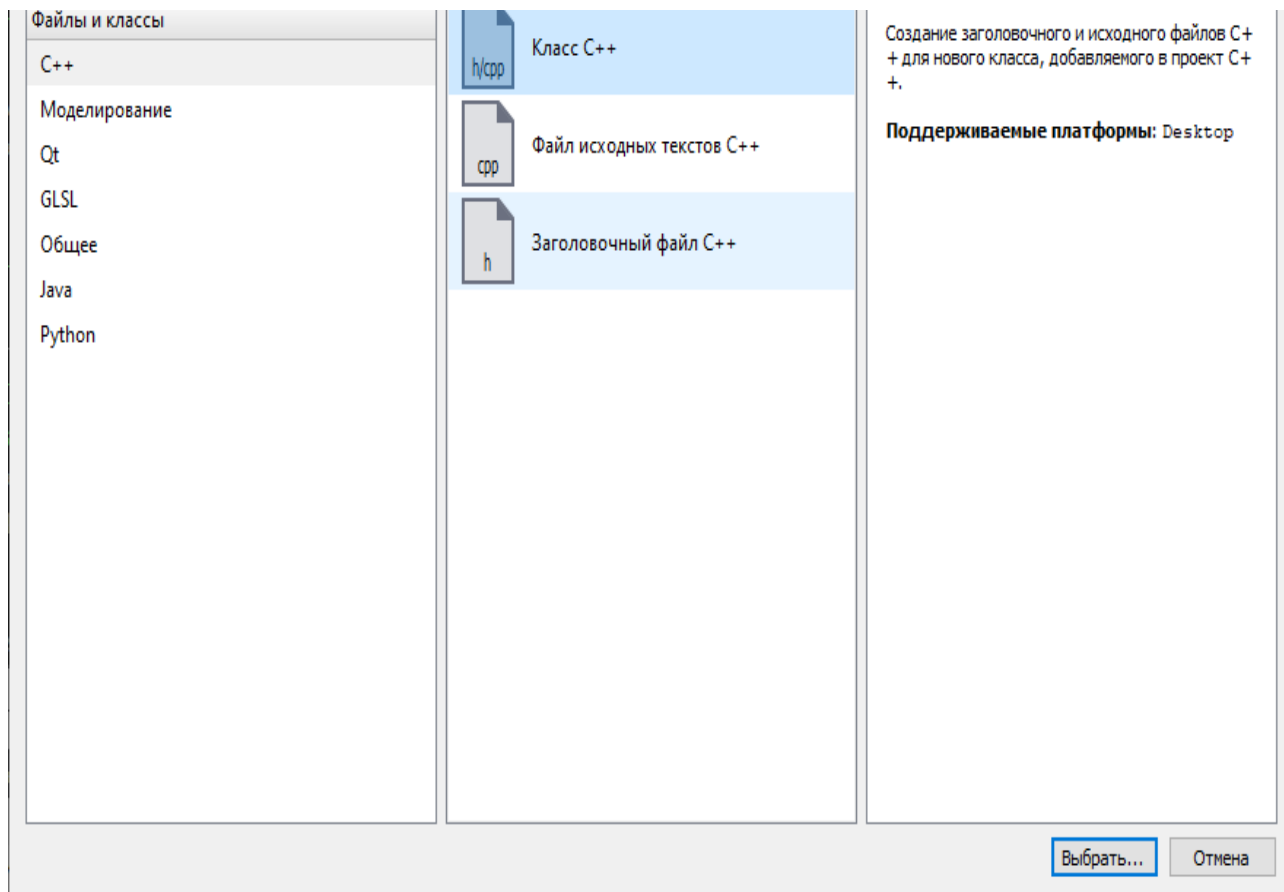
Палитра состоит из трех цветовых групп: Активный, Недоступный и Неактивный. Все виджеты в Qt содержат палитру и используют ее для отрисовки, поэтому пользовательский интерфейс легко программировать и настраивать. Цвет палитры задается тремя цветами: красным, зеленым и синим (в случае использования аппаратного ускорения используется четвертый параметр — альфа-канал). Основные классы для работы с палитрой — **QColor** и **QPalette**.

Конструктор **QColor** создает цвет на основе значений RGB. Чтобы создать **QColor** на основе значений HSV или CMYK, используйте функции **toHsv()** и **toCmyk()** соответственно. Эти функции возвращают копию цвета, используя нужный формат. Кроме того, статические функции **fromRgb()**, **fromHsv()** и **fromCmyk()** создают цвета из указанных значений. Цвет может быть преобразован в любой из трех форматов с помощью функции **convertTo()** (возвращающей копию цвета в нужном формате) или функций **setRgb()**, **setHsv()** и **setCmyk()**, изменяющих цветовой формат. Функция **spec()** сообщает, как был указан цвет. Цвет можно установить, передав строку RGB (например #ffffff — белый цвет, #000000 — черный) или строку ARGB (например #ff112233) или имя цвета (например blue) в функцию **setNamedColor()**. Названия цветов взяты из имен цветов SVG 1.0. Функция **name()** возвращает название цвета в формате #RRGGBB.

Цвета также могут быть установлены с помощью **setRgb()**, **setHsv()** и **setCmyk()**. Чтобы получить более светлый или более темный цвет, используется функции **lighter()** и **darker()** соответственно. Также объект цвета позволяет получить одну из составляющих цвета, например с помощью **red()** выдает значения красной составляющей, **hue()** возвращает текущий оттенок и **cyan()**. Значения всех цветов можно получить в одном объекте, используя функции **getRgb()**, **getHsv()** и **getCmyk()**. Например, используя цветовую модель RGB, к цветовым компонентам можно также получить доступ с помощью метода **rgb()**, который вернет объект **QRGB**.

Создание собственных виджетов

Помимо стандартных виджетов можно создать и собственный для специфических задач или нестандартного графического интерфейса программного обеспечения. Создание главного окна — это тоже создание собственного виджета. Создадим виджет для отображения списка файлов и каталогов на диске, который будет содержать кнопку сворачивания, область просмотра картинки и текста с описанием. Новый проект создадим без формы с базовым классом **QMainWindow**. Через контекстное меню проекта добавим новый класс C++:



Назовем новый класс **FirstMyQtWidget**:

C++

Подробнее

Итог

Определить класс

Имя класса:

FirstMyWidget

Базовый класс:

QWidget

☐ Подключить QObject

☒ Подключить QWidget

☐ Подключить QMainWindow

☐ Подключить QDeclarativeItem - Qt Quick 1

☐ Подключить QQuickItem - Qt Quick 2

☐ Подключить QSharedData

Заголовочный файл:

firstmywidget.h

Файл исходных текстов:

firstmywidget.cpp

Путь:

C:\Users\Vadiv\Documents\lessons Qt\Lesson5\ItMyWidget\itsmywidget

Обзор

Далее

Отмена

Подключаем заголовочные файлы к **QTreeView**, **QPushButton**, **QGridLayout** для компоновки, **QStandardItemModel** для заполнения **QTreeView**, **QApplication** для получения стандартных значков.

```
#ifndef FIRSTMYQTWIDGET_H
#define FIRSTMYQTWIDGET_H

#include <QWidget>
#include <QGridLayout>
#include <QTreeView>
#include <QComboBox>
#include <QPushButton>
#include <QStandardItemModel>
#include <QApplication>

Q_PROPERTY(QStandardItemModel *model READ getCurrentModel WRITE setNewModel)

class FirstMyQtWidget : public QWidget
{
    Q_OBJECT
public:
    explicit FirstMyQtWidget(QWidget *parent = nullptr);
    void clearTree();
    QStandardItemModel *getCurrentModel() const
    {
        return model;
    }
};
```

```

    }
    void setNewModel(QStandardItemModel *newmodel);
    void rebuildModel(QString str);
private:
    QGridLayout *gridLay;
    QTreeView *tree;
    QPushButton *mainPath;
    QComboBox *diskSelBox;
private slots:
    void chgDisk(int index); // получаем индекс выбранного диска
    void goMainPath();      // Для UNIX-подобных ОС верхним уровнем является
                            // путь /
private:
    QStandardItemModel *model;
    QString curretnPath;
protected:
};

#endif // FIRSTMYQTWIDGET_H

```

Компоновка виджетов на родительском виджете происходит с использованием слоев; в данном примере используется слой компоновки по сетке. Рассмотрим конструктор класса:

```

#include "firstmyqtwidget.h"
#include <QDir>

FirstMyQtWidget::FirstMyQtWidget(QWidget *parent) : QWidget(parent),
model(nullptr)
{
    gridLay = new QGridLayout(this);           // создаем слой для компоновки
    this->setLayout(gridLay);                  // устанавливаем слой на виджет
    tree = new QTreeView(this);

    gridLay->addWidget(tree, 1, 0, 10, 10);    // размещен на первой строке
                                                // с нулевого столбца
                                                // занимает 10 условных строк
                                                // и столбцов
    setMinimumSize(500, 600);                  // ограничиваем размер виджета

    if(QSysInfo::productType() == "windows")
    {
        diskSelBox = new QComboBox(this);
        QFileInfoList list = QDir::drives();
        QFileInfoList::const_iterator listdisk = list.begin();
        int amount = list.count();
        for (int i = 0; i < amount; i++)
        {
            diskSelBox->addItem(listdisk->path());
            listdisk++;
        }
    }
}

```

```

        if (amount > 0)
        {
            rebuildModel(list.at(0).path());
        }
        gridLay->addWidget(diskSelBox, 0, 0, 1, 2); // координаты
        connect(diskSelBox, SIGNAL(activated(int)), this, SLOT(chgDisk(int)));
    } else {
        mainPath = new QPushButton(this);
        mainPath->setText("/");
        gridLay->addWidget(mainPath, 0, 0, 1, 2);
        connect(mainPath, SIGNAL(clicked()), this, SLOT(goMainPath()));
        rebuildModel("/");
    }
}

```

Первым делом создаем слой и устанавливаем его слоем создаваемого виджета. Создаем виджет **tree**. Устанавливаем этот виджет на созданный слой. Первый параметр — наш виджет, вторым является номер строки (от 0), далее номер столбца. Последними двумя параметрами указаны условное количество занимаемых строк и столбцов (**rowspan**, **colspan** в терминах веб-программирования). При создании кнопки и списка выбора диска указано, что компоненты занимают одну строку, а дерево — десять строк, то есть слой условно делится на одиннадцать строк. Теперь при изменении масштаба будут соблюдаться пропорции согласно объявленному количеству строк и столбцов на слое. Определение дисков с помощью метода **QDir::drives()** актуально только под ОС Windows, в Unix-подобных ОС нет такого понятия, как диск, поэтому данный метод игнорируется. Следовательно, если используется Windows, создаем список, а для других ОС — кнопку для перехода в верхний каталог «/». После чего перестраиваем модель отображения дерева:

```

void FirstMyQtWidget::chgDisk(int index)
{
    QFileInfoList list = QDir::drives();
    rebuildModel(list.at(index).path());
}

void FirstMyQtWidget::goMainPath()
{
    rebuildModel("/");
}

void FirstMyQtWidget::setNewModel(QStandardItemModel *newmodel)
{
    tree->setModel(newmodel);
    model = newmodel;
}

```

Этот индекс используем для получения имени диска, а для кнопки устанавливаем верхний уровень.

Теперь построим модель. В [предыдущих уроках](#) уже рассматривался принцип создания моделей для дерева.

```
void FirstMyQtWidget::rebuildModel(QString str)
{
    curretnPath = str;
    QStandardItemModel *model = new QStandardItemModel(this);
    QList<QStandardItem*> items;
    items.append(new
QStandardItem(QIcon(QApplication::style()->standardIcon(QStyle::SP_DriveHDIcon))
, str));
    model->appendRow(items);

    QDir dir(str);
    dir.setFilter(QDir::Hidden | QDir::NoSymLinks | QDir::Dirs);
    QStringList list = dir.entryList();
    int amount = list.count();
    QList<QStandardItem*> folders;
    for (int i = 0; i < amount; i++)
    {
        QStandardItem* f = new
QStandardItem(QIcon(QApplication::style()->standardIcon(QStyle::SP_DirIcon)),
list.at(i));
        folders.append(f);
    }

    items.at(0)->appendRows(folders);

    dir.setFilter(QDir::Hidden | QDir::NoSymLinks | QDir::Files); amount =
list.count();
    QList<QStandardItem*> files;
    for (int i = 0; i < amount; i++)
    {
        QStandardItem* f = new
QStandardItem(QIcon(QApplication::style()->standardIcon(QStyle::SP_FileIcon)),
list.at(i));
        files.append(f);
    }

    items.at(0)->appendRows(files);
    setNewModel(model);
}
```

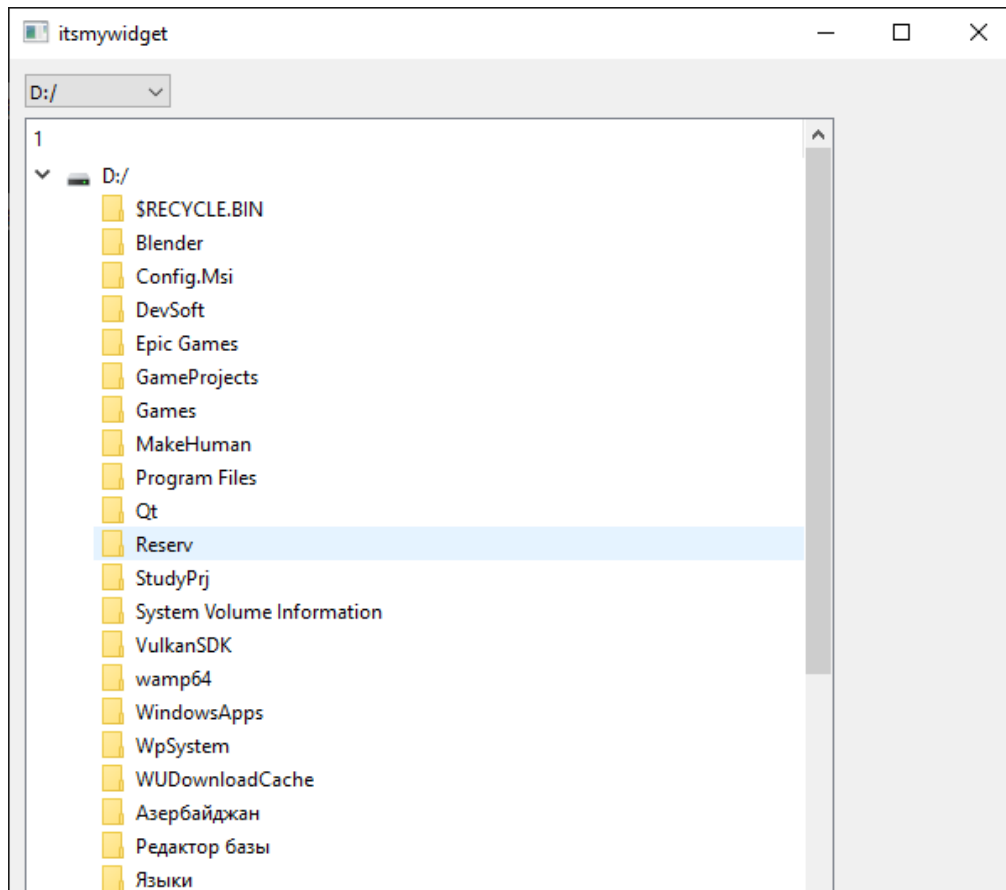
Добавим виджет на главное окно:

```
MainWindow::MainWindow(QWidget *parent)
: QMainWindow(parent)
```

```
{  
    fmqw = new FirstMyQWidget(this);  
}
```

Не забываем подключить заголовочный файл и добавить переменную ссылки в объявление класса.

Запустим проект:



В данной программе можно модифицировать объект **QStandardItem** так, чтобы вместо редактирования текущего значения передавалось имя каталога (например, при перехвате событий), поэтому данный виджет может стать удобной основой для файлового менеджера.

Практическое задание

1. Добавить в текстовый редактор выбор из нескольких таблиц стилей. Сделать темную и светлую тему.
2. Перевести текстовый редактор на английский язык (названия кнопок из предыдущих заданий). Добавить возможность динамически переключать языки во время работы приложения.
3. Написать собственный виджет - просмотрщик файловой системы. Добавить строку навигации, в которой выводится текущий каталог.

Дополнительные материалы

1. <https://doc.qt.io/qt-5/third-party-libraries.html>.
2. <http://doc.crossplatform.ru/qt/4.5.0/stylesheet.html>.
3. https://www.opennet.ru/docs/RUS/qt3_prog/c2572.html.

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. <http://doc.crossplatform.ru/qt/4.5.0/stylesheet.html>.
2. <https://doc.qt.io/qt-5/stylesheet-reference.html>.
3. <https://doc.qt.io/qt-5/qglwidget.html>.