ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Qt – это кросс-платформенный инструментарий разработки ПО на языке программирования C++, состоящий из библиотеки классов и набора специализированных инструментальных средств.

Что делает Qt привлекательным для разработчиков программного обеспечения?

Прежде всего, это кросс-платформенность Qt, а это значит ваша программа, написанная с использованием только средств Qt, будет переносима на уровне исходного кода на все платформы, поддерживаемые Qt. Это большинство Unix платформ с оконной системой X-Windows, некоторые встраиваемые операционные системы на базе Linux-ядра без оконных систем (framebuffer), а также для мобильных платформ на базе ядра Linux, в том числе и Android, операционные системы Mac Os фирмы Apple и, естественно, большинство версий Windows фирмы Microsoft.

Для создания приложений с применением Qt можно использовать следующие компиляторы (инструментальные средства):

– Windows – различные версии Visual Studio или GNU C++;

– Mac OS X – инструментальную среду XCODE;

– Linux – GNU C++;

– для других операционных систем – это чаще всего компилятор GNU C++.

Библиотека классов Qt используется для создания множества программных продуктов, среди них – *Skype*, *Google Earth*, *Яндекс Онлайн*, *Opera*, *Adobe Photoshop Album*, *VLC Media Player* и другие.

Несмотря на то, что Qt имеет репутацию средства для разработки кросс-платформенных приложений, благодаря своему интуитивному и мощному программному интерфейсу, Qt используется не только для создания кросс-платформенных приложений.

В учебном пособии на простых примерах и задачах показано, как создавать приложения с использованием библиотеки Qt. Упор сделан именно на практику, а не на теорию, хотя и она тоже присутствует в достаточно удобном для восприятия виде. Теоретическим основам и посвящен первый раздел лабораторных работ. Там, где это необходимо, теория присутствует и во второй части, которая и представляет собой детально разобранные примеры программ и практические основы написания приложений с использованием Qt.

Данное пособие рассчитано в первую очередь на студентов III–V курсов, изучающих кросс-платформенное программирование, но может быть полезно также тем, кто хочет изучить программирование с использованием Qt. Все, что от вас потребуется, – это знание синтаксиса языка C++, да и, желательно, наличие опыта программирования.

Стоит заметить один важный аспект данного пособия: работа с Qt будет рассмотрена в основном под управлением операционных систем (ОС) семейства Windows.

1 ОСНОВЫ РАБОТЫ С QT

Цель работы *– заложить основы навыков создания и проектирования кросс-платформенных программных приложений посредством Qt.*

Для начала вам следует ознакомиться с процессом работы над данным лабораторным практикумом:
1. Вы читаете раздел «Теоретическое введение» к лабораторной работе.
2. Вы читаете и последовательно выполняете все пункты практической части.
3. Вы самостоятельно выполняете практические задания, которые встречаются в тексте лабораторной работы.
4. Вы тестируете разработанное вами программное обеспечение (ПО), разбираетесь в написанном коде, отвечаете на контрольные и тестовые вопросы.
5. Вы сдаете лабораторную работу преподавателю.
Своевременная сдача предполагает наличие работоспособного ПО, выполнение всех практических самостоятельных заданий, понимание написанного кода, а также способность ответить на каверзные вопросы преподавателя.

Теоретическое введение

В этом разделе будет даваться информация об основных характерных особенностях данной среды разработки, с которыми вы встретитесь в процессе разработки.

В целом Qt не имеет кардинальных различий с другими средами визуальной разработки ПО.

Qt позволяет создавать приложения для следующих ОС:

- Linux/X11;
- Mac OS X;
- Windows;

а также для ОС смартфонов и коммуникаторов:

- Windows Mobile и Windows CE;
- iOS;
- Symbian, Maemo.

Windows Mobile и **Windows CE –** компактная ОС компании Microsoft, выпускается с 1996 года и занимает крупный сегмент рынка ОС для смартфонов.

iOS (до 2010 года – iPhone OS) **–** мобильная операционная система, разрабатываемая и выпускаемая американской компанией Apple. Была выпущена в 2007 году; первоначально – для iPhone и iPod touch, позже – для таких устройств, как iPad и Apple TV.iOS разработана на основе Mac OS X и использует тот же POSIX-совместимый набор основных компонентов Darwin.

Symbian OS – это операционная система для смартфонов и коммуникаторов, разрабатываемая консорциумом Symbian, основанным в июне 1998 года компаниями: Psion, Nokia, Ericsson и Motorola. Позже к консорциуму присоединились компании: Sony Ericsson, Siemens, Panasonic, Fujitsu, Samsung, Sony, Sharp и Sanyo.

Maemo – базирующаяся на Debian Linux платформа для портативных устройств. Используется в интернет-планшетах Nokia 770, N800, N810 и N900.

Также есть версия Qt на **Android**.

Виджеты

Для дальнейшей работы вам также необходимо знать, что же такое виджет, т.к. с этим словом вы часто будете сталкиваться в дальнейшем.

По терминологии Qt и Unix ***виджетом*** *(widget) называется любой визуальный элемент графического интерфейса пользователя*. Этот термин происходит от «window gadget» и соответствует элементу управления («control») и контейнеру («container») по терминологии Windows. Кнопки, меню, полосы прокрутки и фреймы являются примерами виджетов. Одни виджеты могут содержать в себе другие виджеты. Например, окно приложения обычно является виджетом, содержащим QMenuBar (панель меню), несколько QToolBar (панель инструментов), QStatusBar (строка состояния) и некоторые другие виджеты. Большинство приложений используют QMainWindow или QDialog в качестве окна приложения, однако Qt настолько гибок, что любой виджет может быть окном.

Виджеты всегда создаются сначала невидимыми, и поэтому до непосредственного вывода на экран вы можете настроить их.

По данному курсу рекомендованы к прочтению следующие книги:

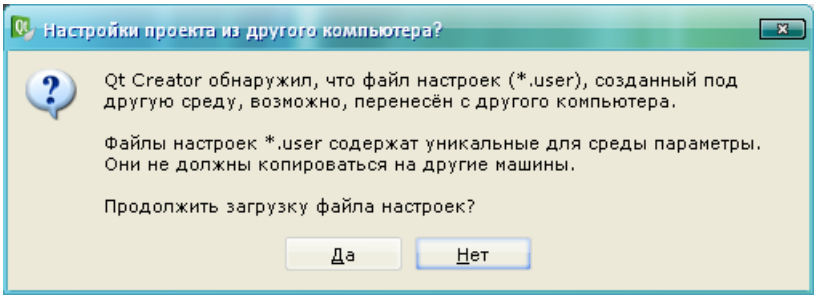
1. Бланшет, Ж. Qt4 : Программирование GUI на C++ [Текст] / Ж. Бланшет, М. Саммерфилд. – Издание второе, дополненное. – Москва: КУДИЦ-ПРЕСС, 2008. – 718 с.

2. Шлее, М. Qt 4.5. Профессиональное программирование на C++ [Текст] / М. Шлее. – Санкт-Петербург: БХВ, 2010. – 896 с.

3. Саммерфилд, М. Qt. Профессиональное программирование. Разработка кроссплатформенных приложений на C++ [Текст] / М. Саммерфилд. – Пер. с англ. – СПб.: Символ-Плюс, 2011. – 560 с.

4. Земсков, Ю. В. Программирование на C++ с использованием библиотеки Qt 4 [Текст] / Ю. В. Земсков. – БХВ-Петербург, 2007. – 357 с.

Это важно: *если вы когда-нибудь увидите на экране нечто подобное тому, что на рисунке, это значит, что вы, скорее всего, ранее работали над проектом на другом компьютере и сейчас вам следует ответить «Нет», иначе программа компилироваться не будет.*



1.1 Hello, World!

Начнем с простейшего приложения, известного всем – «Hello, World!».

Последующее описание действий излишне детализировано, впрочем, это только лишь для первого примера – в дальнейшем не будет приводиться столько экранных форм. Сейчас же они служат для большей наглядности и понимания программистом своих действий.

Также стоит заметить, что большинство из тех действий, что вы будете сейчас совершать, полностью автоматизированы, и их можно сделать всего за пару кликов, воспользовавшись мастером, но в целях обучения мы будем их совершать последовательно вручную.

Для начала запустите *Qt Creator*.

1. Выберите *Файл → Новый файл или проект…*

Либо просто воспользуйтесь комбинацией *Ctrl+N*. Выберите *Другой проект* и затем из списка: *Пустой проект Qt* и нажмите кнопку *Выбрать…* (рис. 1.1).

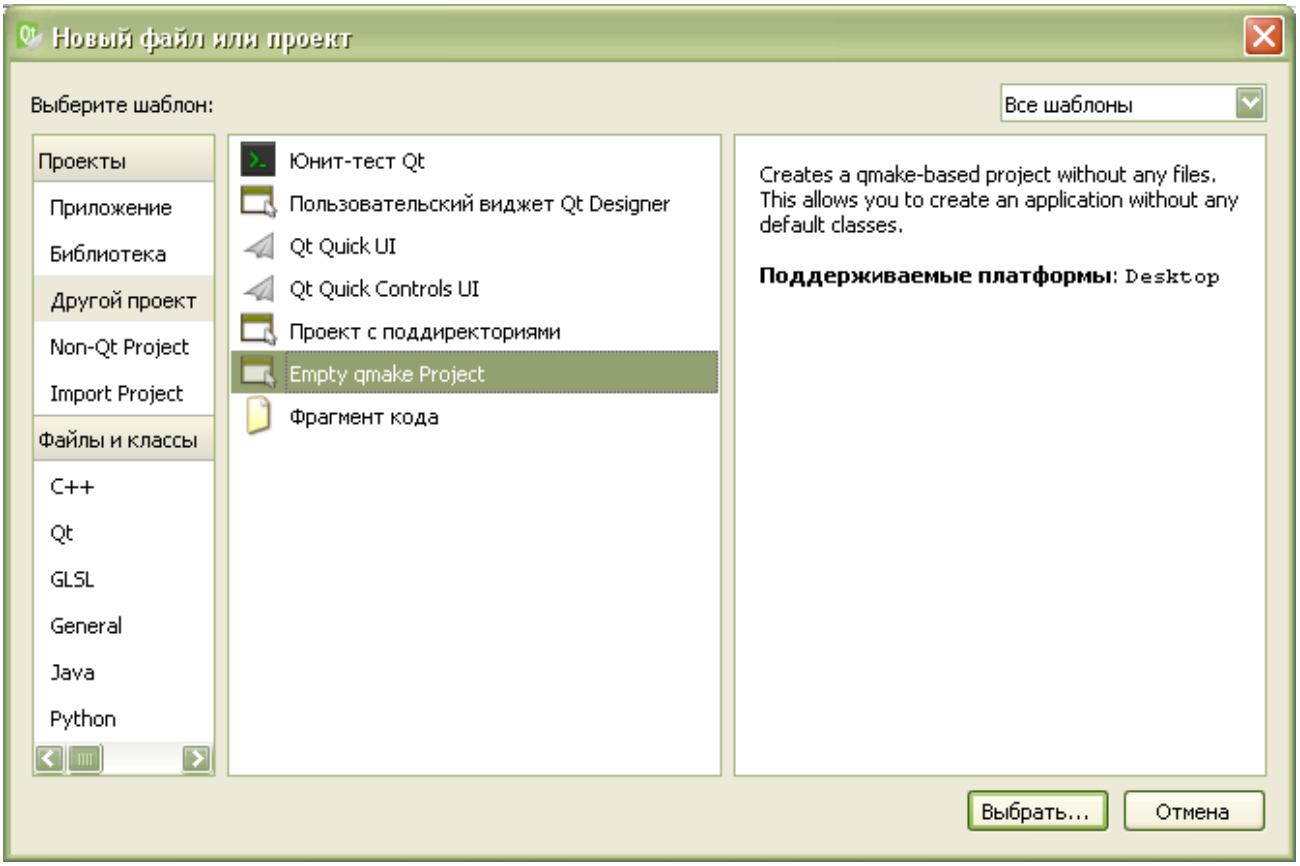


Рисунок 1.1 – Создание пустого проекта Qt

Это важно: *прописывая путь к директории проекта, избегайте символов кириллицы, иначе проект не будет компилироваться.*

2. Введите имя проекта и путь к нему (избегайте символов кириллицы!!!). Когда будете задавать путь, выделите для ваших проектов отдельную директорию

(рис. 1.2), а также поставьте галочку *Размещение проекта по умолчанию* (теперь все ваши проекты по умолчанию будут размещаться в этом каталоге), нажмите *Далее* (2 раза), а потом – *Завершить*.

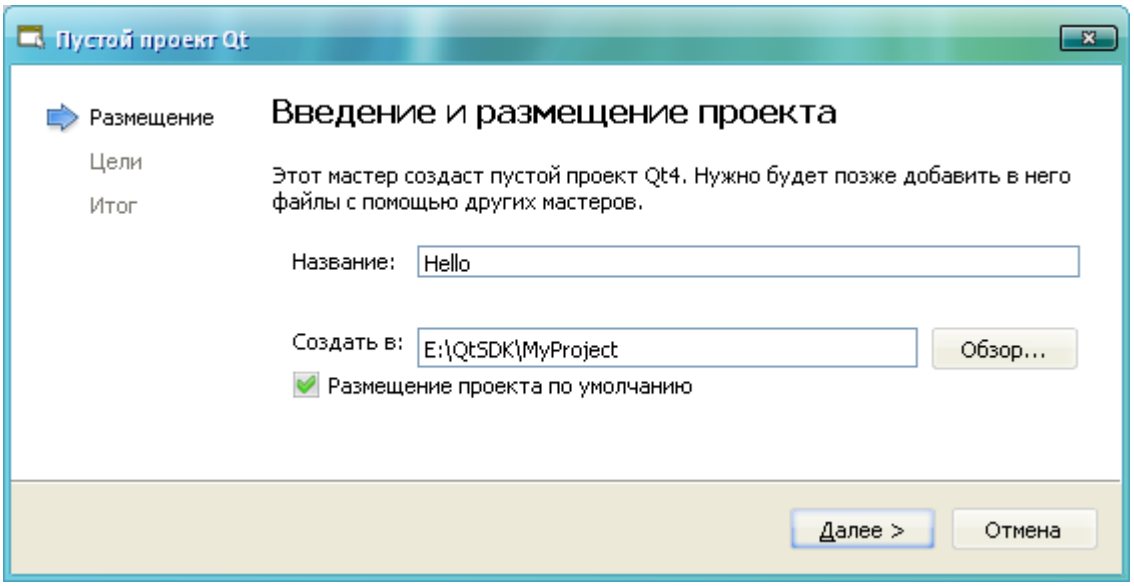


Рисунок 1.2 – Создание пустого проекта Qt

3. Теперь повторите пункты 1–2, только выберите *C++*, а затем *Файл исходных текстов C++* (рис. 1.3).

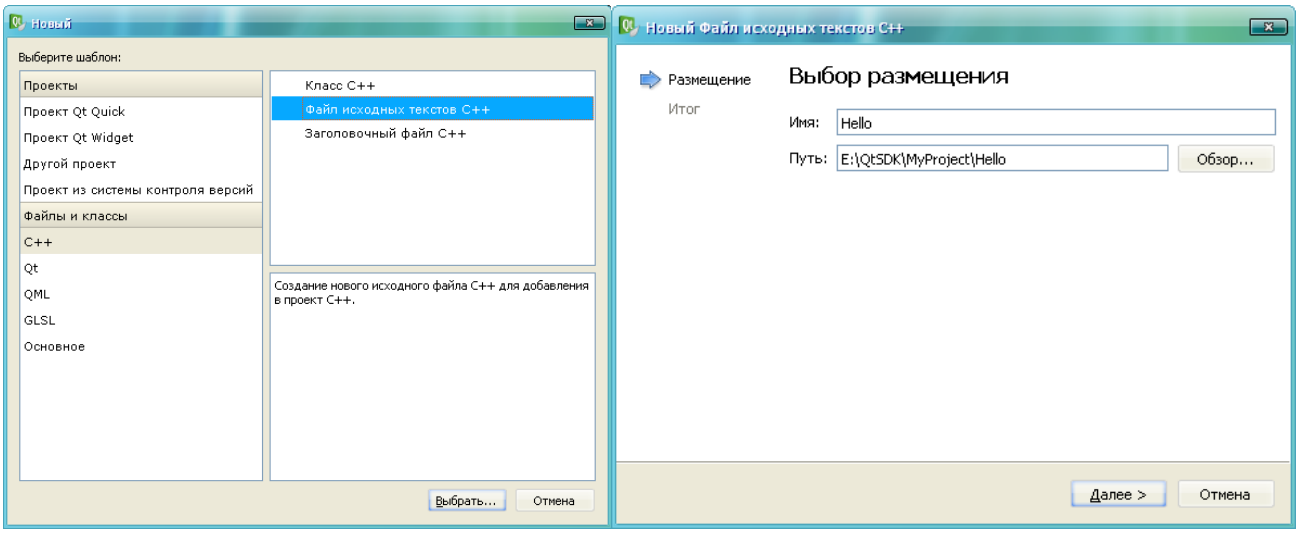


Рисунок 1.3 – Добавление в проект файла исходных текстов C++ с помощью мастера

Это важно: *начиная с версии Qt 5.0 в файл проекта (.pro) стоит дописать:*

```
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
```

Это важно: *если вы будете переносить проект (или сохранять на флеш-носитель), то вам необходимы лишь файлы с расширением: pro, h, cpp, ui (если вы создавали форму графически).*

4. Введите следующий код (в файле hello.cpp):

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

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

    QPushButton button("Hello, World!");
    button.resize(200, 60);
    button.show( );

    return app.exec( );
}
```

5. Нажмите на кнопку *Выполнить*  либо Ctrl+R (рис. 1.4).

Если программа не запустилась – прочитайте, что пишет компилятор, проверьте путь к исполняемому файлу (он не должен содержать символов кириллицы), посмотрите, не запущена ли программа прямо сейчас…

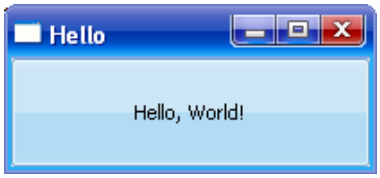


Рисунок 1.4 – Результат работы программы

Пояснения к программе:

– (1–2) подключили заголовочные файлы с определениями классов *QApplication* (приложение) и *QPushButton* (кнопка);

– (4) обычный для C++ заголовок главной функции *main* с аргументами командной строки;

– (6) объявили переменную типа *QApplication* (приложение), передав конструктору параметры командной строки, которые, возможно, указаны при запуске программы (*argc* – число параметров, *argv* – указатель на массив строковых значений);

– (8) создали главное окно приложения, которое представляет собой обычную кнопку с текстом «*Hello, World!*» (пока мы избегаем использовать символы кириллицы, этому вопросу будут посвящены следующие примеры);

– (9) определили размеры окна (ширину и высоту) в пикселях;

– (10) вывели окно на экран;

– (12) запустили цикл обработки событий, происходящих с элементами приложения. Пока в нашей программе никакие события не определены, кроме стандартных реакций на действия пользователя (изменение размеров и положения окна, нажатие кнопок в строке заголовка).

6. Прежде чем перейти к следующему примеру, позволим себе небольшое отступление, а именно, заменим строки:

```
    QPushButton button("Hello, World!");
```

```
    button.resize(200, 60);
    button.show( );
```

на строки:

```
    QLabel *label = new QLabel ("<h2><i>Hello,</i> " "<font color = red > Qt!
</font> </h2>");
    label->show( );
```

заметьте, теперь мы используем указатели, а могли написать просто:

```
    QLabel label("<h2><i>Hello,</i> " "<font color = red > Qt! </font></h2>");
    label.show( );
```

вставьте тот вариант, который вам больше нравится.

Также стоит добавить в начало файла:

```
    #include <QLabel>
```

и снова выполним построение приложения. При запуске окно будет выглядеть, как показано на рис. 1.5. Как иллюстрирует этот пример, совсем не трудно выделять элементы пользовательского интерфейса Qt-приложения с применением некоторых простых средств форматирования документов HTML.

Рисунок 1.5 – Результат работы программы
с HTML-форматированием

7. Продолжим модифицировать эту несложную программу и постепенно осваивать Qt. Откатим все изменения, сделанные нами в п. 6, переделаем программу под указатели и допишем следующие строки:

```
    QPushButton *button = new QPushButton("Quit");
    QObject::connect (button, SIGNAL(clicked( )), &app, SLOT(quit( )));
    button->resize(200, 60);
    button->show( );
```

Эти строки привязывают действие выхода из программы к нашей кнопке, используя механизм сигналов и слотов.

Виджеты Qt (о том, что такое виджет, см. стр. 6) генерируют сигналы в ответ на выполнение пользователем какого-то действия или изменение состояния. Например, QPushButton генерируют сигнал clicked() при нажатии пользователем кнопки. Сигнал может быть связан с функцией (называемой слотом в данном контексте) для автоматического ее выполнения при получении данного сигнала. Сигнал может быть связан с любым количеством слотов. В нашем примере мы связываем сигнал кнопки clicked() со слотом quit() объекта приложения QApplication. Макросы SIGNAL() и SLOT() являются частью синтаксиса.

8. Добавим «горячую клавишу». Если текст подписи содержит амперсанд “&”, то символ после амперсанда автоматически устанавливается в качестве «горячей клавиши», переопределяя ранее заданное значение. Вы можете определить собственный акселератор для кнопки с помощью метода setShortcut(const QKeySequence &). Метод и свойство *shortcut* возвращают значение последовательности «горячих клавиш» для данной кнопки.

Чтобы получить доступ к методу, следует напечатать имя экземпляра класса (например, button), затем поставить точку (или «->») и выбрать из выпадающего списка нужный метод.

Коды QKeySequence (советую посмотреть файл справки по этому классу) могут быть заданы как числовыми значениями символов, так и строками специального формата, которые могут восприниматься в Qt как допустимые. Например, “Ctrl+Q” является корректным сочетанием в качестве комбинации “горячих клавиш”. Но Qt не желает напрямую воспринимать текстовые константы – необходимо заключить их в функцию tr().

Впрочем, можно поступить и по-другому – использовать константные выражения, например: Qt::Key_Q – для клавиши Q.

Это интересно: *в Qt клавише Enter соответствует константа Qt::Key_Return, а не Qt::Key_Enter.*

Это интересно: *для клавиши Ctrl константой будет Qt::CTRL.*

Это важно: *tr следует применять в данном случае таким образом – QObject::tr(<Строковые символы в кавычках>).*

9. Есть еще один интересный метод, связанный с кнопкой: animateClick(). Этот метод анимирует щелчок на кнопке: она будет отрисована в “нажатом”, а через указанное время (в миллисекундах) – в “отжатом” состоянии. Такой эффект часто реализуется в демонстрационных целях, когда нужно эмулировать работу пользователя. Опробуйте этот метод в действии.

Это интересно: *1 секунда равняется 1000 миллисекундам.*

10. Специфика кнопок действия (QPushButton) состоит в следующем: одна из них может быть кнопкой по умолчанию, т.е. генерировать сигнал активизации во время нажатия клавиши <Enter> на форме. Обычно такое поведение используется в окнах диалога. Для установки/проверки свойства “кнопка по умолчанию” используются методы QPushButton::setDefault() и isDefault() соответственно.

Самостоятельная работа

В качестве практического задания реализуйте:

– анимацию трехсекундного нажатия на клавишу;

– выход из программы по нажатию Ctrl+Q и Enter.

Задания взаимоисключающие, посему реализуйте сначала одно, затем закомментируйте его и приступайте к другому.

11. Давайте научим программу здороваться с нами.

До этого мы уже использовали QLabel и QPushButton в качестве главного виджета, обычно же используют QMainWindow или QDialog, однако на сей раз главным виджетом будет (как бы это странно не звучало) виджет.

В примере ниже стоит обратить особое внимание на взаимодействие spinBox и slider; подумайте, насколько это сложно реализовать в других средах.

```
#include <QApplication>
#include <QPushButton>
#include <QHBoxLayout>
#include <QSlider>
#include <QSpinbox>
#include <QLabel>
#include <QLineEdit>
int main(int argc, char *argv[ ])
{
QApplication app(argc, argv);
QWidget *window = new QWidget;
window->setWindowTitle("Enter Your Name and Age");

QLabel *label = new QLabel("<font color = red > Name </font>");
QLineEdit *lineEdit = new QLineEdit;
QLabel *label2 = new QLabel("<font color = yellow > Age </font>");
spinBox->setValue(35);
QLabel *label3 = new QLabel("<font color = red > Greetings, my Overlord!
</font>");
QPushButton *button = new QPushButton("Hello!");

QSpinBox *spinBox = new QSpinBox;
QSlider *slider = new QSlider(Qt::Horizontal);
spinBox->setRange(0, 130);
slider->setRange(0, 130);

QObject::connect(spinBox, SIGNAL(valueChanged(int)),slider,SLOT(setValue(int)));
QObject::connect(slider, SIGNAL(valueChanged(int)),spinBox, SLOT(setValue(int)));

QHBoxLayout *topLayout = new QHBoxLayout;
topLayout->addWidget(label);
topLayout->addWidget(lineEdit);
QHBoxLayout *centralLayout = new QHBoxLayout;
centralLayout->addWidget(label2);
centralLayout->addWidget(spinBox);
centralLayout->addWidget(slider);
QHBoxLayout *downLayout = new QHBoxLayout;
downLayout->addWidget(button);
QHBoxLayout *outLayout = new QHBoxLayout;
outLayout->addWidget(label3);
QVBoxLayout *Layout = new QVBoxLayout;
Layout->addLayout(topLayout);
Layout->addLayout(centralLayout);
Layout->addLayout(downLayout);
Layout->addLayout(outLayout);
window->setLayout(Layout);
window->resize(400,100);
window->show( );
return app.exec( );
}
```

Самостоятельная работа

Разберите код, описанный выше, постарайтесь понять, что и как реализовано. Обратите внимание на то, как реализована компоновка виджетов в окне приложения, а также на названия подключаемых директивой `#include` модулей.

Также обратите свое внимание на древо классов в Qt, которое представлено в приложении к данному учебному пособию.

В качестве **контрольного задания** сделайте так, чтобы:

– код компилировался без ошибок (в этой программе преднамеренно допущена ошибка);

– приложение поздоровалось именно с вами, т.е. вывело на экран то, что вы напишете в поле lineedit (вывод стоит осуществить в заголовок окна и в label3);

– измените компоновку виджетов в окне, будьте готовы рассказать преподавателю, что такое слои (Layout) и как они работают;

– ответьте на контрольные и тестовые вопросы.

Контрольные вопросы

1. Что собой представляет Qt?
2. Что такое «виджет»?
3. В чем состоит смысл понятия «кросс-платформенность»?
4. На каких ОС могут компилироваться программы, написанные с помощью Qt?
5. Какие средства форматирования доступны в Qt?
6. Как работает механизм сигналов и слотов в Qt?
7. Как работает механизм компоновки виджетов в Qt?

Тест

1. Какой базовый класс для всех кнопок?

– QRadioButton;
– QAbstractButton;
– QButton;
– QAllButton.

2. Укажите главное различие между сигналами и событиями:

– события помещаются в очередь перед обработкой, а сигналы нет;
– события – это более устаревшая технология, чем слоты и сигналы;
– методы обработки событий можно перегружать, а слоты нельзя;
– события могут обрабатываться лишь одним методом, а сигналы многими слотами.

3. Что произойдет с программой, если при соединении сигнала со слотом сигнатура слота будет указана неверно (возможны несколько вариантов ответа)?

– при выполнении не будет вызван необходимый слот;
– произойдет ошибка выполнения;
– программа не скомпилируется;
– программа выполнится после успешной компиляции.

4. Какой механизм используется в Qt для коммуникации между объектами?

– механизм вызова callback-функций;
– механизм слушателей;
– механизм сигналов и слотов.

2 СОЗДАНИЕ ГРАФИЧЕСКОГО ИНТЕРФЕЙСА С ПОМОЩЬЮ БИБЛИОТЕКИ КЛАССОВ QT

Цель работы – разработка интерфейса программного продукта с помощью Qt Designer.

Теоретическое введение

Программа *Qt Designer* – это средство быстрой разработки приложений (Rapid Application Development, RAD). Прежде всего, этот инструмент предназначен для дизайнеров, принцип его работы отвечает принципу WYSIGWYG (What You See Is What You Get, "что видишь, то и получаешь"). Он предоставляет возможность быстро создавать прототипы приложений, которые базируются на диалоговых окнах, а также могут иметь главное окно, меню, строку состояния и панель инструментов.

***Это важно**: в симбиозе с Qt Designer работает компилятор пользовательского интерфейса (User Interface Compiler, uic), читает файл описания пользовательского интерфейса в формате XML (.ui), сгенерированный Qt Designer, и создает соответствующий заголовочный файл C++.*

Средства разработки Qt спроектированы таким образом, чтобы было приятно программировать «вручную» и чтобы этот процесс был интуитивно понятен; и нет ничего необычного в разработке всего приложения Qt на «чистом» языке C++. Все же многие программисты предпочитают применять визуальные средства проектирования форм, поскольку этот метод представляется более естественным и позволяет получать конечный результат быстрее, чем при программировании «вручную», и такой подход дает возможность программистам быстрее и легче экспериментировать и изменять дизайн.

Qt Designer расширяет возможности программистов, предоставляя визуальные средства проектирования. Qt Designer может использоваться для разработки всех или только некоторых форм приложения. Формы, созданные с помощью Qt Designer, в конце концов представляются в виде программного кода на C++, поэтому Qt Designer может быть использован совместно с обычными средствами разработки, и он не налагает никаких специальных требований на компилятор.

Существует два основных способа работы с этой программой – автономный и через *Qt Creator*.

В этой лабораторной работе мы опишем автономный способ.

***Примечание:** иногда установщик не делает отдельной плашки для запуска Qt Designer через Пуск, посему его предстоит искать самостоятельно где-то в «<Путь установки Qt>\<№ версии Qt>\mingw492_32\bin\designer.exe».*

***Это важно:** процесс работы с Qt Designer описан подробнее в книге «Qt 4 программирование GUI на C++» (см. стр. 22–38).*

2.1 Основы работы с Qt Designer

1. Запустите *Qt Designer*. Перед вами появится вот такой диалог (рис. 2.1):

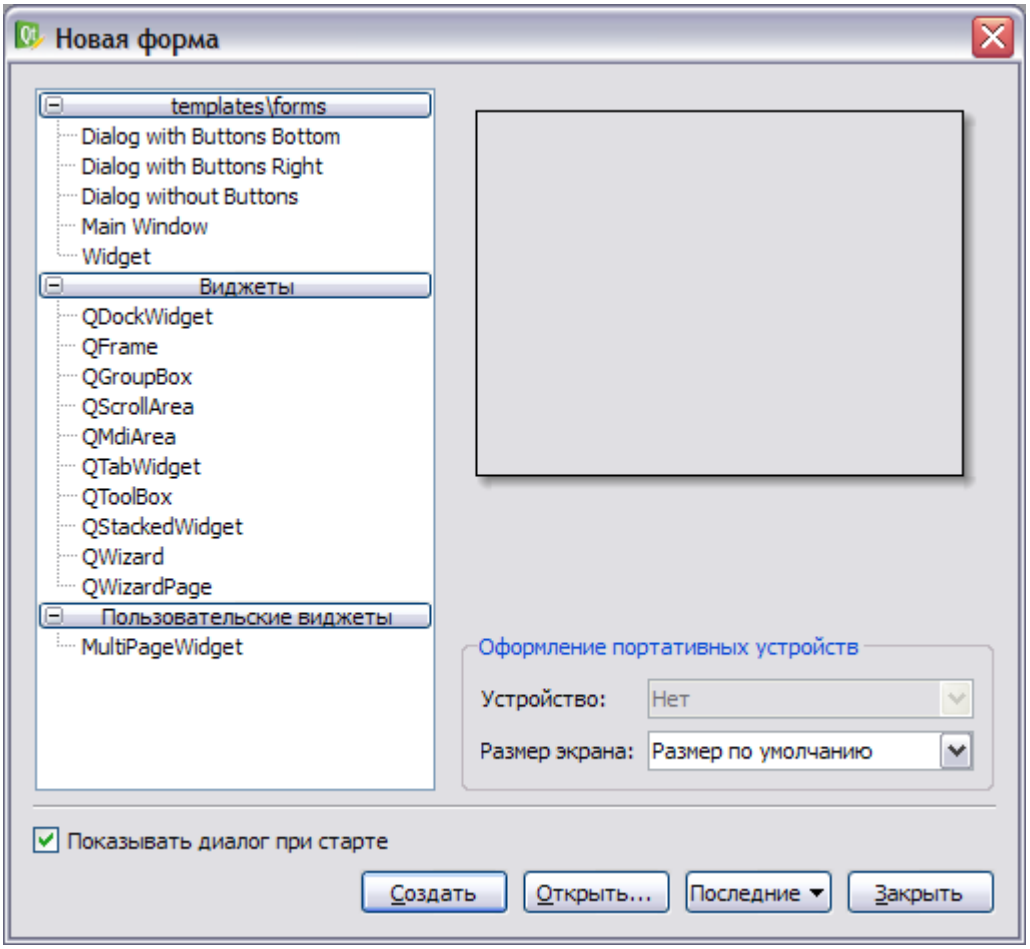


Рисунок 2.1 – Стартовое окно

Здесь все интуитивно понятно – вы можете создать диалог с кнопками (снизу, сбоку) или же без них, обычное главное окно и виджет (если забыли, что это такое, посмотрите теоретическое введение к предыдущей лабораторной работе), также есть кое-какие другие варианты. Можем сразу указать размер окна.

Также вы можете открыть для редактирования уже имеющуюся экранную форму.

Давайте создадим простой диалог: выберите Dialog with Buttons Bottom и нажмите *Создать*.

2. Перейдем к заполнению формы содержимым. В левой части экрана найдите ***Label*** на ***панели виджетов*** в разделе ***Display Widgets*** и поместите его на форму. Теперь уже в правой части экрана в ***редакторе свойств*** в разделе ***QLabel*** в поле ***text*** напишите что-нибудь (либо просто щелкните два раза на самом ***Label***).

Если текст не поместится, то растяните Label – его размеры можно менять, как на самой форме, так и редактируя значения свойства ***geometry*** раздела ***QWidget***.

3. Нажмите комбинацию *Ctrl+R* или выберите пункт меню *Форма – Предпросмотр* (рис. 2.2).

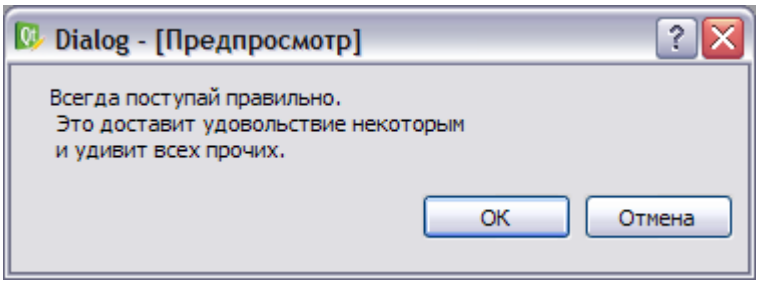


Рисунок 2.2 – Предпросмотр созданного диалога

4. Обратите внимание на кнопку . Дабы снабдить ее некой функциональностью, стоит заполнить свойство ***whatsThis*** для интересующих пользователя элементов формы. Сделать это можно не только через окошко ***Свойства*** (оно же ***Property Editor***), но и с помощью контекстного меню (рис. 2.3). Кстати, в качестве подсказки может выступить даже картинка, для этого всего лишь стоит нажать ***Insert Image*** .

В этом случае вам предстоит подключить файл ресурсов, воспользовавшись кнопкой ***Edit Recourses*** . О подключении файла ресурсов см. раздел 3.1.2, п.3.

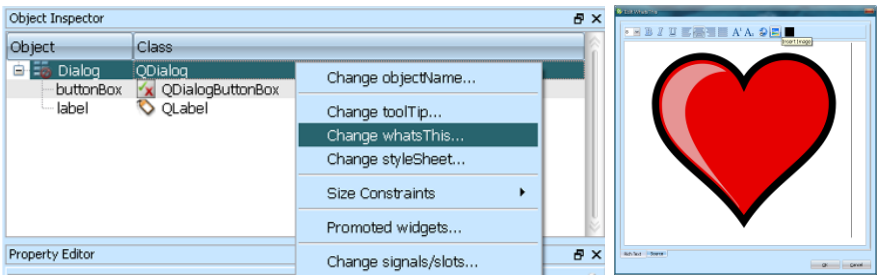


Рисунок 2.3 – Редактирование подсказок

Если же мы хотим, чтобы подсказка всплывала сама по себе, то следует работать со свойством ***toolTip***.

Расположите на форме еще несколько элементов и изучите панель инструментов (рис. 2.4): как минимум, освойте группировку элементов на форме.

Рисунок 2.4 – Панель инструментов

Помимо этого, мы еще можем поменять шрифт (свойство ***font***) и курсор (свойство ***cursor***), когда он находится над одним из элементов нашей формы.

А теперь можно и ознакомиться с различными вариантами отображения (рис. 2.5), выбирая *Форма – Предпросмотр в*.

Это интересно: *в Qt 5 оставили все стили для Windows, но устранили Linux-стили вроде Cleanlooks - взамен появился универсальный для всех платформ Fusion.*

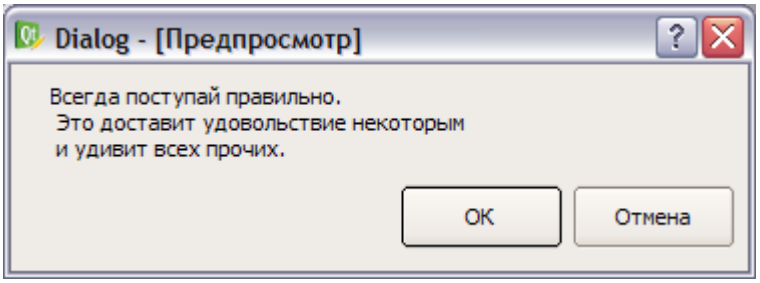


Рисунок 2.5 – Предпросмотр того же диалога в стиле Cleanlooks

5. Сохраните ваш труд: *Файл → Сохранить*. Дайте ему некое осмысленное название, например: Simple_dialog.ui.

6. Закройте Qt Designer и откройте Qt Creator. Там загрузите проект Hello (созданный в ходе предыдущей лабораторной). И щелкните на нем правой кнопкой мыши – появится контекстное меню, в котором выберите *Добавить существующие файлы…* (рис. 2.6) и добавьте в проект ваш диалог.

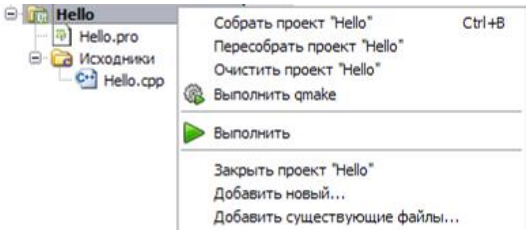


Рисунок 2.6 – Контекстное меню для проекта Hello

7. А теперь попытаемся его включить в программный код. Если вы внимательно просмотрите проектный файл, то увидите, что в него добавилась строчка:

```
FORMS += Simple_dialog.ui,
```

а еще у нас в каталоге (где находится исполняемый файл) появится (после того, как вы выполните пункт 7.1 и сделаете сборку проекта ) файл: ui_Simple_dialog.h. Этот файл автоматически генерируется на основе ui-файла, его и необходимо подключать к проекту.

Есть два варианта, как вывести эту форму на экран: простой и через наследование.

7.1. Начнем с простого.

```
#include <QtGui>
#include "ui_Simple_dialog.h"

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

    QDialog* form = new QDialog;
    Ui::Dialog ui;
    ui.setupUi(form);
    form->show();

    return app.exec();
}
```

Пояснение: во 2-й строке мы подключаем файл с нашей формой.

Это важно: *если вы назвали файл как-то иначе (не Simple_dialog.ui), то для того, чтобы его подключить, следует написать: «*#include "ui_<имя файла без расширения>.h"*».*

Наибольший интерес представляют строки с 8-й по 10-ю.

(8) Теперь мы ведь работаем с диалогом, посему его и создаем.

(9) Создаем переменную для формы, разработанной в Qt Designer. Когда напишете Ui, можете щелкнуть по этим двум буквам правой кнопкой мыши и выбрать *Перейти к символу под курсором* (рис. 2.7); таким образом, вы очутитесь в листинге ui_Simple_dialog.h (здесь ничего менять нельзя, но посмотреть можно).

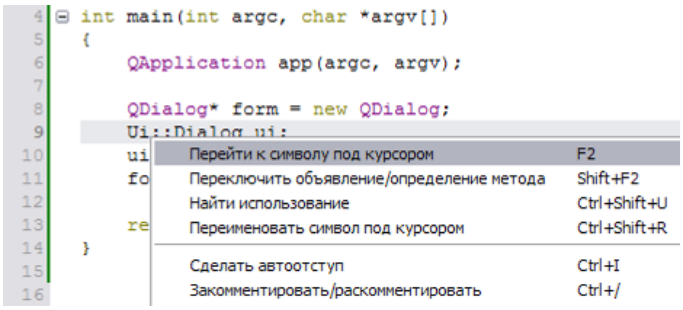


Рисунок 2.7 – Контекстное меню для Ui

Dialog – это название созданной нами в Qt Designer формы, его можно поменять при желании (рис. 2.8).

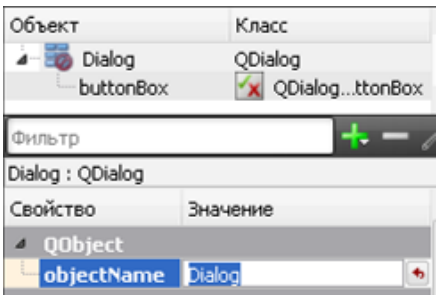


Рисунок 2.8 – Редактор свойств, изменение имени формы

(10) Связываем переменную ui с переменной form.

7.2. А теперь попробуем сделать то же самое, но только с наследованием. Для этого добавим в проект еще два файла:

Dialog.h

```
#ifndef DIALOG_H
#define DIALOG_H
#include "ui_Simple_dialog.h"

class Dialog : public QDialog
{
    Q_OBJECT

public:
    Dialog(QDialog *parent = 0);
```

```
private:
    Ui::Dialog ui;
};
#endif // DIALOG_H
```

и **Dialog.cpp**

```
#include "dialog.h"

Dialog::Dialog(QDialog *parent) :
    QDialog(parent)
{
    ui.setupUi(this);
}
```

а также изменим (удалив все, что там было) основной файл:

```
#include <QWidget>
#include <QApplication>
#include "dialog.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    Dialog form;
    form.show();
    return app.exec();
}
```

2.2 Работа с формами

Создадим заготовку для следующей лабораторной работы, сделав диалог поиска (при этом стоит реализовать такие особенности, как раскрытие дополнительных параметров для поиска, по нажатию на кнопку *Больше>>*, различные вариации поиска и др.) – см. рис. 2.9 или нажмите *Ctrl+H* в MS Word.

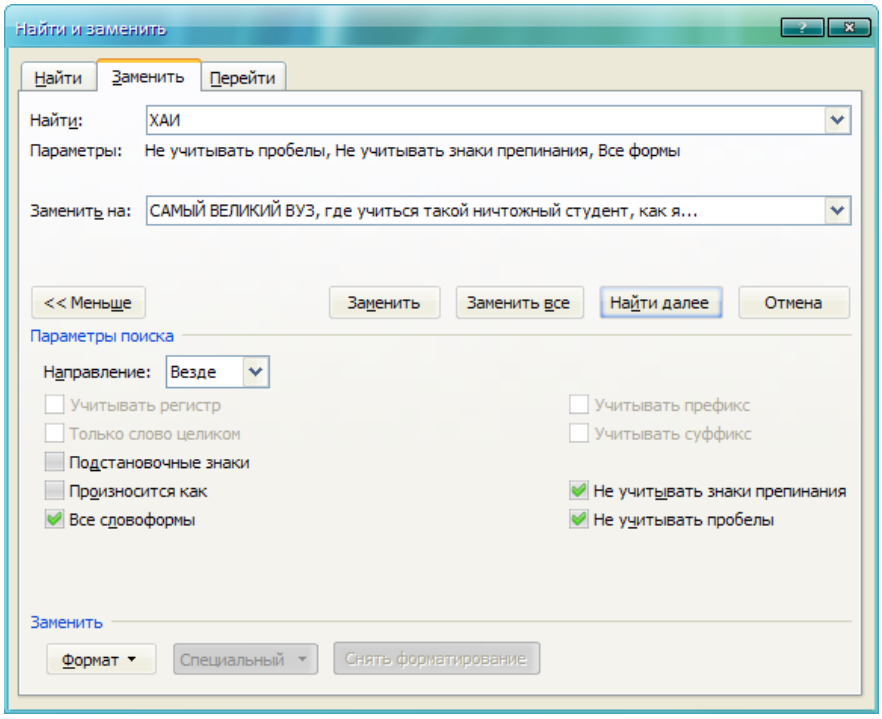


Рисунок 2.9 – Диалог поиска и замены

2.2.1 Создание формы

Создадим пустой проект, назовем его «*Find*».

Добавьте в файл проекта: `greaterThan(QT_MAJOR_VERSION, 4): QT += widgets` либо `QT += core gui` – в зависимости от версии Qt (первый вариант для версии старше 4.0, собственно, он вам и нужен).

Для того чтобы подключить новую форму к вашей программе, следует: щелкнуть правой кнопкой мыши на корневой папке в дереве проектов (рис. 2.10), затем выбрать *Добавить новый…*, в появившемся окне выбрать, соответственно, *Qt* и *Класс формы Qt Designer* (рис. 2.11).

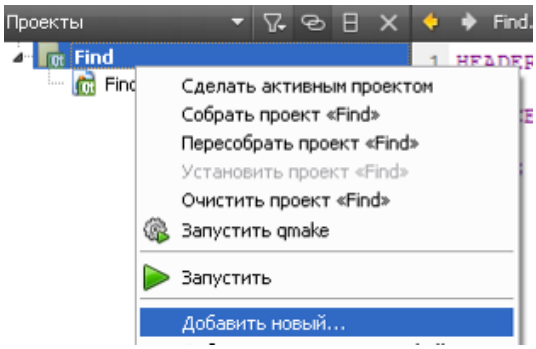


Рисунок 2.10 – Добавление файлов к проекту

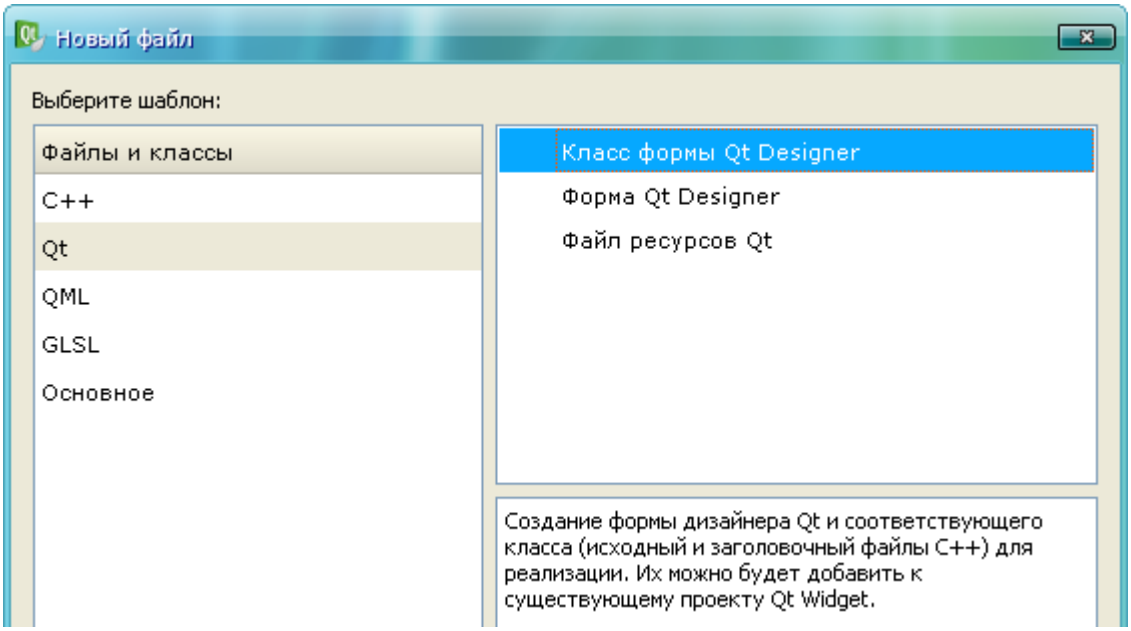


Рисунок 2.11 – Подключение новой формы

Давайте создадим простой диалог без кнопок: выберите Dialog without Buttons и нажмите *Создать*.

Это важно: *НЕ выбирайте MainWindow, мы создаем обычное диалоговое окно.*

Затем задайте имя класса: «Find_Dialog», соответственно, файл формы будет носить имя «find_dialog.ui». Такие же имена, но с другим расширением будут иметь заголовочный файл и файл с исходными кодами (рис. 2.12).

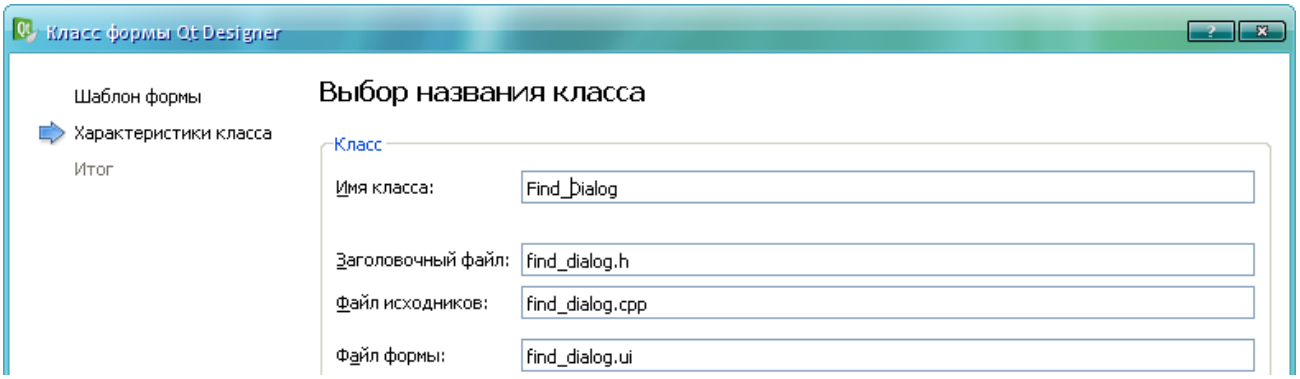


Рисунок 2.12 – Подключение новой формы. Выбор названия класса

2.2.2 Подключение формы и наполнение формы

Собственно, само подключение реализуется так (в созданном вами файле main.cpp, например):

```
#include <QWidget> //или #include <QtGui> для версии Qt ниже 4.0
#include <QApplication>
#include "find_dialog.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    Find_Dialog form_for_find;
    form_for_find.show();
    return app.exec();
}
```

Добавьте на форму элементы, как минимум, указанные на рис. 2.13, как максимум – на рис. 2.9.

Это важно: *элементы, которые вы будете размещать под кнопкой «Больше», следует располагать на отдельном виджете, который будет называться «GroupBox».*

В общем виде с компоновкой в инспекторе объектов ваше окно будет выглядеть вот так (рис. 2.13):

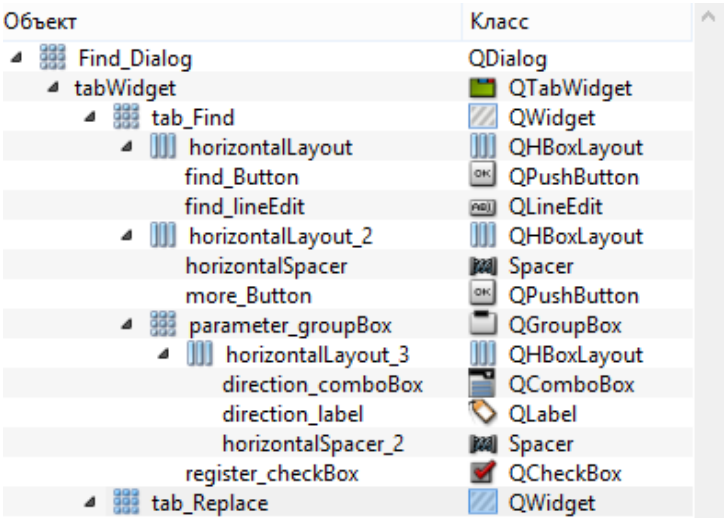
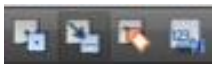


Рисунок 2.13 – Пример компоновки и набора виджетов

2.2.3 Создание динамически изменяемых окон

1. Для того чтобы настроить видимость виджета единоразово, стоит воспользоваться методом *setVisible*(bool).

> ***Это интересно:*** *в Qt в инспекторе свойств вы не найдете свойство visible – приходится устанавливать его вручную, вызывая для объекта метод setVisible(bool) в конструкторе.*

2. Зайдите в режим изменения сигналов и слотов на форме: нажмите на вторую кнопку –  или F4.

3. Соедините кнопку «Больше» и виджет, на котором вы размещали дополнительные настройки поиска (рис. 2.14).

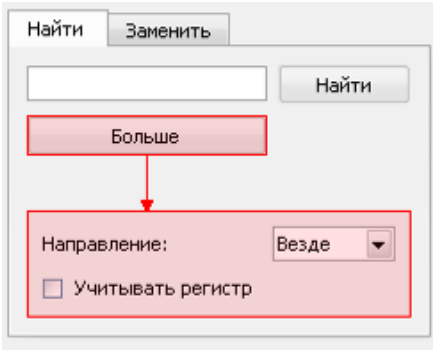


Рисунок 2.14 – Создание соединений

4. Настройте соединение (рис. 2.15).

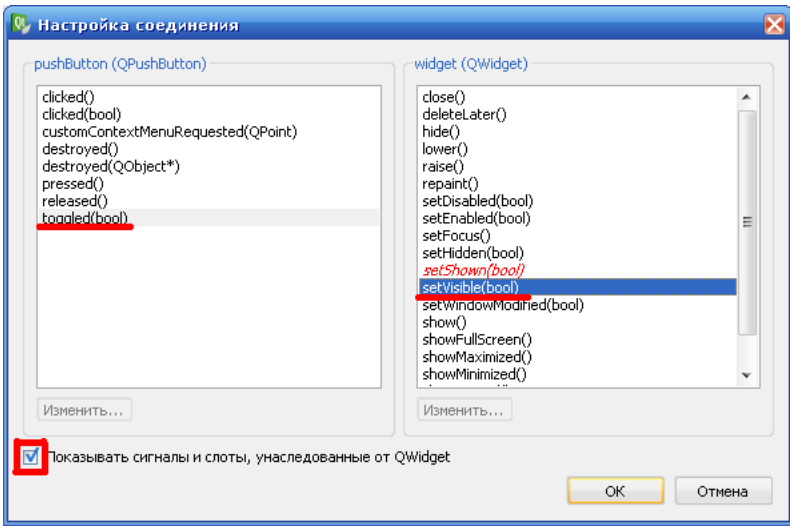


Рисунок 2.15 – Настройка соединений

Теперь, когда кнопка посылает сигнал о том, что она находиться в нажатом / отжатом состоянии, для виджета вызывается слот, который изменит его видимость.

> ***Это важно:*** *на один слот можно назначить любое количество сигналов.*

5. Настройте свойство *checkable* для кнопки «Больше», установив у него значение *true* – тем самым вы укажете, что кнопка при клике по ней будет оставаться в нажатом положении, при повторном – в отжатом, соответственно.

6. Используя компоновку  , добейтесь фиксированного расположения элементов на экране (см. рис. 2.12).

7. Если вы хотите, чтобы у вашей формы были фиксированные размеры, необходимо будет дописать следующий код в конструкторе класса Find_Dialog:

```
    layout()->setSizeConstraint(QLayout::SetFixedSize);
```

либо это можно изменить через редактор свойств объекта Find_Dialog (рис. 2.16):

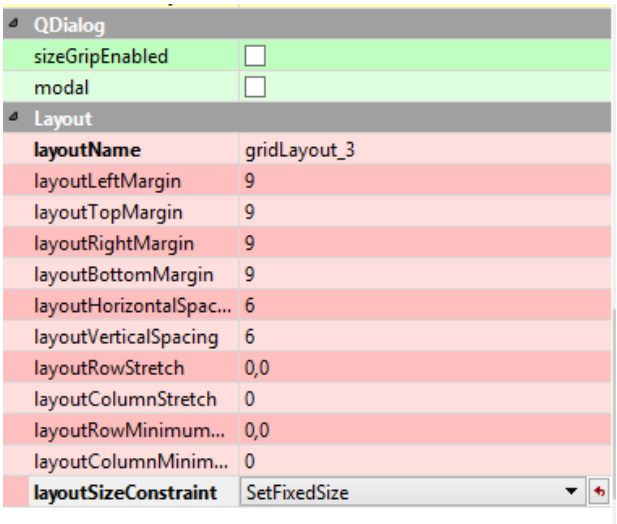


Рисунок 2.16 – Установка фиксированных размеров формы

8. Для того чтобы проверить, есть ли текст для поиска, и, соответственно, можно ли нажимать кнопку «*Поиск*»:

```
    ui->find_Button->setEnabled(!(ui->find_lineEdit->text().isEmpty()));
```

или:

```
   ui->find_Button->setEnabled(!(arg1.isEmpty()));      // для процедуры
textChanged(const QString &arg1)
```

P.S. Если у вас не получается реализовать расширяемое окно – то посмотрите: QT 4: программирование GUI на C++, 31 с.

P.P.S. Как вы, наверное, заметили, на формах, созданных в Qt Designer, корректно отображаются символы кириллицы – все потому, что он сам проводит перекодировку символов. Это еще один несомненный плюс Qt Designer.

2.3 Установка пиктограммы приложения

Пиктограмма приложения, обычно отображаемая в верхнем левом углу окон верхнего уровня приложения, устанавливается с помощью вызова метода QWidget::setWindowIcon() на виджетах верхнего уровня.

В Qt Designer для изменения пиктограммы приложения следует модифицировать свойство ***windowIcon*** главного виджета приложения.

Для того, чтобы изменить пиктограмму самого исполняемого файла приложения, которая отображается на рабочем столе (т. е. до запуска приложения), необходимо использовать другой, платформо-независимый метод.

2.3.1 Установка пиктограммы приложения в Windows

Прежде всего, создайте файл формата ICO с растровым изображением пиктограммы. Сделать это можно с помощью, например, Microsoft Visual C++: Выберите *File → New*, затем в появившемся диалоге выберите вкладку *File* и выберите *Icon* (обратите внимание на то, что вам не нужно загружать своё приложение в Visual C++; мы используем только редактор пиктограмм).

Сохраните файл ICO в каталоге исходных кодов вашего приложения под именем, например, myappico.ico. Затем создайте текстовый файл с именем, скажем, myapp.rc, в котором поместите единственную строку текста:

```
IDI_ICON1 ICON DISCARDABLE "myappico.ico"
```

В заключение добавьте следующую строку в свой файл проекта (с расширением – «.pro»):

```
    RC_FILE = myapp.rc
```

Соберите приложение заново. Файл .exe теперь будет отображаться в Проводнике с вашей пиктограммой.

Контрольные вопросы

1. Что такое Qt Designer?
2. Какие преимущества и возможности предоставляет Qt Designer?
3. Какая программа создает заголовочный C++ файл для *.ui файла, созданного в Designer?

Тест

1. Как называется главный графический компонент в Qt?
– QObject;
– QWidget;
– QtGui;
– QFrame.

2. Как много слотов можно назначить на один сигнал?
– сколько угодно;
– 4;
– не более двух;
– 256.

3. Перед уничтожением объекта, наследованного от QObject, необходимо отсоединить все сигналы и слоты?
– да;
– нет.

3 РАЗРАБОТКА ТЕКСТОВОГО РЕДАКТОРА В QT

Цель работы *– обучение работе с мастером в среде Qt Creator, разработка собственного мобильного текстового редактора.*

Теоретическое введение

В качестве теории рассмотрим работу с файлами в библиотеке Qt. Для представления файла в программе используется класс QFile.

Рассмотрим типовую последовательность работы с файлом:

1. Подключить библиотеку для работы с файлами:

```
#include <QFile>
```

2. Узнать имя файла, используя стандартный диалог (QFileDialog).

Стандартный диалог выбора файла (рис. 3.1) предназначен для того, чтобы дать пользователю возможность выбрать файл (каталог) для открытия или сохранения. В библиотеке Qt стандартный диалог выбора файла реализуется классом QFileDialog.

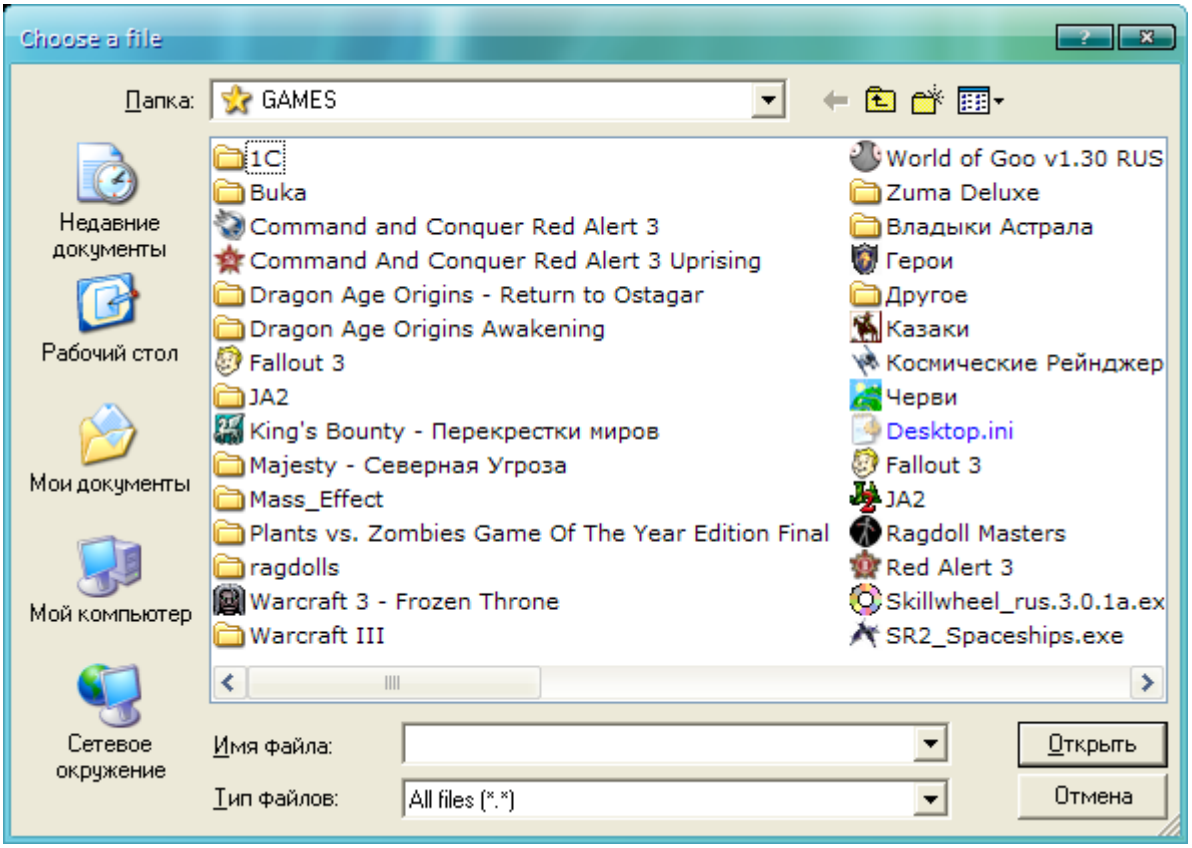


Рисунок 3.1 – Стандартный диалог выбора файла

В большинстве случаев при работе с классом QFileDialog используется одна из следующих статических функций:

– getOpenFileName(...)
– getSaveFileName(...)
– getOpenFileNames(...)

– getExistingDirectory(...)

Все параметры этих функций имеют значения по умолчанию и их можно не указывать при необходимости.

Диалог выбора файлов может отбирать файлы по указанным расширениям, при этом пользователь может выбрать один из предложенных вариантов фильтрации. Строка фильтров состоит из описаний фильтров, разделенных двойным знаком – точка с запятой. Описание фильтра состоит из имени и перечня шаблонов для имен файлов в круглых скобках:

“Open Office document (*.odt, *.odp);; Portable document format (*.pdf)”.

QFileDialog::getOpenFileName(

– QWidget * – указатель на окно-родитель;

– const QString & – строка заголовка;

– const QString & – начальный каталог (пустая строка, если использовать текущую директорию);

– const QString & – фильтр файлов по расширению;

– QString * – изначально выбранный фильтр, обычно 0;

– Options – опции настройки, обычно 0;

) возвращает QString – имя выбранного файла, пустая строка в случае отмены.

3. Создать объект файла (QFile).

Один из конструкторов QFile принимает имя файла в качестве параметра:

QFile::QFile(const QString & – имя файла*).*

Созданный объект привязан к файлу, но файл при этом не открывается – он должен быть открыт методом open().

4. Открыть файл (QFile).

QFile::open(

– OpenMode – режим доступа:

QIODevice::ReadOnly – только для чтения;

QIODevice::WriteOnly – только для записи, имеющиеся данные затираются;

QIODevice::ReadWrite – для чтения и записи, новые данные добавляются к уже существующим;

QIODevice::Append – открыть файл для дополнения файла в конец; дополнительно можно указать QIODevice::Text для взаимодействия с файлом в текстовом режиме (через операцию побитового ИЛИ)

) возвращает bool – успешно ли открытие.

5. Создать поток для ввода/вывода (QDataStream или QTextStream) и связать его с файлом.

Используя объект файла (QFile), можно читать и записывать данные, хранящиеся в файле. Однако чтение/запись выполняется на низком уровне – побайтово. Для высокоуровневой работы с файлом (чтения/записи чисел, строк,

дат и т.д.) используются потоки. В библиотеке Qt потоки ввода/вывода представлены классами:

– QDataStream – записывает и читает данные в двоичном формате;

– QTextStream – записывает и читает данные в текстовом формате.

Чтобы поток мог читать и записывать данные в файл, он должен быть связан с объектом файла. Связь потока с файлом осуществляется при создании потока - в конструктор потока передается указатель на объект файла.

```
    if (!file_Name.isEmpty()) // Если имя файла задано, то...
    {
        QFile file(file_Name); // Создаем файл
        if (file.open(QIODevice::WriteOnly)) // Открываем файл только для записи
        {
            QDataStream output(&file); // Создаем поток для записи данных
            output<<"Текст"; // Записывае данные
        }
    }
```

Не забудьте их подключить, если они вам необходимы:

```
#include <QDataStream>
#include <QTextStream>
```

Пример программного кода, который позволяет ввести в поток output строку str и число N:

```
QTextStream out(&file);
QString str = "Текст";
int N = 2;
output << str << N;
```

6. Считать/записать данные с использованием потока (QDataStream или QTextStream).

Ввод (чтение) из потока осуществляется через перегруженную операцию >>:

поток >> переменная_для_ввода;

Несколько операций ввода могут быть записаны в цепочку:

поток >> переменная1 >> переменная2;

Вывод (запись) в поток осуществляется аналогичным образом с использованием операции <<.

Операции ввода/вывода в поток сами определяют тип вводимых/выводимых данных и действуют соответственно.

7. Закрыть файл (QFile).

Пример:

```
    QString file_Name = QFileDialog::getOpenFileName(this, QString("Открыть
файл"), QString(), QString("Текстовые файлы (*.txt,*.bat);; Все файлы (*.*)"));
    QFile file(file_Name); // либо file("c://test.txt");
    file.open(QIODevice::Append | QIODevice::Text);
    QTextStream out(&file);
    QString str="Текст";
    out<<str;
    out<<"\n";
    file.close();
```

3.1 Создание полноценного текстового редактора

3.1.1 Создание меню

Наших знаний уже достаточно для создания полноценного приложения.

1. Для начала создадим новый проект. Так как до этого мы создавали почти все с нуля, вручную подключали формы и полностью вникли в этот процесс, то воспользуемся мастером.

Выберите *Файл → Создать файл или проект…*, а там *Приложение → Приложение Qt Widgets* (рис. 3.2).

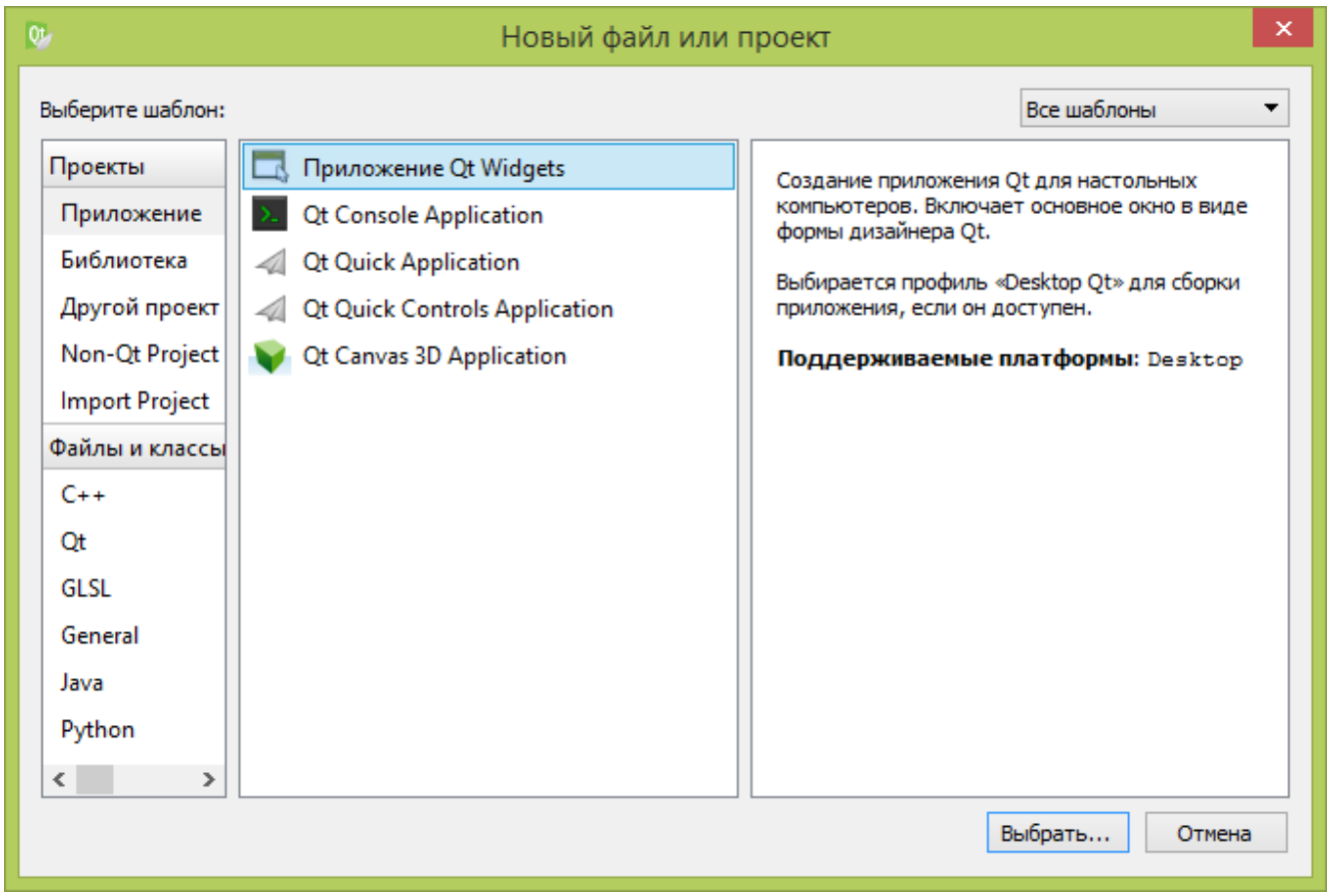


Рисунок 3.2 – Создание проекта с помощью мастера

Зададим имя для проекта: QReader.

Напоминаем – следует избегать в пути размещения символов кириллицы (рис. 3.3).

Нажмите *Далее*. Нам предложат выбрать варианты, для какой платформы будет создано наше приложение, но мы пока не собираемся заниматься разработками для мобильных телефонов, поэтому просто еще раз нажмем *Далее*.

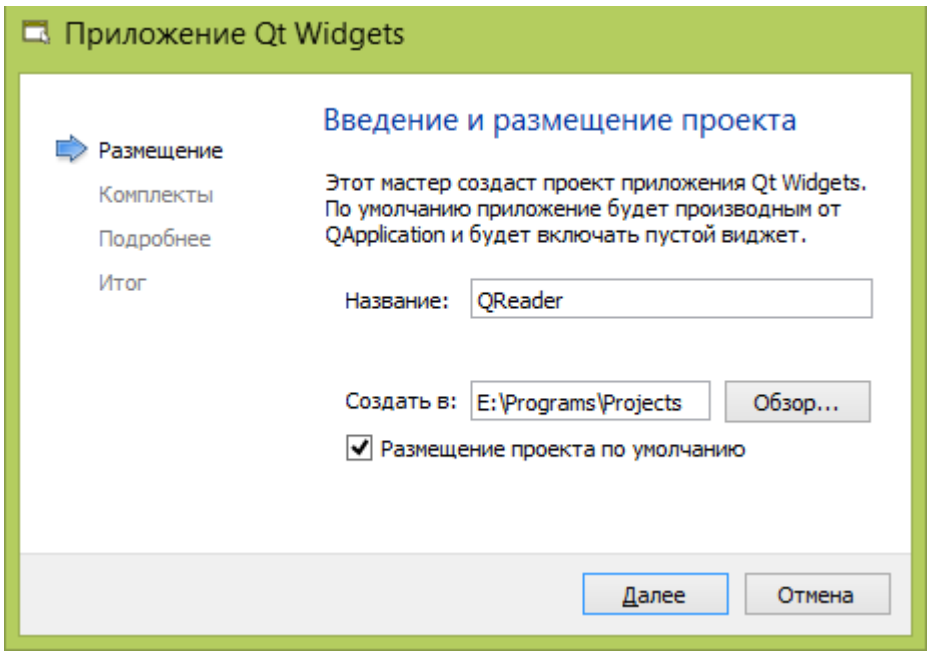


Рисунок 3.3 – Создание проекта, в который уже будет включена форма, с использованием мастера

Перед нами предстанет информация о генерируемой форме (рис. 3.4). Если внимательно изучить это окно, то можно понять, что предыдущее приложение (диалоговое окно) мы могли создать тремя кликами мыши. Ничего не меняйте, просто опять нажмите *Далее* и *Завершить*.

Проект готов. Можете посмотреть сгенерированные файлы и запустить приложение. Должно работать.

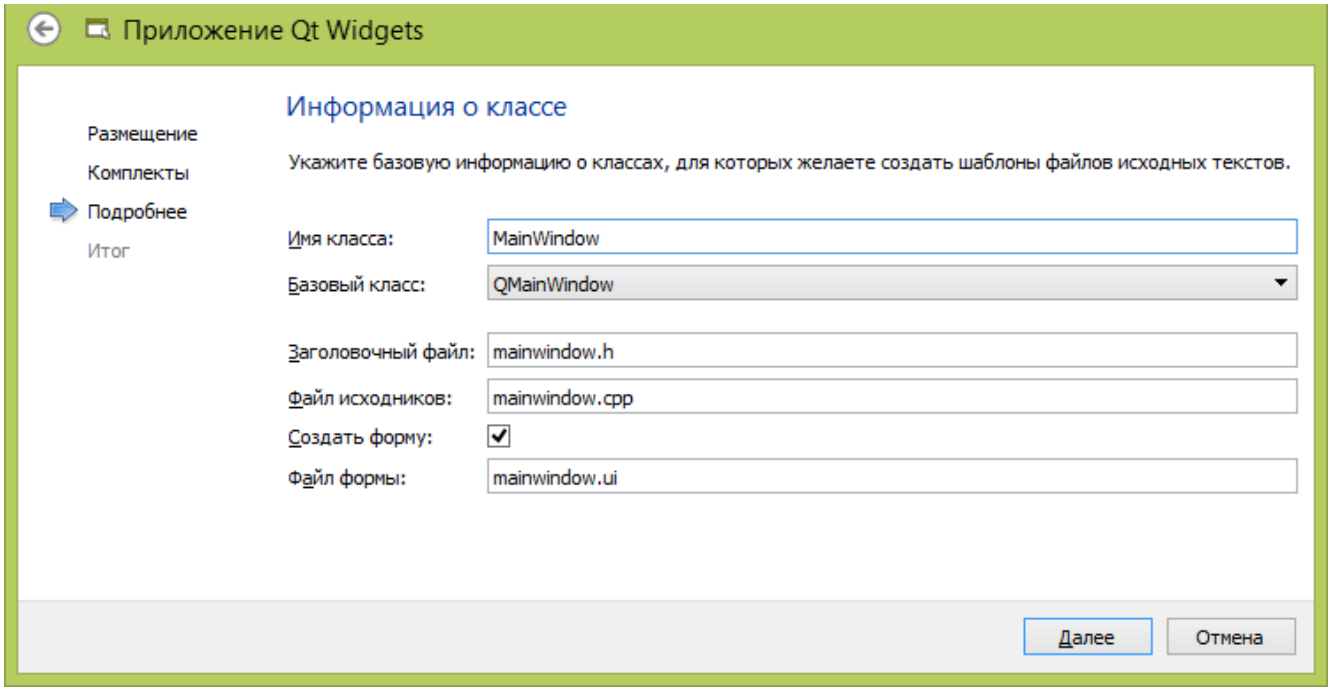


Рисунок 3.4 – Информация о классе главного окна

2. Перейдем к созданию меню. Щелкните дважды на форме mainwindow.ui – откроется визуальный редактор (рис. 3.5). Найдите глазами слова *Пишите здесь* и последуйте этому указанию.

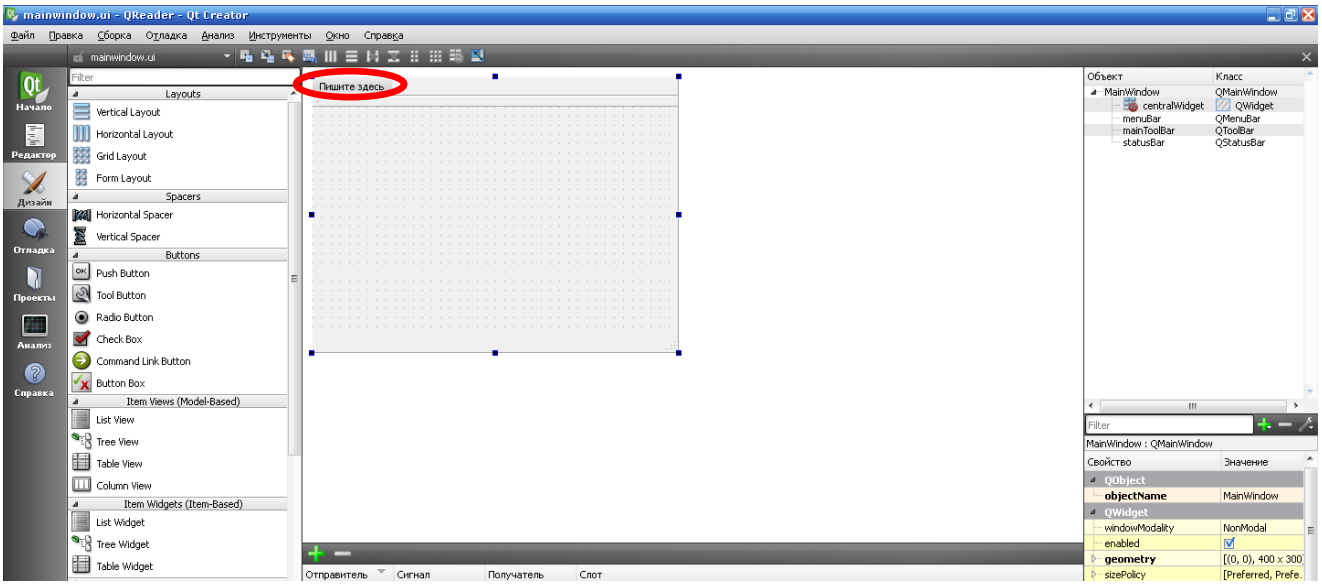


Рисунок 3.5 – Редактирование главного окна

Создайте стандартное меню *Файл* с пунктами *Новый*, *Открыть*, *Сохранить*, *Сохранить как…*, *Выход*. Между *Сохранить как…* и *Выход* добавьте разделитель.

Разделитель добавляется над активной темой меню.

> ***Это важно:*** *пункты меню лучше сразу создавать на английском языке, а потом уже переименовывать на русском, так действия, привязанные к этим пунктам меню, сразу будут носить осмысленное имя.*

3.1.2 Редактирование действий

1. Если вы внимательно посмотрите вниз экрана, то обнаружите, что там появились так называемые *Действия* (*Action*) – реакции на нажатие соответствующих пунктов меню.

2. Давайте модифицируем их, заменив названия на более осмысленные, например, action_New, action_Open, action_Save, action_Save_as и action_Quit, а также добавим к ним горячие комбинации клавиш. Делается это просто: в окне редактирования действия (рис. 3.6, чтоб попасть в него дважды щелкните на самом действии) нажмите: Ctrl+N, Ctrl+O, Ctrl+S, Ctrl+Q для соответствующих действий.

> ***Это важно:*** *вводите наименования действий внимательно, например, не стоит случайно ставить в нем пробел в конце (это не так просто заметить).*

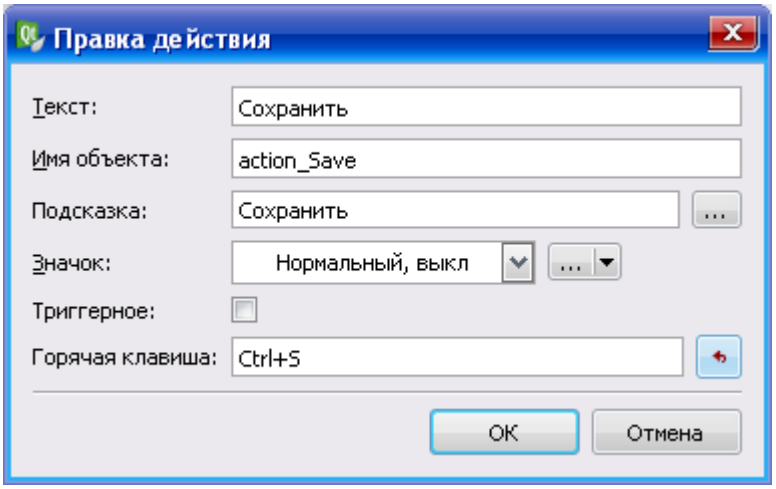


Рисунок 3.6 – Окно редактирования действия

3. Для того, чтобы сделать меню более привлекательным, добавим к каждому действию свою иконку, а чтоб иконкам было где жить, создадим файл ресурсов (рис. 3.7, 3.8).

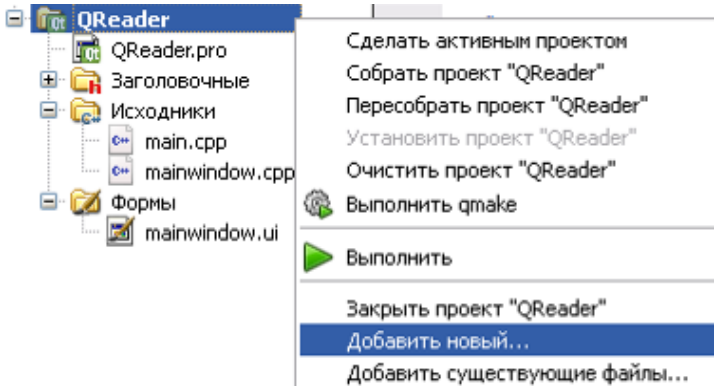


Рисунок 3.7 – Добавление нового файла в проект

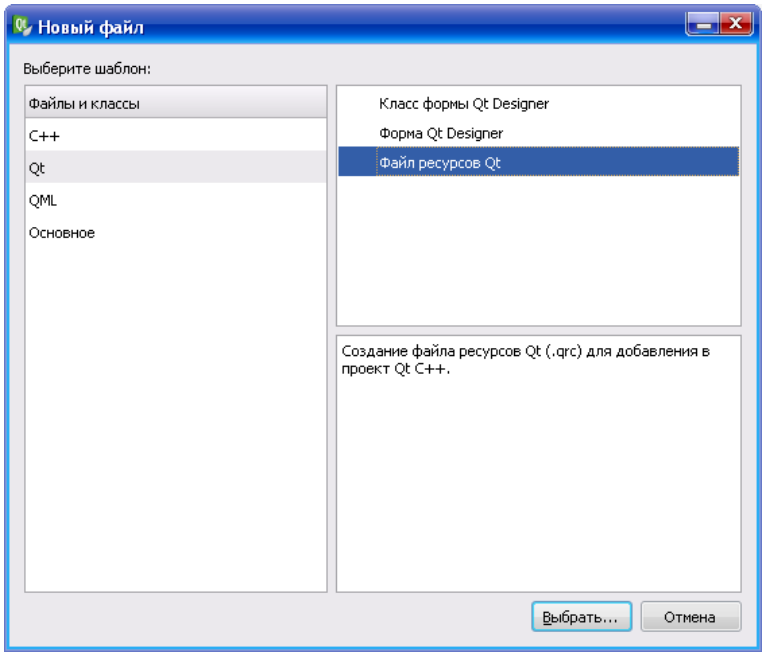


Рисунок 3.8 – Создание файла ресурсов

Нас попросят ввести его имя (рекомендую использовать имя проекта, в данном случае QReader, но это остается на вашем усмотрении), а также разместить ресурсный файл (в принципе его можно разместить, где угодно, но опять же лучше расположить его в директории проекта, что услужливо рекомендует программа). Нажмите *Далее* и *Завершить*.

Теперь дважды щелкните на QReader.qrc (если вы так назвали файл) на боковой панели. Затем нажмите на кнопку *Добавить* и выберите *Добавить префикс* (рис. 3.9). Можете поменять название префикса на более осмысленное.

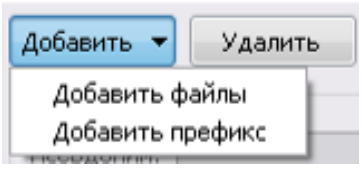


Рисунок 3.9 – Добавление префиксов и файлов
в ресурсный файл

Сейчас нам следует найти (или самим создать) файлы, которые мы будем использовать в качестве иконок. Размеры иконок должны быть 32x32. Рекомендую взять файлы из *«<Путь установки Qt>\<№ версии Qt>\ \mingw492_32\examples\widgets\mainwindows\application\images»*, либо из *«\\K105\labpract\Qt\Pictures\images»*. Добавьте их в ресурсный файл.

Советую переместить файлы изображений в каталог с программой и поместить их, например, в созданную вами папку «images».

Снова вернемся в окно редактирования формы и зададим иконки для всех действий. Сделать это можно через Окно редактирования действия (см. рис. 3.6), либо через окно редактирования свойств (рис. 3.10). Таким образом, задайте иконки для всех пунктов меню (рис. 3.11).

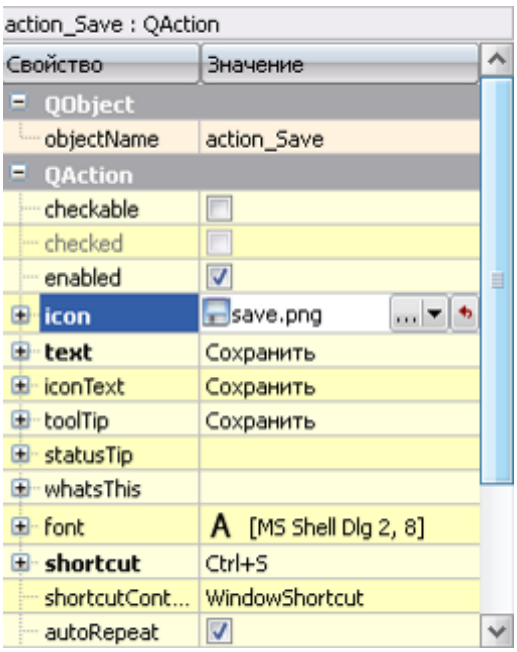


Рисунок 3.10 – Окно редактирования свойств действия

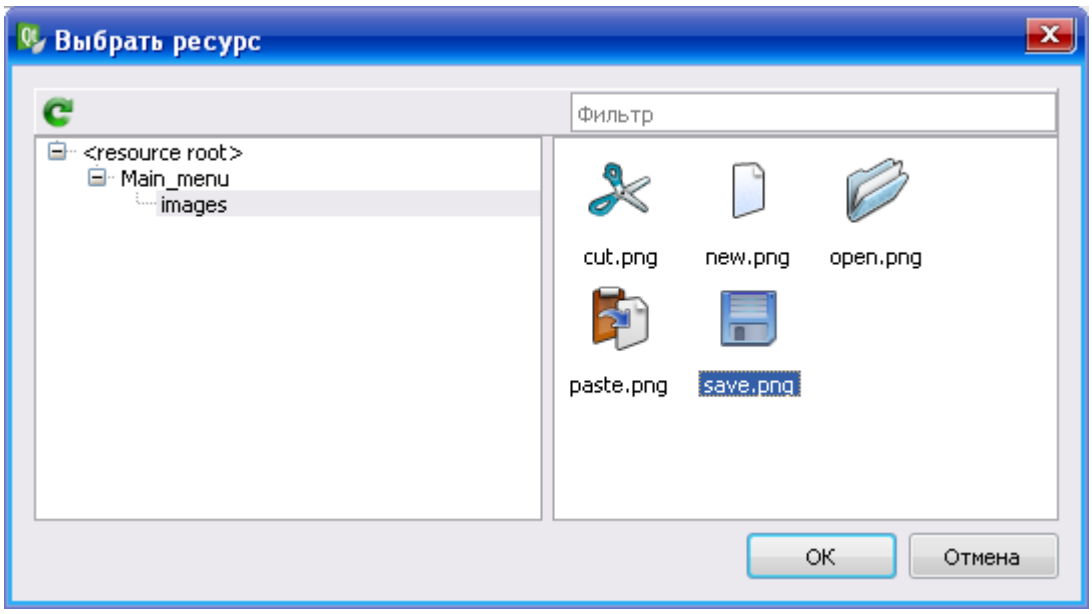


Рисунок 3.11 – Выбор иконок для пунктов меню

3. Рядом с **редактором действий** есть **редактор сигналов и слотов**. Перейдем в него и последовательно выберем: action_Quit, triggered(), MainWindow, close(). Теперь по нажатию на пункт меню *Выход* главное окно приложения закроется.

4. Добавим на главное окно элемент *Text Edit* (он находится в разделе Input Widgets). Назовем его «textEdit».

Дабы окошко с текстом масштабировалось вместе с приложением, необходимо настроить его компоновку.

5. Начнем работать с файлами. Создадим свою процедуру для открытия файла: выберите действие action_Open, кликните правой кнопкой мыши и выберите *Перейти к слоту…* (рис. 3.12).

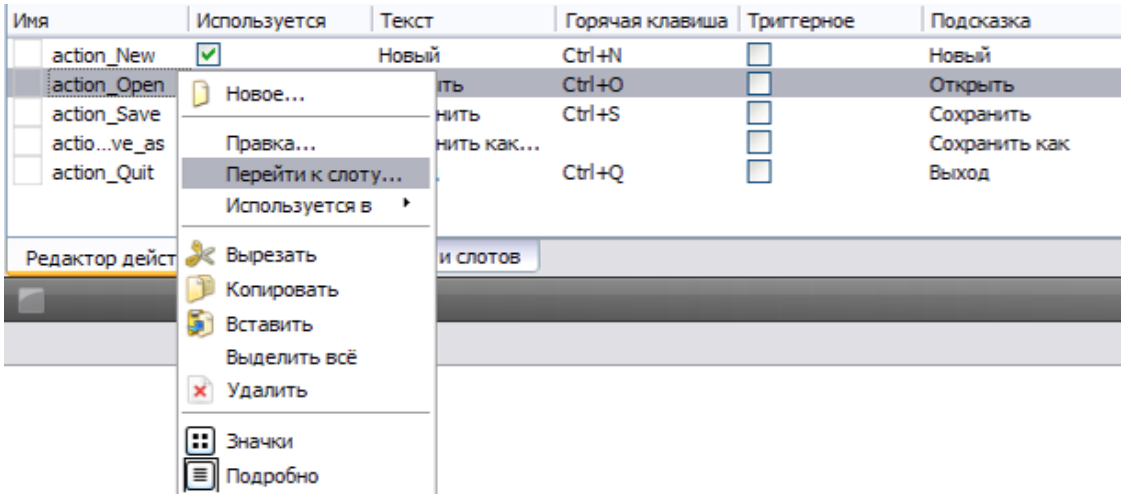


Рисунок 3.12 – Привязка сигналов и слотов

В открывшемся диалоговом окне выберите сигнал triggered() и нажмите *OK*. Вы попадете в окно редактирования программного кода. Здесь стоит написать то, что произойдет при выборе пункта меню *Открыть*.

6. Займемся написанием программного кода.

Добавим в эту функцию следующий код:

```
void MainWindow::on_action_Open_triggered()
{
        QString fileName = QFileDialog::getOpenFileName(this);
        if (!fileName.isEmpty()) loadFile(fileName);
}
```

Нам приходится дописать еще пару своих функций – первая устанавливает имя файла, с которого идет чтение, вторая, собственно, и читает из файла:

```
void MainWindow::setCurrentFileName(const QString &fileName)
{
    this->fileName = fileName;
    ui->textEdit->document()->setModified(false);
    QString shownName;
    if (fileName.isEmpty())
        shownName = "untitled.txt";
    else
        shownName = QFileInfo(fileName).fileName();
    setWindowTitle(tr("%1[*] - %2").arg(shownName).arg(tr("Rich Text")));
    setWindowModified(false);
}

void MainWindow::loadFile(const QString &fileName)
{
    if (!QFile::exists(fileName))
    {
     QMessageBox::warning(this, tr("Application"),
                          tr("Cannot find file %1:\n%2.")
                          .arg(fileName));
        return;
    }
    QFile file(fileName);
    if (!file.open(QFile::ReadOnly))
    {
        QMessageBox::warning(this, tr("Application"),
                             tr("Cannot open file %1:\n%2.")
                             .arg(fileName));
           return;
    }
    #ifndef QT_NO_CURSOR
        QApplication::setOverrideCursor(Qt::WaitCursor); //делаем курсор "часики"
    #endif
    QByteArray data = file.readAll();
    QTextCodec *codec = Qt::codecForHtml(data);
    QString str = codec->toUnicode(data);
    if (Qt::mightBeRichText(str)) ui->textEdit->setHtml(str);
    else {
        str = QString::fromLocal8Bit(data);
        ui->textEdit->setPlainText(str);
    }

    setCurrentFileName(fileName);
    #ifndef QT_NO_CURSOR
    QApplication::restoreOverrideCursor(); //восстанавливаем обычный курсор
```

```
    #endif
    statusBar()->showMessage(tr("File loaded"), 2000);
}
```

Следует не забыть добавить описание заголовков этих методов в класс (файл mainwindow.h):

```
private:
       void loadFile(const QString &fileName);
       void setCurrentFileName(const QString &fileName);
```

и одного атрибута:

```
private:
        QString fileName;
```

да и не забыть подключить заголовочные файлы (в файле mainwindow.h):

```
#include <QFileDialog>
#include <QTextCodec>
#include <QMessageBox>
#include <QTextStream>
```

а еще вызвать одну из процедур в конструкторе нашего главного класса:

```
      setCurrentFileName(QString());
```

Все! На данном этапе у нас готов простенький просмотрщик текстовых файлов. Проверьте работоспособность приложения.

7. Для того, чтоб наш просмотрщик стал простеньким текстовым редактором, следует добавить возможность сохранения файла. Для этого повторите пункт 5, но только для действий action_Save и action_Save_as.

8. Дописываемый код будет выглядеть следующим образом:

```
void MainWindow::on_action_Save_triggered()
{
    if (fileName.isEmpty())
    {
        on_action_Save_as_triggered(); return;
    }
    QTextDocumentWriter writer(fileName);
    bool success = writer.write(ui->textEdit->document());
    if (success) ui->textEdit->document()->setModified(false);
}
void MainWindow::on_action_Save_as_triggered()
{
    QString fn = QFileDialog::getSaveFileName(this, tr("Save as..."), QString(),
tr("ODF files (*.odt);;HTML-Files (*.htm *.html);;All Files (*)"));
    if (fn.isEmpty()) return;
    if (! (fn.endsWith(".odt", Qt::CaseInsensitive) || fn.endsWith(".htm",
Qt::CaseInsensitive) || fn.endsWith(".html", Qt::CaseInsensitive)) ) fn += ".odt";
// default
    setCurrentFileName(fn);
    on_action_Save_triggered();
}
```

Нам необходимо добавить еще один заголовочный файл:

```
     #include <QTextDocumentWriter>
```

а еще изменить процедуру открытия файла:

```
     void MainWindow::on_action_Open_triggered()
     {
        if (maybeSave())
```

```
        {
            QString fileName = QFileDialog::getOpenFileName(this);
            // Здесь стоит написать ваши собственные поля метода!!!
            if (!fileName.isEmpty()) loadFile(fileName);
        }
    }
```

Как видим, здесь нам следует проверить, стоит ли сохранять файл:

```
bool MainWindow::maybeSave()
{
    if (!ui->textEdit->document()->isModified())
        return true;
    if (fileName.startsWith(QLatin1String(":/")))
        return true;
    QMessageBox::StandardButton ret;
    ret = QMessageBox::warning(this, tr("Application"),
                               tr("The document has been modified.\n"
                                  "Do you want to save your changes?"),
                               QMessageBox::Save | QMessageBox::Discard
                               | QMessageBox::Cancel);
    if (ret == QMessageBox::Save)
    {
        on_action_Save_triggered();
        return true;
    }
    else if (ret == QMessageBox::Cancel) return false;
    return true;
}
```

Не забудьте дописать в заголовочный файл, в секцию private:

```
    bool maybeSave();
```

9. Давайте еще добавим простенькую функцию создания нового файла. Для этого повторите пункт 5, но только для действия action_New.

Дописываемый код будет выглядеть следующим образом:

```
    void MainWindow::on_action_New_triggered()
    {
        if (maybeSave())
        {
            ui->textEdit->clear();
            setCurrentFileName(QString());
        }
    }
```

10. Установите фокус ввода на текстовое поле, для этого напишите в конструкторе:

```
    setCentralWidget(ui->textEdit);
    ui->textEdit->setFocus();
```

На данном этапе мы можем открывать обычные текстовые файлы, их редактировать и сохранять (рис. 3.13).

Самостоятельная работа

Сделайте так, чтобы по умолчанию при запуске редактора открывался какой-нибудь файл.

Рисунок 3.13 – Результат – простой текстовый редактор без поддержки форматирования

Напоследок, чтобы украсить – подключите к вашему ПО иконку, как для главного окна, так и для исполняемого файла (см. подразд. 2.3).

3.2 Добавление средств форматирования в текстовый редактор

3.2.1 Создание средств форматирования

Добавление новых команд меню:

– создайте меню *Правка* с пунктами *Отменить*, *Повторить*, далее поставьте разделитель и пункты *Вырезать*, *Копировать*, *Вставить*, а также добавьте снова через разделитель пункты *Найти* и *Заменить*;

– создайте меню *Формат* с пунктами *Жирный*, *Курсив*, *Подчеркнутый* и через разделитель *Влево*, *По центру*, *Вправо*, *По ширине*;

– создайте меню *Справка* с пунктом *О себе – великом творце этой прекрасной программы*.

Модификация действий:

– дайте им осмысленные имена:

а) для операций правки: action_Undo, action_Redo; action_Cut, action_Copy, action_Paste; action_Find, action_Replace;

б) для операций форматирования: action_Bold, action_Italic, action_Underline; action_Left, action_Center, action_Right, action_Justify;

– задайте им горячие клавиши:

а) для операций правки: Ctrl+Z, Ctrl+Y; Ctrl+X, Ctrl+C, Ctrl+V; Ctrl+F, Ctrl+H;

б) для операций форматирования: Ctrl+B, Ctrl+I, Ctrl+U; Ctrl+L, Ctrl+E, Ctrl+R, Ctrl+J;

– действиям из меню «Формат» задайте свойство checkable либо поставьте галочку – триггерное;

– установите для каждого из действий свою иконку: на сей раз возьмите недостающие из «<Путь установки Qt>\ <№ версии Qt>\ \mingw492_32\examples\widgets\richtext\textedit\images».

3.2.2 Работа с панелью инструментов

Когда вы попробуете добавить что-либо на панель инструментов, у вас, скорее всего, возникнут некоторые сложности, т.к. контекстное меню не позволяет что-либо добавить, кроме объекта под названием *Разделитель*.

Все достаточно просто: выделите желаемое действие (в редакторе действий) и перетащите его на панель инструментов – оно появится на панели инструментов (рис. 3.14). Ничего более прописывать не надо: к этим кнопкам автоматически подключаются уже созданные вами процедуры. Осталось лишь добавить разделители, расположив их так же, как и в меню.

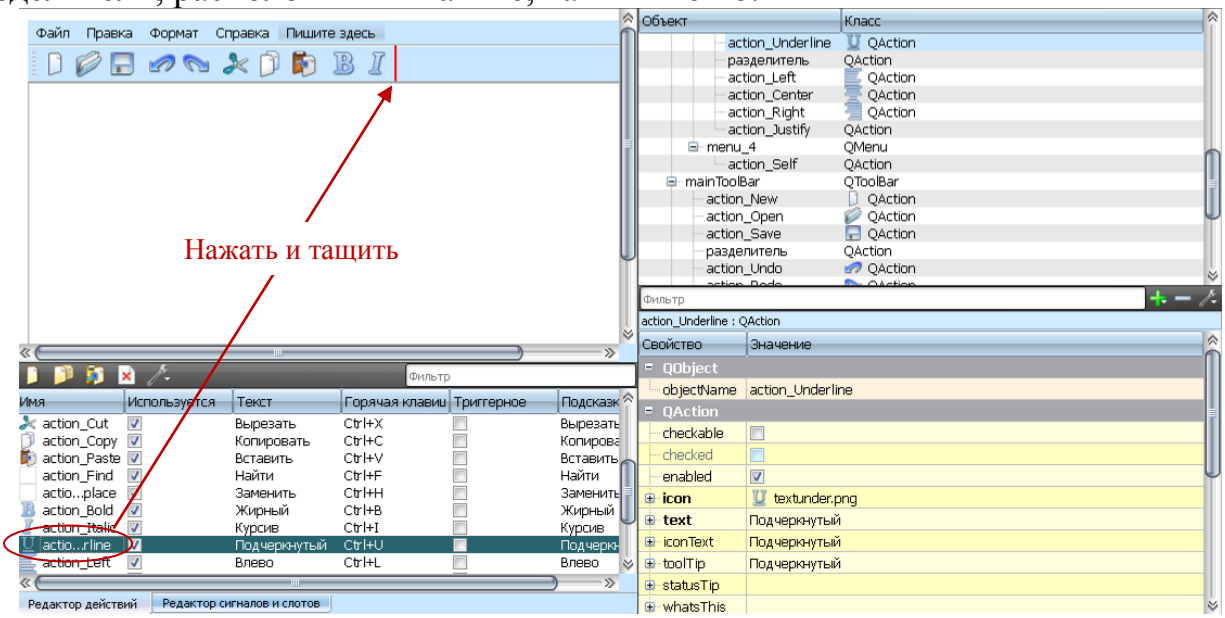


Рисунок 3.14 – Добавление действий на панель инструментов

3.2.3 Наполнение действий по форматированию функциональностью

Начнем с самых простых: *Отменить*, *Повторить*, *Вырезать*, *Копировать*, *Вставить*. Если вы были внимательны, то заметили, что «магические» комбинации клавиш, связанные с этими операциями, уже работают. Не работают лишь кнопки – подключим их.

Следует повторить действия, которые вы осуществляли для подключения кнопки *Выход*: откройте вкладку **редактор сигналов и слотов** и, например, последовательно выберите: action_Cut, triggered(), textEdit, cut(). Нечто подобное

совершите и для остальных действий. Вот так это все выглядело бы в программном коде:

```
    connect(ui->action_Undo,  SIGNAL(triggered()), ui->textEdit, SLOT(undo()) );
    connect(ui->action_Redo,  SIGNAL(triggered()), ui->textEdit, SLOT(redo()) );
    connect(ui->action_Cut,   SIGNAL(triggered()), ui->textEdit, SLOT(cut())  );
    connect(ui->action_Copy,  SIGNAL(triggered()), ui->textEdit, SLOT(copy()) );
    connect(ui->action_Paste, SIGNAL(triggered()), ui->textEdit, SLOT(paste()));
```

Пожалуй, также следует проверять: должны ли эти кнопки быть доступны:

```
    connect(ui->textEdit, SIGNAL(copyAvailable(bool)), ui->action_Cut,
SLOT(setEnabled(bool)));
    connect(ui->textEdit, SIGNAL(copyAvailable(bool)), ui->action_Copy,
SLOT(setEnabled(bool)));
    connect(ui->textEdit, SIGNAL(undoAvailable(bool)), ui->action_Undo,
SLOT(setEnabled(bool)));
    connect(ui->textEdit, SIGNAL(redoAvailable(bool)), ui->action_Redo,
SLOT(setEnabled(bool)));
```

Теперь уберем галочку *enable* с действий *Отменить*, *Повторить*, *Вырезать*, *Копировать*, дабы они не были доступны нам изначально.

А для действия *Вставить* пропишите в конструкторе:

```
#ifndef QT_NO_CLIPBOARD
    if (const QMimeData *md = QApplication::clipboard()->mimeData())
        actionPaste->setEnabled(md->hasText());
     #endif
```

Правда, чтобы этот код заработал, следует подключить:

```
#include <QClipboard>
#include <QMimeData>
```

Все. Первые пять инструментов мы подключили. Протестируйте приложение.

Теперь приступим к работе над пунктами *Жирный*, *Курсив*, *Подчеркнутый*. Создадим ряд своих процедур: выберем действие *action_Bold*, кликнем правой кнопкой мыши и выберем *Перейти к слоту…* То же самое повторите для остальных действий.

```
void MainWindow::on_action_Bold_triggered()
{
    QTextCharFormat fmt;
    fmt.setFontWeight(ui->action_Bold->isChecked() ? QFont::Bold : QFont::Normal);
    mergeFormatOnWordOrSelection(fmt);
}
void MainWindow::on_action_Italic_triggered()
{
    QTextCharFormat fmt;
    fmt.setFontItalic(ui->action_Italic->isChecked());
    mergeFormatOnWordOrSelection(fmt);
}
void MainWindow::on_action_Underline_triggered()
{
    QTextCharFormat fmt;
    fmt.setFontUnderline(ui->action_Underline->isChecked());
    mergeFormatOnWordOrSelection(fmt);
}
```

Как видно, стоит дописать еще одну процедуру:

```
void MainWindow::mergeFormatOnWordOrSelection(const QTextCharFormat &format)
{
    QTextCursor cursor = ui->textEdit->textCursor();
    if (!cursor.hasSelection()) cursor.select(QTextCursor::WordUnderCursor);
    cursor.mergeCharFormat(format);
    ui->textEdit->mergeCurrentCharFormat(format);
}
```

Следует описать ее в классе и подключить необходимые модули:

```
#include <QTextCharFormat>
void mergeFormatOnWordOrSelection(const QTextCharFormat &format);
```

Это интересно: *попробуйте также вставить в процедуру подчеркивания строчку:*

```
fmt.setFontOverline(ui->action_Underline->isChecked());
```

Запустите. Форматирование будет работать, правда, не совсем правильно. Следует установить для этих действий свойство триггерности (рис. 3.15).

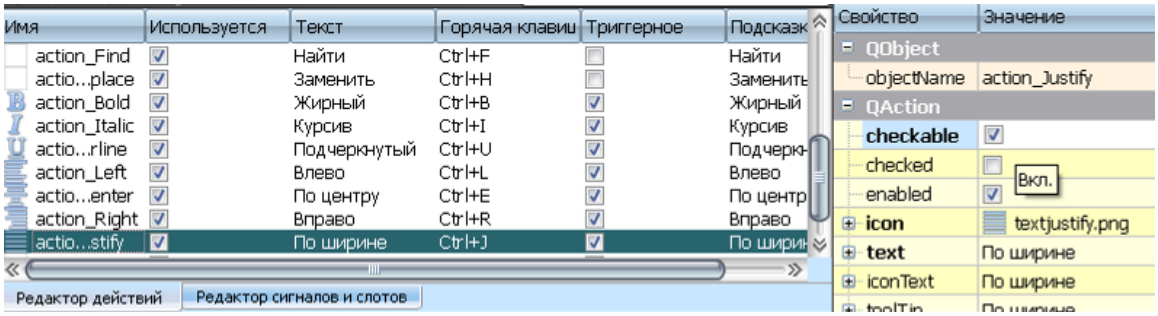


Рисунок 3.15 – Добавление действиям свойства триггерности

Теперь займемся пунктами *Влево*, *По центру*, *Вправо*, *По ширине*. Повторите действия, описанные выше, но уже для этих пунктов.

```
void MainWindow::on_action_Left_triggered()
{
    ui->textEdit->setAlignment(Qt::AlignLeft | Qt::AlignAbsolute);
}
void MainWindow::on_action_Center_triggered()
{
    ui->textEdit->setAlignment(Qt::AlignHCenter);
}
void MainWindow::on_action_Right_triggered()
{
    ui->textEdit->setAlignment(Qt::AlignRight | Qt::AlignAbsolute);
}
void MainWindow::on_action_Justify_triggered()
{
    ui->textEdit->setAlignment(Qt::AlignJustify);
}
```

Этим действиям, как и предыдущим, также следует задать свойство – триггерное и дописать процедуру, которая проверяет и устанавливает это свойство в зависимости от выбранного выравнивания:

```
void MainWindow::alignmentChanged(Qt::Alignment a)
{
    if (a & Qt::AlignLeft) {
        ui->action_Left->setChecked(true);
        ui->action_Center->setChecked(false);
```

```
        ui->action_Right->setChecked(false);
        ui->action_Justify->setChecked(false);
    } else if (a & Qt::AlignHCenter) {
        ui->action_Left->setChecked(false);
        ui->action_Center->setChecked(true);
        ui->action_Right->setChecked(false);
        ui->action_Justify->setChecked(false);
    } else if (a & Qt::AlignRight) {
        ui->action_Left->setChecked(false);
        ui->action_Center->setChecked(false);
        ui->action_Right->setChecked(true);
        ui->action_Justify->setChecked(false);
    } else if (a & Qt::AlignJustify) {
        ui->action_Left->setChecked(false);
        ui->action_Center->setChecked(false);
        ui->action_Right->setChecked(false);
        ui->action_Justify->setChecked(true);
    }
}
```

Процедуру следует вызвать в конструкторе:

```
    alignmentChanged(ui->textEdit->alignment());
```

а также не забыть прописать в классе:

```
    void alignmentChanged(Qt::Alignment a);
```

еще допишем ее во все 4 действия для работы с выравниванием, к примеру:

```
void MainWindow::on_action_Left_triggered()
{
    ui->textEdit->setAlignment(Qt::AlignLeft | Qt::AlignAbsolute);
    alignmentChanged(ui->textEdit->alignment());
}
```

Как вы, должно быть, догадываетесь это еще не все необходимые процедуры.

Следить за состоянием триггерных действий мы должны динамически в зависимости от позиции курсора и текста под ним:

```
void MainWindow::currentCharFormatChanged(const QTextCharFormat &format)
{
    fontChanged(format.font());
//    colorChanged(format.foreground().color());
}
void MainWindow::cursorPositionChanged()
{
    alignmentChanged(ui->textEdit->alignment());
}
void MainWindow::fontChanged(const QFont &f)
{
    ui->action_Bold->setChecked(f.bold());
    ui->action_Italic->setChecked(f.italic());
    ui->action_Underline->setChecked(f.underline());
}
```

В конструкторе необходимо дописать:

```
connect(ui->textEdit, SIGNAL(currentCharFormatChanged(QTextCharFormat)), this,
SLOT(currentCharFormatChanged(QTextCharFormat)));
    connect(ui->textEdit, SIGNAL(cursorPositionChanged()), this,
SLOT(cursorPositionChanged()));
```

Также сделаем вызовы в конструкторе для шрифта и цвета:

```
    fontChanged(ui->textEdit->font());
//    colorChanged(ui->textEdit->textColor());
```

Естественно, процедуры необходимо описать в классе:

```
// (в секции private)
void fontChanged(const QFont &f);
// (в секции private slots):
    void currentCharFormatChanged(const QTextCharFormat &format);
    void cursorPositionChanged();
```

Самостоятельная работа

Наполните программным кодом каждый из оставшихся пунктов.

Для поиска и замены – подключите созданный в лаб. работе № 2 диалог.

А также в меню Справка реализуйте возможность вызова окошка с данными о разработчиках, т.е. о таких способных студентах, как вы. Реализовать окошко вы можете как через Qt Designer, так и посредством создания его программным кодом, а можете просто использовать QMessageBox – на ваш выбор.

Установите иконку для вашего приложения (если вы этого еще не сделали).

Это важно: *модальным называется диалог, который "забирает" на себя фокус ввода, не позволяя переключиться на другие окна до тех пор, пока он не будет закрыт. В немодальном режиме диалог выводится на экран с помощью метода show(). Для модального режима предназначен метод exec().*

3.3 Дополнительные возможности

3.3.1 Цвет

Добавим возможность разукрасить наш текст всеми цветами палитры. Для этого создайте action_Color и добавьте его в меню *Формат*, а также через разделитель на панель инструментов. Создайте реакцию на его активизацию:

```
void MainWindow::on_action_Color_triggered()
{
    QColor col = QColorDialog::getColor(ui->textEdit->textColor(), this);
    if (!col.isValid()) return;
    QTextCharFormat fmt;
    fmt.setForeground(col);
    mergeFormatOnWordOrSelection(fmt);
    colorChanged(col);
}
```

Здесь стоит подключить модуль:

```
#include <QColorDialog>
```

А также можете обратить внимание на метод colorChanged():

```
void MainWindow::colorChanged(const QColor &c)
{
    QPixmap pix(16, 16);
    pix.fill(c);
    ui->action_Color->setIcon(pix);
}
```

Если мы допишем этот код, то иконка будет меняться динамически в зависимости от указанного цвета. И не забудьте раскомментировать строчки с вызовом процедуры `colorChanged` в конструкторе и процедуре `currentCharFormatChanged`, а также добавить в описание класса:

```
    void colorChanged(const QColor &c);
```

3.3.2 Работа с принтером

Для начала необходимо дописать в файл проекта:

```
qtHaveModule(printsupport): QT += printsupport
```

Затем подключить библиотеки для работы с принтером:

```
#ifndef QT_NO_PRINTER
#include <QPrintDialog>
#include <QPrinter>
#include <QPrintPreviewDialog>
#endif
```

Также добавьте пункты *Печать* и *Предварительный просмотр* в меню *Файл*, а также код для подключения и работы с принтером:

```
void filePrint()
{
#ifndef QT_NO_PRINTER
    QPrinter printer(QPrinter::HighResolution);
    QPrintDialog *dlg = new QPrintDialog(&printer, this);
    if (textEdit->textCursor().hasSelection())
        dlg->addEnabledOption(QAbstractPrintDialog::PrintSelection);
    dlg->setWindowTitle(tr("Print Document"));
    if (dlg->exec() == QDialog::Accepted) textEdit->print(&printer);
    delete dlg;
#endif
}
void filePrintPreview()
{
#ifndef QT_NO_PRINTER
    QPrinter printer(QPrinter::HighResolution);
    QPrintPreviewDialog preview(&printer, this);
    connect(&preview, SIGNAL(paintRequested(QPrinter*)),
SLOT(printPreview(QPrinter*)));
    preview.exec();
#endif
}
void printPreview(QPrinter *printer)
{
#ifdef QT_NO_PRINTER
    Q_UNUSED(printer);
#else
    textEdit->print(printer);
#endif
}
```

3.3.3 Добавление стилей, шрифтов и их размеров

Для начала добавим еще одну панель инструментов (рис. 3.16).

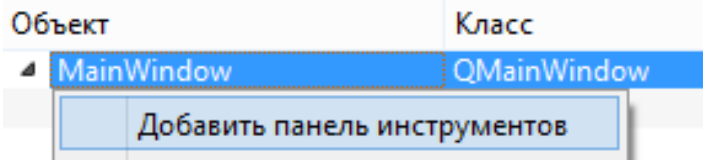


Рисунок 3.16 – Добавление новой панели инструментов

В нее добавим ComboBox, это возможно сделать лишь с помощью программного кода:

```
    comboStyle = new QComboBox(ui->toolBar);
    ui->toolBar->addWidget(comboStyle);
    comboStyle->addItem("Standard");
    comboStyle->addItem("Bullet List (Disc)");
    comboStyle->addItem("Bullet List (Circle)");
    comboStyle->addItem("Bullet List (Square)");
    comboStyle->addItem("Ordered List (Decimal)");
    comboStyle->addItem("Ordered List (Alpha lower)");
    comboStyle->addItem("Ordered List (Alpha upper)");
    comboStyle->addItem("Ordered List (Roman lower)");
    comboStyle->addItem("Ordered List (Roman upper)");
    connect(comboStyle, SIGNAL(activated(int)),this, SLOT(textStyle(int)));
```

Давайте сразу же добавим возможность работать со шрифтами и их размерами:

```
    comboFont = new QFontComboBox(ui->toolBar);
    ui->toolBar->addWidget(comboFont);
    connect(comboFont, SIGNAL(activated(QString)), this,
SLOT(textFamily(QString)));

    comboSize = new QComboBox(ui->toolBar);
    comboSize->setObjectName("comboSize");
    ui->toolBar->addWidget(comboSize);
    comboSize->setEditable(true);
    QFontDatabase db;
    foreach(int size, db.standardSizes()) comboSize-
>addItem(QString::number(size));
    connect(comboSize, SIGNAL(activated(QString)), this, SLOT(textSize(QString)));
    comboSize->setCurrentIndex(comboSize-
>findText(QString::number(QApplication::font().pointSize())));
```

Не забудьте подключить в описание класса необходимые переменные.

Все это теперь должно отобразиться на экране, но пока что еще не будет функционировать.

```
void MainWindow::textFamily(const QString &f)
{
    QTextCharFormat fmt;
    fmt.setFontFamily(f);
    mergeFormatOnWordOrSelection(fmt);
}
void MainWindow::textSize(const QString &p)
{
    qreal pointSize = p.toFloat();
    if (p.toFloat() > 0) {
        QTextCharFormat fmt;
        fmt.setFontPointSize(pointSize);
        mergeFormatOnWordOrSelection(fmt);
    }
}
void MainWindow::textStyle(int styleIndex)
{
    QTextCursor cursor = ui->textEdit->textCursor();
    if (styleIndex != 0) {
        QTextListFormat::Style style = QTextListFormat::ListDisc;
        switch (styleIndex) {
            default:
            case 1:
                style = QTextListFormat::ListDisc;
                break;
            case 2:
```

```
                style = QTextListFormat::ListCircle;
                break;
            case 3:
                style = QTextListFormat::ListSquare;
                break;
            case 4:
                style = QTextListFormat::ListDecimal;
                break;
            case 5:
                style = QTextListFormat::ListLowerAlpha;
                break;
            case 6:
                style = QTextListFormat::ListUpperAlpha;
                break;
            case 7:
                style = QTextListFormat::ListLowerRoman;
                break;
            case 8:
                style = QTextListFormat::ListUpperRoman;
                break;
        }
        cursor.beginEditBlock();
        QTextBlockFormat blockFmt = cursor.blockFormat();
        QTextListFormat listFmt;
        if (cursor.currentList()) {
            //listFmt = cursor.currentList()->format();
        } else {
            listFmt.setIndent(blockFmt.indent() + 1);
            blockFmt.setIndent(0);
            cursor.setBlockFormat(blockFmt);
        }
        listFmt.setStyle(style);
        cursor.createList(listFmt);
        cursor.endEditBlock();
    } else {
        QTextBlockFormat bfmt;
        bfmt.setObjectIndex(-1);
        cursor.mergeBlockFormat(bfmt);
    }
}
```

Остается лишь подключить необходимые библиотеки и прописать процедуры в заголовочном файле класса.

3.3.4 Диалог поиска

Вы уже должны были подключить диалог поиска, но пока он у вас не функционирует.

Поиск должен стартовать по нажатию на кнопку «Найти». Она должна быть недоступна, если в поле для поиска ничего не введено (см. лаб. работу № 2).

Для того, чтобы работать с текстовым полем из другого класса (класса диалога поиска), следует создать переменную такого же класса (QTextEdit) и передать в нее указатель на оригинал:

```
void Find_Dialog::SetEditor(QTextEdit *ed)
{
    textEdit2=ed;
}
```

Запустить ее следует после создания диалога поиска.

Самостоятельная работа

Ваша задача состоит в том, чтобы диалог поиска искал заданные слова.

В качестве примера используйте программы «Extension example» и «Text Finder example».

Также проверьте корректность работы всех возможностей вашего текстового редактора. По внешнему виду должно получиться нечто похожее на рис. 3.17.

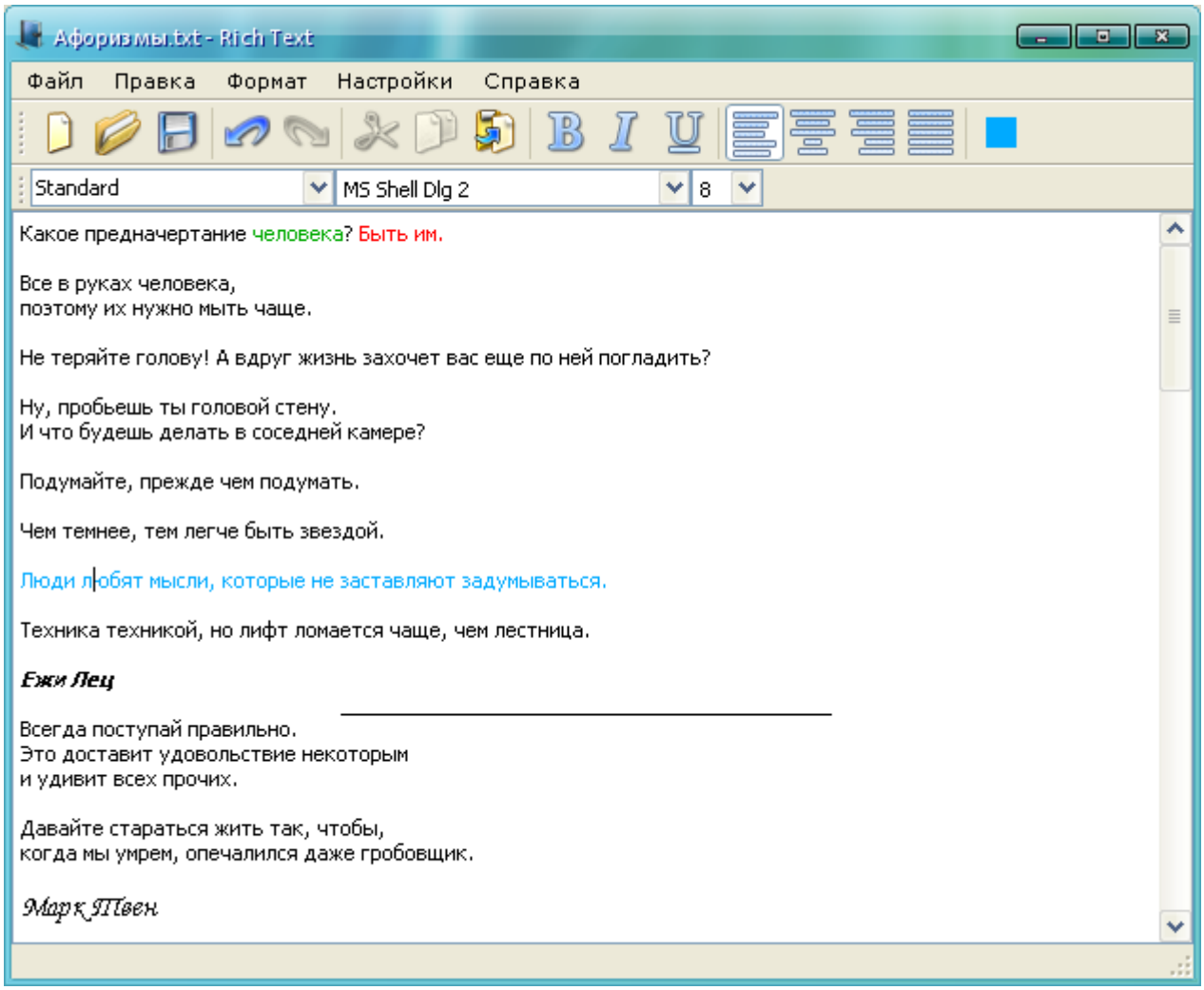


Рисунок 3.17 – Вид полностью работоспособного текстового редактора

Контрольные вопросы

1. Какие стандартные процедуры есть в Qt для работы с файлами?
2. Как подключить файл ресурсов?
3. Как добавить действия на панель инструментов?
4. Как заставить иконки становиться активными при перемещении курсора? Например, если поставить курсор в текст, написанный курсивом, то кнопка «Курсив» будет отрисована нажатой, как это обеспечивается?

4 РАБОТА СО СТИЛЯМИ И СОЗДАНИЕ ПЕРЕНОСИМОГО ПРИЛОЖЕНИЯ

Цель работы *– научиться работе со стилями; освоить навыки создания полноценного переносимого кросс-платформенного программного продукта с помощью Qt.*

Теоретическое введение

Один из немаловажных аспектов при программировании с использованием библиотеки Qt – это управление видом и поведением приложения (look & feel). Ввиду того, что Qt-приложения создаются для большого числа платформ, необходимо, чтобы их внешний вид не выбивался из колеи при запуске в какой-либо операционной системе, и не создавалось впечатление того, что программа "чужая" на этой платформе.

Qt предоставляет специальные классы стилей, позволяющие изменять внешний вид и поведение для любых классов виджетов. Стили программы можно изменять даже в процессе ее работы, а это значит, что пользователю можно предоставить в меню целый список стилей, чтобы он смог выбрать оптимальный для себя. Стили можно создавать самому или использовать уже готовые, встроенные в библиотеку Qt.

Возможность реализации и использования классов стилей, не зависящих от кода программы, дает большую свободу, позволяющую разделить разработку проекта на команды, которые могут работать независимо друг от друга над кодом самой программы и над ее дизайном. В контексте Qt стиль – это класс, унаследованный от **QStyle** и реализующий возможности для рисования рамок, кнопок, растровых изображений и т. д. Он делает каждый виджет ответственным за свое отображение, что повышает скорость отображения и гибкость.

Теперь поговорим о переносе вашего ПО на другие компьютеры.

Допустим, вы написали приложение, и оно успешно компилируется и запускается в Qt Creator'e. А вам нужно выпустить готовое приложение, которое можно будет запускать отдельно, как любую другую программу. Как это сделать?

Для этого необходимо откомпилировать проект в режиме релиза. И потом вместе с вашим приложением нужно прикладывать файлы:

– QtCore5.dll;
– libgcc_s_dw2-1.dll;
– libwinpthread-1.dll;
– libstdc++-6.dll;
– QtGui5.dll;
– Qt5Widgets.dll.

Остальное зависит от того, какие еще опции вы укажете в файле .pro.

Это интересно: *часто возникает вопрос – откуда брать эти библиотеки?*
Ответ: из каталога, где установлен Qt, например: «D:\Qt\5.5\mingw492_32\bin».

Это интересно: *если собрано в режиме релиза, то брать надо библиотеки без d на конце имени файла: Qt5Core.dll и т.д., если собрано в debug режиме, то Qt5Cored.dll и т.д.*

Или можно перекомпилировать сам Qt по инструкции Отучение Qt от *.dll, но тогда сам exe'шник будет размером больше 10 МБ... Так как для этого необходимо переустанавливать Qt, то делать этого мы не будем.

4.1 Работа со стилями

Это важно: *в этой и следующей лабораторной вы должны работать над проектом, разработанным в лабораторной № 3.*

1. Хорошее приложение должно обладать возможностью изменения стилей.

Создадим пункт меню *Настройка* в разработанном вами текстовом редакторе, а в нем же – пункт *Стили* и добавим вышеназванные стили подпунктами.

Это важно: *стилей в Qt предопределено несколько:*
- *Windows;*
- *Fusion;*
- *а также в зависимости от типа ОС: WindowsXP, WindowsVista, Gtk, Macintosh.*

Работа со стилями и настройками приложения требует подключения ряда библиотек:

```
    #include <QStyleFactory>
    #include <QSettings>
```

Например, так мы установим стиль Fusion, подобным образом можно установить другие стили:

```
    QApplication::setStyle(QStyleFactory::create("Fusion"));
```

Передача объекта стиля в метод setstyle() приводит к тому, что этот метод сначала удаляет, с помощью оператора delete , старый объект стиля. Поэтому можно создавать объекты стиля непосредственно в самом методе setstyle() и не создавать дополнительных указателей на эти объекты.

2. Создадим соответствующие действия и наполним их, а также продублируем эти действия на панели инструментов (в конструкторе):

```
    combo_App_Style = new QComboBox;
    combo_App_Style->addItems(QStyleFactory::keys());
    ui->toolBar->addWidget(combo_App_Style);
    connect(combo_App_Style,SIGNAL(activated(const
QString&)),SLOT(slotChangeStyle(const QString&)));

    QString defaultStyle = QApplication::style()->metaObject()->className();
    QRegExp regExp(".(.*)\\+?Style"); //QWindowsVistaStyle -> WindowsVista
    if (regExp.exactMatch(defaultStyle)) defaultStyle = regExp.cap(1);
    combo_App_Style->setCurrentIndex(combo_App_Style->findText(defaultStyle,
Qt::MatchContains));
```

Вам следует описать соответствующий слот и его реализацию.

Таблица 4.1 – Некоторые из состояний виджетов

Обозначение	Описание
:checked	Активировано
:closed	Виджет находится в закрытом либо свернутом состоянии
:disabled	Виджет недоступен
:enabled	Виджет доступен
:focus	Виджет находится в фокусе ввода
:hover	Указатель мыши находится над виджетом
:indeterminate	Кнопка находится в промежуточном неопределенном состоянии
:off	Выключено (для виджетов, которые могут быть в фиксированном состоянии нажато/не нажато)
:on	Включено (для виджетов, которые могут быть в фиксированном состоянии нажато/не нажато)
:open	Виджет находится в открытом или развернутом состоянии
:pressed	Виджет был нажат мышью
:unchecked	Деактивировано

Для своего собственного стиля вы можете подключить файл с расширением .qss, например, такого содержания:

```
.QWidget {
   background-color: beige;
}
QPushButton {
    background-color: palegoldenrod;
    border-width: 2px;
    border-color: darkkhaki;
    border-style: solid;
    border-radius: 5;
    padding: 3px;
    min-width: 9ex;
    min-height: 2.5ex;
}
QPushButton:hover {
   background-color: khaki;
}
QPushButton:pressed {
    padding-left: 5px;
    padding-top: 5px;
    background-color: #d0d67c;
}
QComboBox, QLineEdit, QSpinBox, QTextEdit, QListView {
    background-color: cornsilk;
    selection-color: #0a214c;
    selection-background-color: #C19A6B;
}
```

Чтобы подключить этот стиль, достаточно написать следующий код (где mystyle.qss – это название вашего файла со стилями, его следует перенести в папку с исполняемым файлом):

```
    QFile file(qApp->applicationDirPath()+"/mystyle.qss");
    file.open(QFile::ReadOnly);
    QString strCSS = QLatin1String(file.readAll());
    qApp->setStyleSheet(strCSS);
```

Если кардинальных изменений в стиле не предполагается, то достаточно:

```
    qApp->setStyleSheet("QPushButton::hover {background-color: blue}");
```

3. Qt предоставляет возможность хранения информации о конфигурации приложений, необходимой для того, чтобы дать пользователю права настраивать приложение под себя. Данные настроек приложения – это совокупность ключей и их значений. Ключи – это значения строкового типа, состоящие из подстрок, разделенных символом /. Значения могут иметь тип, поддерживаемый классом QVariant. Все значения можно читать методом value() и записывать методом setValue().

Работа с конфигурацией предполагает изменение вида конструктора:

```
MainWindow::MainWindow(QWidget *parent) : // QSettings::NativeFormat
    QMainWindow(parent),    //- для Windows - реестр, unix - текстовый файл
    ui(new Ui::MainWindow) // QSettings::IniFormat - хранить в файле ini
    , settings(QSettings::NativeFormat, QSettings::SystemScope, "KhAI", "QReader")
// название вашей компании и приложения
```

и добавление переменной, хранящей настройки, в описание класса:

```
    QSettings settings;
```

4. Давайте научим приложение считывать при старте (в конструкторе) и запоминать при закрытии (в деструкторе) свои размеры, а также количество запусков программы:

```
void MainWindow::readSettings()
{
    settings.beginGroup("/Settings");
    int app_Width = settings.value("/width", width()).toInt(); // Ширина
    int app_Height = settings.value("/height", height()).toInt(); // Высота
    app_Counter = settings.value("/counter", 1).toInt(); // Кол-во запусков
    QString str = tr("This program has been started ") +
QString().setNum(app_Counter++) + tr(" times");
    statusBar()->showMessage(str, 3000);
    this->resize(app_Width, app_Height);
    settings.endGroup();
}
void MainWindow::writeSettings()
{
    settings.beginGroup("/Settings");
    settings.setValue("/counter", app_Counter);
    settings.setValue("/width", width());
    settings.setValue("/height", height());
    settings.endGroup();
}
```

5. Сделайте так, чтобы ваше приложение можно было бы запустить простым нажатием на исполняемый файл (.exe).

Самостоятельная работа

Вам всего лишь необходимо создать и подключить свой собственный стиль.

Измените хотя бы один элемент.

В качестве примера рассмотрите стандартное приложение Styles и книгу Шлее «Qt 5.3. Профессиональное программирование на C++», стр.382.

Контрольные вопросы

1. Какие средства по управлению стилями есть в Qt?
2. Какие файлы необходимы для автономной работы приложения?
3. Что такое регулярные выражения и как они работают?

5 ИНТЕРНАЦИОНАЛИЗАЦИЯ ПРОГРАММ В QT

Цель работы *– создание мультиязычного кросс-платформенного программного продукта.*

Теоретическое введение

Кодировка

В этой лабораторной работе мы постараемся показать, как сделать так, чтобы символы кириллицы корректно отображались на экране. Это не так сложно, да и родные буковки на экране видеть приятнее.

Для решения этой задачи мы будем использовать кодеки (специальные объекты для перекодировки строк). Соответствующий класс *QTextCodec* определён в заголовочном файле с тем же именем (заметьте – без расширения: заголовочные файлы с расширением .h используются только в старых проектах Qt3). Для указания кодировки, используемой функцией перевода, надо создать соответствующий кодек, указав название используемой кодовой таблицы:

```
QTextCodec* codec = QTextCodec::codecForName("CP1251");
```

и передать его в качестве аргумента методу *setCodecForLocale*:

```
QTextCodec::setCodecForLocale(codec);
```

Поскольку при создании кодека указана конкретная кодовая таблица, то исход компиляции исходных текстов не зависит от системной кодировки той платформы, на которой производится сборка программы: результат будет везде одним и тем же. Но файлы с исходными текстами программ в этом случае нельзя подвергать перекодировке (или после перекодировки требуется изменить название кодовой таблицы, заданной при создании кодека).

Это интересно: *найти примеры программ, использующих Qt Linguist, можно в «<Путь установки Qt>\<№ версии Qt>\mingw492_32\examples\linguist».*

Наиболее предпочтительный метод работы с символами национальных алфавитов связан с использованием специальной функции перевода *tr*, с помощью которой осуществляется интернационализация приложений. Подробнее этот вопрос мы обсудим позже, а пока договоримся все строковые константы, указанные в тексте программы, передавать в качестве параметра функции *tr*.

Пара слов о *tr*. Эта статическая функция является членом всех классов Qt, порождённых от базового класса *QObject*, но если мы собираемся вызвать её в главной программе, а не в каком-либо методе класса, то приходится указывать какой-нибудь подходящий объект, например, *QObject::tr*.

Это интересно. *Для того чтобы осуществить проверку символа на цифру или прописную букву:*

```
if (ch.isDigit() || ch.isUpper()) // если ch это цифра или прописная буква, то…
```

Этот фрагмент кода правильно работает для любых алфавитов, в которых различаются символы верхнего и нижнего регистра, в том числе для латинского, греческого и кириллицы.

Строки и Unicode

Класс Qt QString хранит строковые значения в кодировке Unicode. Каждый символ QString имеет 16-битовый тип QChar, а не 8-битовый тип char. Ниже приведены два способа установки первого символа строки на значение «A»:

```
str[0] = 'A' ;
str[0] = QChar(0x41);
```

Мы можем задать любой символ Unicode с помощью его числового кода. Например, ниже показано, как задается прописная буква «сигма» греческого алфавита («I») и символ валюты евро («€»):

```
str[0] = QChar(0x03A3);
str[0] = QChar(0x20AC);
```

Для получения числового кода символа QChar вызовите для него функцию unicode(). Для получения кода ASCII переменной типа QChar (в виде char) вызовите функцию toLatinl(). Для символов, отсутствующих в кодировке Latin-1, функция toLatinl() возвращает «\0».

Мультиязычность

Если мы хотим иметь многоязыковую версию нашего приложения, мы должны сделать две вещи:

– убедиться, что все строки, которые видит пользователь, проходят через функцию tr();

– заключить все обозначения валюты, клавиши ускорителей в статический метод tr() класса QObject;

– загрузить файл перевода (. qm) при запуске приложения.

Перевод Qt-приложений, которые содержат в себе вызовы tr(), выполняется в три приема:

1. Утилитой lupdate извлекаются все строки из исходного текста приложения.
2. Выполняется перевод строк с помощью утилиты Qt Linguist.
3. С помощью утилиты lrelease создается двоичный файл .qm с переводом, который потом может быть загружен приложением.

Пункты 1 и 3 выполняются разработчиком приложения, пункт 2 – переводчиком. Этот процесс может повторяться неоднократно, в ходе разработки и эксплуатации приложения.

Обычно исходные файлы с переводом (создаваемые утилитой lupdate) имеют расширение .ts. Они записываются в формате XML и потому занимают больше места на диске, чем скомпилированные файлы с переводом .qm.

Одновременно объект QTranslator может работать только с одним файлом перевода (с расширением ".qm").

Это интересно: *.ts означает "translation source" (исходный текст перевода), а .qm – "Qt message".*

5.1 Русификация

Проще всего будет модифицировать уже созданный вами проект Hello, но, если хотите, можете создать новый проект, благо вы это уже умеете.

Кстати, вместо кнопки мы теперь создадим пустое окно (12): первый параметр конструктора указывает на родительский элемент (в данном случае 0 – окно не имеет родителя), а второй – набор битовых флагов, влияющих на внешний вид окна (флаг *Qt::Windows* означает, что элемент будет выглядеть, как окно приложения, т.е. будет иметь строку заголовка с системными кнопками для сворачивания на панель задач, закрытия и т.д.). И укажем заголовок этого окна (13) – «Пустое окно Qt».

```
#include <QApplication>
#include <QMainWindow>
#include <QTextCodec>

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

    QTextCodec* codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);

    QMainWindow* mw = new QMainWindow(0, Qt::Window);
    mw->setWindowTitle(QMainWindow::tr("Пустое окно Qt"));
    mw->resize(400, 300);
    mw->show();

    return app.exec();
}
```

Вот и появилось на экране пустое окошко… (рис. 5.1)

Рисунок 5.1 – Пустое окно Qt с кириллицей

Но, наверное, у вас есть желание набросать в него всяких разных кнопочек и текста. Для этого имеется специальный инструмент визуального программирования *Qt Designer*. Рассмотрением его работы мы и занимались в разделе «Создание графического интерфейса с помощью библиотеки классов Qt».

Впрочем, как вы, наверное, заметили, все графические элементы в Qt можно создавать вручную (так мы уже создали кнопку и окно), но проще и нагляднее применять все же *Qt Designer*.

Кстати, при запуске многие приложения показывают так называемое «предшествующее окно» (Splash Screen). Это окно отображается на время, необходимое для инициализации приложения, и информирует о ходе запуска приложения. Зачастую такое окно используют для маскировки длительного процесса старта программы.

5.2 Qt Linguist. Создание переводимого интерфейса приложения

1. Перед тем, как приступить к выполнению этой части лабораторной работы прочтите раздел «Мультиязычность» теоретической части.

2. В качестве примера рассмотрим процесс перевода приложения QReader, которое было написано нами в процессе выполнения предыдущей лабораторной работы. Оно уже содержит все необходимые вызовы tr(). Откройте его в Qt Creator.

3. Прежде всего, необходимо внести изменения в файл проекта .pro, чтобы указать – какие языки будут поддерживаться приложением. Допустим, что мы собираемся включить поддержку немецкого, французского и английского языков, дополнительно к русскому, тогда необходимо в файл QReader.pro добавить раздел TRANSLATIONS:

```
TRANSLATIONS = QReader_de.ts \
               QReader_fr.ts \
               QReader_en.ts \
               QReader_ru.ts
```

Эти файлы будут созданы при первом запуске утилиты lupdate, а на последующих запусках будут просто дополняться.

Для продолжения работы с переводами необходимо собрать  приложение.

4. Далее запускаем lupdate из меню: *Инструменты* → *Внешние* → *Linguist* → *Обновить переводы* (рис. 5.2).

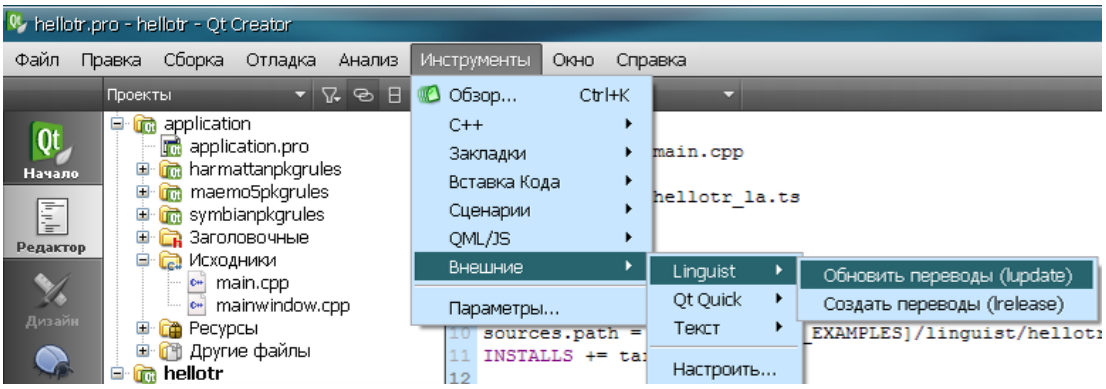


Рисунок 5.2 – Вызов утилиты lupdate

Каждая строка, которая "завернута" в вызов tr(), заносится в .ts, с пустым местом для перевода. Строки, которые находятся в файле .ui, также включаются в исходный файл перевода. Теперь в каталоге должны появиться три файла.

5. Перевод в файлы QReader_de.ts, QReader_fr.ts, QReader_en.ts и QReader_ru.ts добавляется переводчиком, с помощью утилиты Qt Linguist. В данном случае – вами.

Чтобы запустить Qt Linguist, выберите пункт *Qt* → *Qt Linguist* в меню *Пуск* либо найдите исполняемый файл в «<Путь установки Qt>\<№ версии

Qt>\mingw492_32\bin\linguist.exe». Затем, с помощью меню *Файл* → *Открыть*, откройте файл с исходным текстом перевода. Осуществите перевод всех строк с русского на английский.

С левой стороны главного окна Qt Linguist находится список контекстов переводов. Для QReader существуют лишь контексты: "MainWindow" и "Find_Dialog". По центру вверху находится список строк для текущего контекста. Каждая строка отображается вместе с переводом и флагом Done ("Готово") в активном (если уже переведено) или пассивном состоянии (если перевод еще не готов). По центру вводится текст перевода для текущей строки (рис. 5.3).

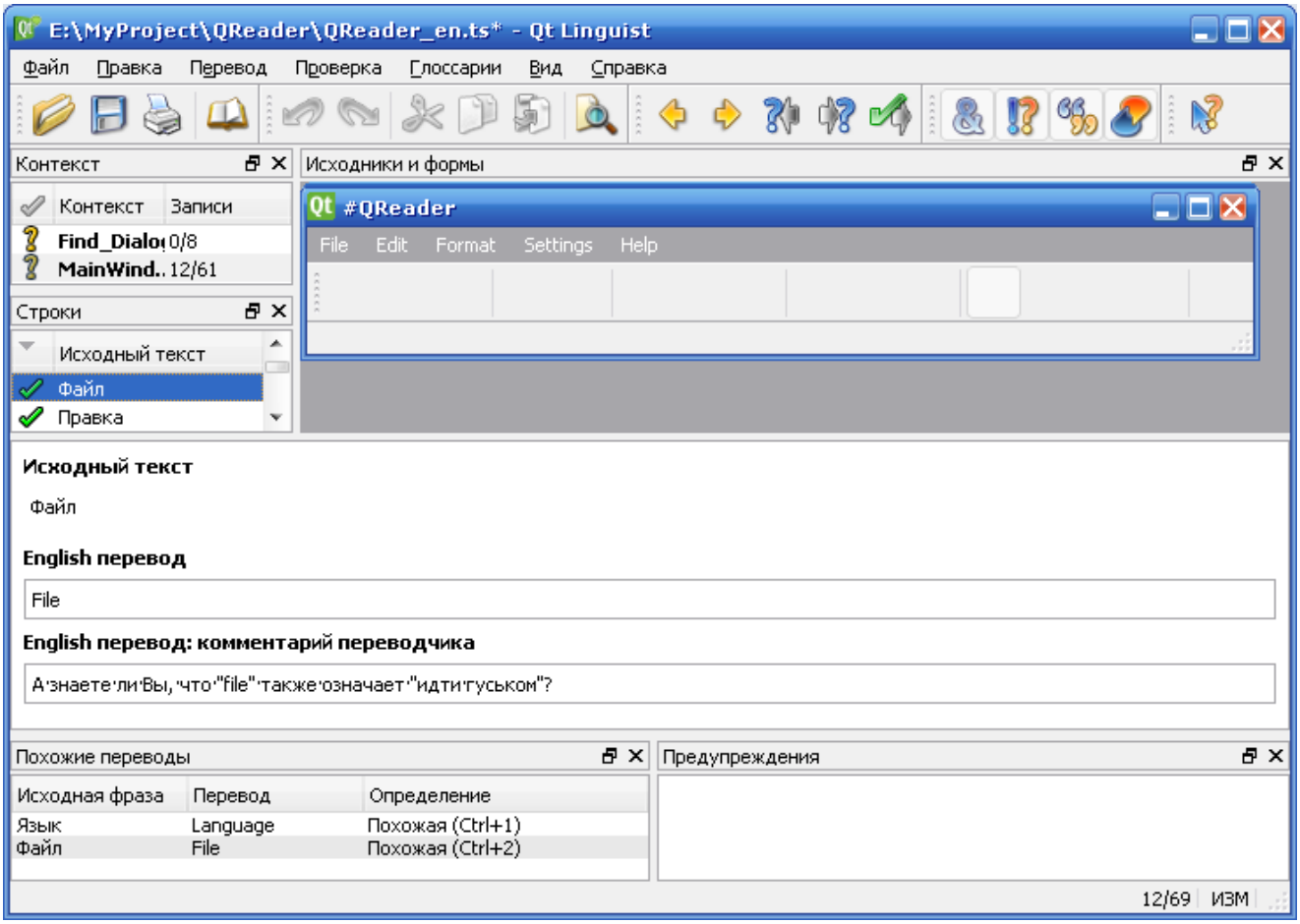


Рисунок 5.3 – Qt Linguist в действии

6. По окончании работы над переводом, файл .ts необходимо преобразовать в файл .qm. Здесь есть два варианта.

7.1. В приложении Qt Linguist выберите пункт меню *File* → *Release* (*Файл* → *Скомпилировать*). Обычно, после перевода нескольких строк, выполняются пробные запуски приложения, с созданным файлом .qm, чтобы визуально оценить качество перевода.

7.2. Запускаем lrelease из меню (*Инструменты* → *Внешние* → *Linguist* → *Создать переводы*), см. рис. 5.2.

Непереведенные строки, при пробном запуске приложения, будут отображаться на языке оригинала. Флаг *Done* никак не используется утилитой

lrelease, он предназначен исключительно для переводчика, чтобы напоминать о том, какие строки имеют окончательный перевод, а какие требуют уточнения.

В случае внесения изменений в исходный код приложения, содержимое файлов .ts может "устареть". Чтобы этого не происходило, нужно всякий раз запускать утилиту lupdate, добавлять перевод для вновь появляющихся строк и пересобирать файлы .qm.

8. Теперь настало время подключить наш перевод посредством программного кода, например, подключим английский язык (в файле main.cpp):

```
    QTranslator translator;
    QString str = QCoreApplication::applicationDirPath() + "/QReader_en";
    translator.load(str);
    a.installTranslator(&translator);
```

Кстати, не забудьте подключить:

```
#include <QTranslator>
```

`QCoreApplication::applicationDirPath()` возвращает путь к исполняемому файлу (.exe), поэтому файлы перевода (.qm) стоит перебросить в новоиспеченную папку debug нашего проекта.

Это важно. *При желании вы можете работать с переводами и из каталога с проектом, правда, это потребует написания специфического кода (в файле проекта):*

```
DEFINES += PRO_FILE_PWD=$$sprintf("\"\\\"%1\\\"\"", $$_PRO_FILE_PWD_)
```

а также необходимо заменить одну строчку в файле main.cpp:

```
    QString str = (QString)PRO_FILE_PWD + "/QReader_en";
```

Это интересно. *Подключив* `#include <QLocale>`*, мы получим доступ к весьма специфичному коду:*

```
translator.load("QReader_" + QLocale::system().name());
```

Эти строчки программного кода подключат к вашей программе родной язык компьютера! В нашем случае это будет русский (`QLocale::system().name()` *– вернет* ru*).*

Для вывода отладочных сообщений в консоль сборки можете воспользоваться следующим кодом:

```
    qDebug("PRO_FILE_PWD: %s", PRO_FILE_PWD);
```

Вывод сообщений в отдельное окошко:

```
    QString str = (QString)PRO_FILE_PWD + "/QReader_en";
   if (!translator.load(str))
       {
        QMessageBox msgBox;
        msgBox.setText(tr("File is not exist: %1.").arg(str));
        msgBox.exec();
    }
```

Следует подключить заголовочный файл:

```
#include <QMessageBox>
```

И определить макрос:

```
#define tr QObject::tr
```

5.3 Динамическое переключение языков

Хорошим тоном считается предоставление пользователю возможности изменить язык интерфейса выбором пункта меню или в диалоговом окне настройки приложения (рис. 5.4).

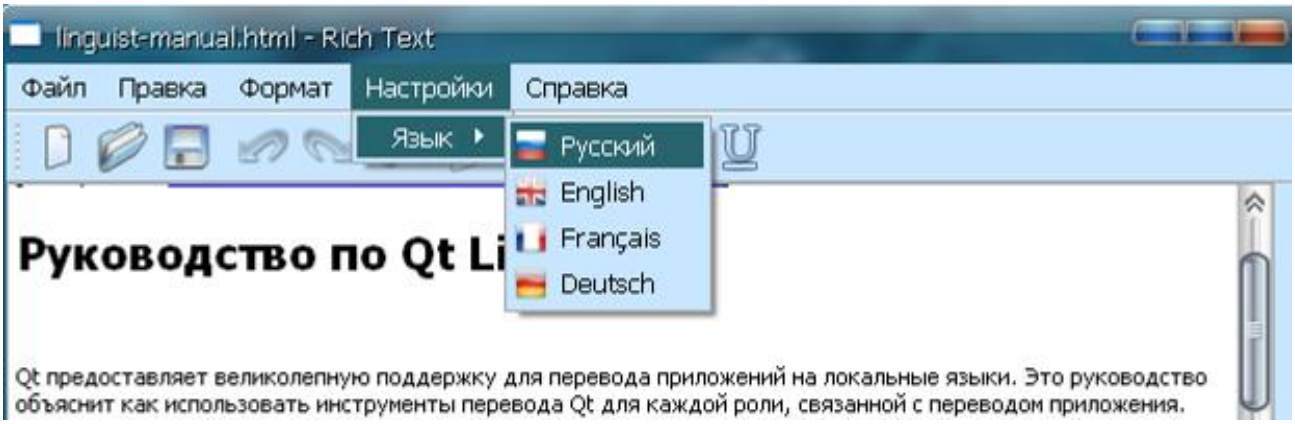


Рисунок 5.4 – Меню выбора языка интерфейса

Пример программного кода:

```
QTranslator translator;
translator.load(qApp->applicationDirPath()+"/QReader_en");
qApp->installTranslator(&translator);
ui->retranslateUi(this);
```

Вернуть русский язык обратно можно так (тогда не нужен QReader_ru.qm):

```
QCoreApplication::removeTranslator(&translator);
ui->retranslateUi(this);
```

Контрольные вопросы

1. Опишите работу с кодировкой в Qt.
2. Как русифицировать консольное приложение?
3. Как сделать программу мультиязычной?

Тест

1. С каким количеством файлов перевода (с расширением ".qm") одновременно может работать объект QTranslator?

– не больше 2^4;
– с любым количеством;
– зависит от настроек;
– с одним файлом.

2. Чтобы приложение можно было интернационализировать, необходимо:

– перевести все строки, используемые в программе, на английский язык;
– установить локаль с помощью класса QLocale;
– заключить все обозначения валюты, клавиши ускорителей в статический метод tr() класса QObject;
– заключить все строки в статический метод tr() класса QObject;
– установить текстовый кодек.

6 РАЗРАБОТКА МЕДИАПЛЕЕРА НА QT

Цель работы *– создание кросс-платформенного программного продукта, способного проигрывать как видео, так и музыку с помощью Qt.*

Теоретическое введение

В основе работы каждого приложения, имеющего пользовательский интерфейс, лежит *обработка событий*.

Механизм сигналов и слотов, по сравнению с событиями, представляет собой механизм более высокого уровня, предназначенный для связи объектов, хотя и то, и другое являются уведомлением о происходящем. Соответственно, когда нам нужно работать с виджетами на более низком уровне, без событий не обойтись.

Есть еще одно отличие сигналов от событий – события обрабатываются лишь одним методом, а сигналы могут обрабатываться неограниченным количеством соединенных с ними слотов.

Qt предоставляет целый ряд классов для различного рода событий: клавиатуры, мыши, таймера и др.

Класс ***QEvent*** является базовым для всех категорий событий. Его объекты содержат информацию о типе произошедшего события. А для каждого типа события имеется целочисленный идентификатор, который устанавливается в конструкторе QEvent и может быть получен при помощи метода QEvent::type().

Класс ***QKeyEvent*** содержит данные о событиях клавиатуры. С его помощью можно получить информацию о клавише, вызвавшей событие, а также ASCII-код отображенного символа.

В методе обработки события с помощью метода QKeyEvent::key() можно определить, какая из клавиш его инициировала.

Когда пользователь набирает что-нибудь на клавиатуре, информацию о нажатых клавишах может принимать только один виджет. Если виджет в этот момент выбран для ввода с клавиатуры, то говорят, что он находится в фокусе. Объект ***QFocusEvent*** передается в методы обработки сообщений focusinEvent() и focusOutEvent().

В объекте класса ***QPaintEvent*** передается информация для перерисовки всего изображения или его части. Событие возникает тогда, когда виджет впервые отображается на экране явным или неявным вызовом метода show(), а также в результате вызова методов repaint() и update ().

Объект класса ***QMouseEvent*** содержит информацию о событии, вызванном действием мыши.

Метод mousePressEvent() вызывается тогда, когда произошло нажатие на одну из кнопок мыши в области виджета. Если нажать кнопку мыши и, не отпуская ее, переместить указатель мыши за пределы виджета, то он будет получать события мыши, пока кнопку не отпустят. При движении мыши станет вызываться метод mouseMoveEvent(), а при отпускании кнопки произойдет вызов метода mouseReleaseEvent().

6.1 Разработка формы медиаплеера

1. Запустите *Qt Creator*. Снова воспользуемся мастером.

Выберите *Файл → Создать файл или проект…*, а там *Приложение → Приложение Qt Widgets*.

2. Дадим ему имя *QPlayer*, укажем месторасположение проекта и нажмем *Далее* (3 раза) и *Завершить*.

3. Займемся созданием и редактированием действий.

3.1. Создайте стандартное меню *Файл* с пунктами *Открыть, Выход*.

Между *Открыть* и *Выход* добавьте разделитель.

Дадим им имена action_Open и action_Quit соответственно, а также добавим к ним горячие комбинации клавиш: Ctrl + O, Ctrl + Q. Для action_Quit (рис. 6.1).

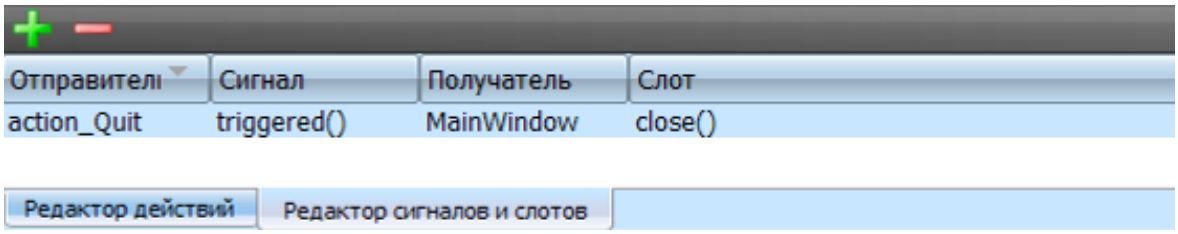


Рисунок 6.1 – Редактирование действия action_Quit

Теперь приступим к action_Open, зададим иконку (предварительно создав и подключив файл ресурсов), бросим на панель инструментов, нажмем правой кнопкой мыши на action_Open в редакторе действий и выберем: *Перейти к слоту…*, далее (рис. 6.2), нажмем *ОК*.

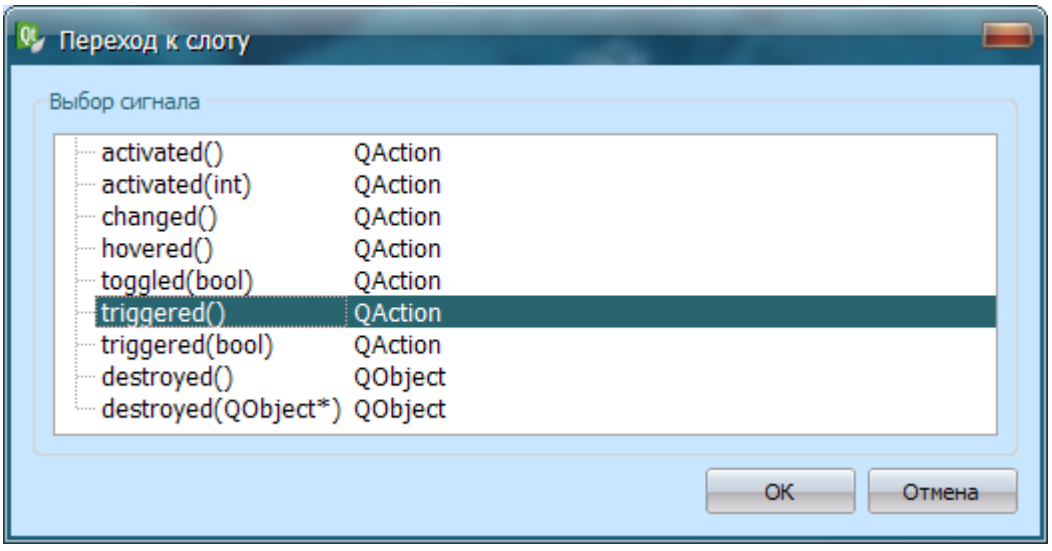


Рисунок 6.2 – Переход к слоту действия

Продолжим наполнять нашу форму содержимым.

3.2. Создайте меню *Навигация*, а в нем пункты: *Воспроизведение*, *Остановить*, *Назад*, *Вперед*. Придайте им соответствующие иконки, переименуйте действия в action_Play, action_Stop, action_Back и action_Forward соответственно, а также задайте им горячие клавиши: Ctrl+P, Ctrl+K, Ctrl+B, Ctrl+F.

Разместите их в панели инструментов.

Заблокируйте действия (свойство **enable**).

3.3. Приложение должно быть красивым: измените заголовок окна на «Самый лучший видеоплеер, созданный САПРовцем» (либо на любой другой по вашему усмотрению) и добавьте иконку к окошку (предварительно подключив ее в файле ресурсов).

4. Работа с панелью инструментов:

– переместите панель инструментов вниз – так будет удобнее;

– выберите в **Инспекторе объектов** *Панель инструментов* (*mainToolBar*) либо просто щелкните по ней на форме, в **Окне свойств** найдите iconSize и установите значения 96 и 48 для ширины и высоты соответственно (рис. 6.3).

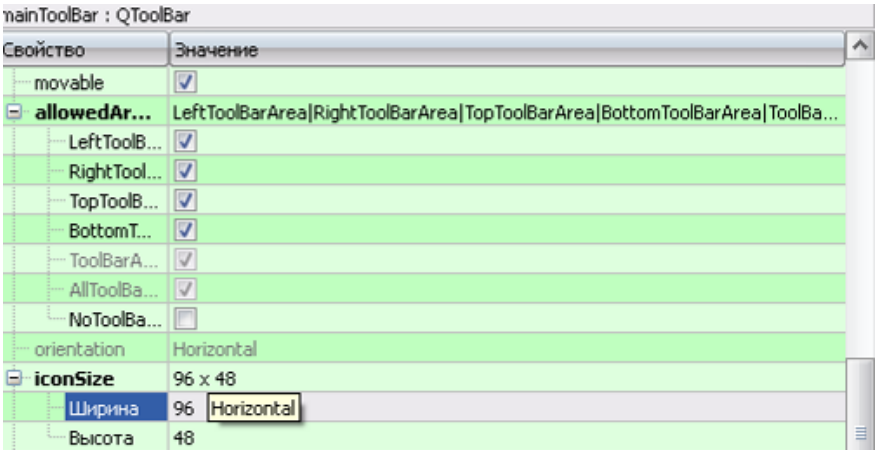


Рисунок 6.3 – Задание размеров иконок

5. Окошко для видео. К сожалению, виджет, воспроизводящий видео, вы не сможете обнаружить среди доступных виджетов в Qt Designer, но мы можем разместить на форме другой виджет, в котором впоследствии и будет размещаться видео – это будет Vertical Layout, добавьте его на форму.

6. В файле проекта (QPlayer.pro) следует добавить:

```
QT+=       multimedia \
           multimediawidgets \
```

6.2 Программирование функционала медиаплеера

1. Начнем с действия action_Open: после нажатия на соответствующую клавишу добавим следующий код (файл mainwindow.cpp):

```
    QString fileName = QFileDialog::getOpenFileName(this, QString("Open music and
video files"),
                                qsDataPath, tr( "Media files (*.vob *.mpg *.mpeg
```

```
*.mp4 *.wmv *.avi) " ) );

    if (!fileName.isEmpty()) {
        player->setMedia(QUrl::fromLocalFile(fileName));
        ui->action_Play->setEnabled(true);
    }
```

Как видно, следует описать еще несколько переменных (файл mainwindow.h, секция private):

```
QString qsDataPath;
QMediaPlayer *player;
```

Соответственно подключим необходимые библиотеки (файл mainwindow.h):

```
#include <QMediaPlayer>
#include <QFileDialog>
```

Объекты следует создать (файл mainwindow.cpp, конструктор):

```
    player = new QMediaPlayer(this);
```

Самостоятельная работа

Давайте сразу сделаем так, чтобы функция открытия файла запоминала, где мы были в последний раз, и по умолчанию открывала нам именно этот каталог.

Сохранение пути следует прописать в конце процедуры открытия:

```
    qsDataPath = QFileInfo(fileName).path();
```

3. Теперь создадим (и сразу же програмно добавим его на форму) виджет, в котором мы и будем проигрывать видео. Сделаем это в конструкторе:

```
    videoWidget = new QVideoWidget(this);
    player = new QMediaPlayer(this);
    player->setVideoOutput(videoWidget);
    ui->verticalLayout->addWidget(videoWidget);
```

и опишем его в заголовочном файле (mainwindow.h, секция private):

```
    QVideoWidget *videoWidget;
```

а также подключим (в файле mainwindow.h):

```
#include <QWidget>
#include <QVideoWidget>
```

4. Теперь создадим событие на нажатие клавиши *Воспроизведение*:

```
    switch(player->state()) {
    case QMediaPlayer::PlayingState:
        player->pause(); break;
    default:
        player->play(); break;
    }
```

5. Допишите в открытие файла:

```
        player->play();
```

теперь видео будет сразу запускаться после открытия файла.

6. Скомпонуйте форму.

Все! Видеоплеер готов! Ура! Фанфары и тестирование. Попробуйте, например, нажать кнопку *Открыть* и посмотрите видео…

Подсказка: видео вы можете найти в «LabPract/Qt/Video».

Это важно: *скопируйте видео из сети на локальный носитель.*

7.1. Перемотка.

Ползунок для перемотки будем добавлять на отдельную панель инструментов: для этого нажмите ПКМ в инспекторе объектов на MainWindow (рис. 6.4).

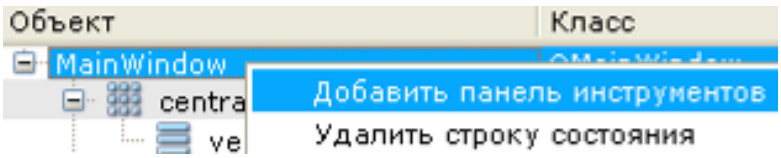


Рисунок 6.4 – Добавление еще одной панели инструментов

Для того, чтобы перескочить сразу на середину фильма, вам поможет следующий код (файл mainwindow.cpp, конструктор):

```
//Перемотка
    slider = new QSlider(Qt::Horizontal, this);
    ui->toolBar->addWidget(slider);
    connect(slider, SIGNAL(sliderMoved(int)), this, SLOT(setPosition(int)));
    connect(player, SIGNAL(durationChanged(qint64)), this,
SLOT(durationChanged(qint64)));
    slider->setRange(0, player->duration() / 1000);
```

Так как типы qint64 и int не взаимодействуют в режиме сигнал-слот – создадим собственные слоты.

```
void MainWindow::setPosition(int position)
{
    player->setPosition(position/1000);
}

void MainWindow::durationChanged(qint64 duration)
{
    this->duration = duration/1000;
    slider->setMaximum(duration / 1000);
}
```

Не забудьте их описать в заголовочном файле (mainwindow.h, секция слотов):

```
    void setPosition(int position);
    void durationChanged(qint64 duration);
```

Две прежде незнакомых компилятору переменных следует описать в классе (файл mainwindow.h):

```
    QSlider *slider;
    qint64 duration;
```

и подключить соответствующие модули (в файле mainwindow.h):

```
#include <QtWidgets>
```

7.2. Таймер воспроизведения.

Для таймера необходима текстовая переменная, куда мы будем выводить время (файл mainwindow.h, секция private):

```
    QLabel *labelDuration;
```

Создадим ее в конструкторе, добавим на панель и подключим слоты:

```
    labelDuration = new QLabel(this);
    ui->toolBar->addWidget(labelDuration);
    connect(slider, SIGNAL(sliderMoved(int)), this, SLOT(seek(int)));
```

```
    connect(player, SIGNAL(positionChanged(qint64)),
SLOT(positionChanged(qint64)));
```

Собственно реализация слотов:

```
void MainWindow::seek(int seconds)
{
    player->setPosition(seconds * 1000);
}

void MainWindow::positionChanged(qint64 progress)
{
    if (!slider->isSliderDown()) {
        slider->setValue(progress / 1000);
    }
    updateDurationInfo(progress / 1000);
}
void MainWindow::updateDurationInfo(qint64 currentInfo)
{
    QString tStr;
    if (currentInfo || duration) {
        QTime currentTime((currentInfo/3600)%60, (currentInfo/60)%60,
currentInfo%60, (currentInfo*1000)%1000);
        QTime totalTime((duration/3600)%60, (duration/60)%60, duration%60,
(duration*1000)%1000);
        QString format = "mm:ss";
        if (duration > 3600)
            format = "hh:mm:ss";
        tStr = currentTime.toString(format) + " / " + totalTime.toString(format);
    }
    labelDuration->setText(tStr);
}
```

В классе необходимо подключить эти два слота (в файле mainwindow.h, в секции слотов):

```
    void seek(int seconds);
    void positionChanged(qint64 progress);
```

и процедуру (в файле mainwindow.h, в секции `private`):

```
    void updateDurationInfo(qint64 currentInfo);
```

8. Громкость.

Для отключения звука добавьте еще одно действие action_Mute, установите галочку «триггерное». Перейдите к слоту (рис. 6.5), напишите:

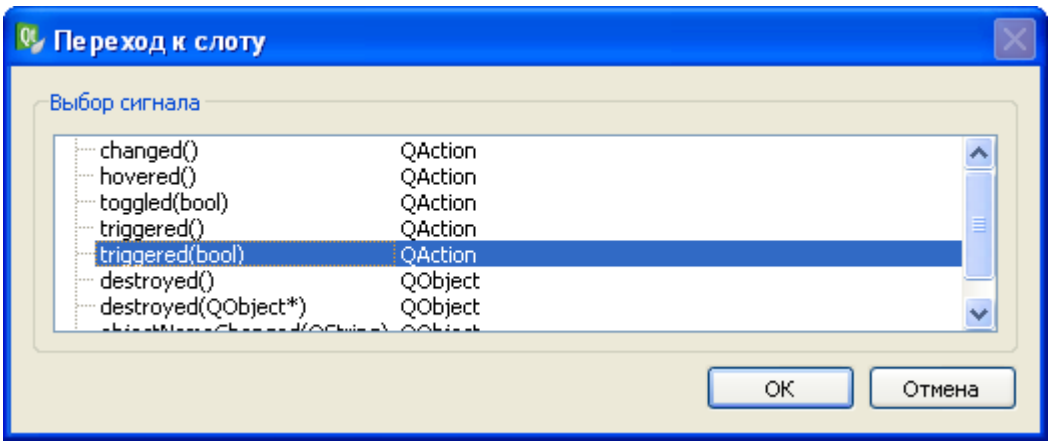


Рисунок 6.5 – Задание действия для выключения звука

```
    player->setMuted(checked);
```

Если мы хотим изменить громкость воспроизводимого, то необходимо дописать следующий код (в конструкторе):

```
    volumeSlider = new QSlider(Qt::Horizontal, this);
    volumeSlider->setRange(0, 100);
    ui->mainToolBar->addWidget(volumeSlider);
    connect(volumeSlider, SIGNAL(sliderMoved(int)), player, SLOT(setVolume(int)));
    volumeSlider->setValue(100);
```

В описании класса определить еще один указатель:

```
    QAbstractSlider *volumeSlider;
```

Протестируйте приложение еще раз.

9. Скорость воспроизведения.

Иногда удобнее перематывать изображение не с помощью ползунка, а посредством ускоренного воспроизведения.

Для начала в конструкторе установим изначальную скорость воспроизведения и выведем ее в строку статуса:

```
    player->setPlaybackRate(player->playbackRate() + 1);
    ui->statusBar->showMessage("Скорость воспроизведения: " +
QString::number(player->playbackRate()), 3000);
```

Теперь добавим два пункта меню Rate_Fast, Rate_Slow и создадим для них соответствующие действия.

Для ускорения видео:

```
    player->setPlaybackRate(player->playbackRate() + 1);
    ui->statusBar->showMessage("Скорость воспроизведения: " +
QString::number(player->playbackRate()), 3000);
```

Для его замедления видео:

```
    player->setPlaybackRate(player->playbackRate() - 0.5);
    ui->statusBar->showMessage("Скорость воспроизведения: " +
QString::number(player->playbackRate()), 3000);
```

10. Полноэкранный просмотр.

Пришла пора создать самую полезную кнопку – *На весь экран*. Итак, добавим данное действие в меню и на панель инструментов, установим ему свойство *триггерное* и создадим обработчик этого действия:

```
   videoWidget->setFullScreen(!videoWidget->isFullScreen());
```

Теперь у вас видео будет раскрываться на полный экран, но вот обратно – никак. Все дело в том, что `videoWidget`, раскрывшись на весь экран, стал «круче», чем `MainWindow`, и теперь нам надо работать именно с ним.

Для этого нам нужно описать в заголовочном файле свой собственный класс (типа `QVideoWidget`):

```
class VideoWidget : public QVideoWidget
{
    Q_OBJECT
public:
    explicit VideoWidget(QWidget * parent = 0);
```

```
protected:
    void keyPressEvent(QKeyEvent *event);
    void mouseDoubleClickEvent(QMouseEvent *event);
};
```

Реализация этого класса:

```
VideoWidget::VideoWidget(QWidget *parent)
{
    setSizePolicy(QSizePolicy::Ignored, QSizePolicy::Ignored);
    QPalette p = palette();
    p.setColor(QPalette::Window, Qt::black);
    setPalette(p);
    setAttribute(Qt::WA_OpaquePaintEvent);
}

void VideoWidget::keyPressEvent(QKeyEvent *event)
{
    setFullScreen(false);
}
void VideoWidget::mouseDoubleClickEvent(QMouseEvent *event)
{
    setFullScreen(!isFullScreen());
    event->accept();
}
```

Осталось лишь заменить

```
videoWidget = new QVideoWidget(this);
```

на

```
videoWidget = new VideoWidget(this);
```

Протестируете приложение, оцените функционал полноэкранного просмотра (рис. 6.6).

Рисунок 6.6 – Полноэкранный просмотр

Имеющаяся на данный момент реализация метода, отлавливающего нажатие, клавиш далека от идеала, расширим ее функционал:

```
    switch (event->key())
    {
    case Qt::Key_Escape: //нажат Escape;
        setFullScreen(false);
        break;
    case Qt::Key_Up: //нажата Up;
        break;
        //...
    default:
        QVideoWidget::keyPressEvent(event);
        break;
    }
```

11. Создание плейлиста.

У каждого уважающего себя медиаплеера должен быть плейлист! Его мы реализуем просто – бросив на форму Table Widget. Допишем в конструкторе:

```
    playlist = new QMediaPlaylist();
    player->setPlaylist(playlist);
```

Ежели мы будем работать со списком файлов, то кардинально поменяется процедура открытия файла (теперь их будет несколько).

```
  QStringList fileNames = QFileDialog::getOpenFileNames(this, tr("Open Files"),
"C:\\TEMP");
    foreach (QString const &argument, fileNames)
    {
        QFileInfo fileInfo(argument);
        if (fileInfo.exists()) {
            QUrl url = QUrl::fromLocalFile(fileInfo.absoluteFilePath());
            if (fileInfo.suffix().toLower() == QLatin1String("m3u")) {playlist-
>load(url);}
            else playlist->addMedia(url);}
        else {
             QUrl url(argument);
             if (url.isValid()) {playlist->addMedia(url);}
             }
    }
         player->setMedia(playlist->media(0));
         player->play();
         ui->tableWidget->show();
         if (ui->tableWidget->columnCount() == 0){ui->tableWidget-
>insertColumn(0);}
         int size = fileNames.size();
         int currentRow = ui->tableWidget->rowCount();
         for (int i = 0; i < size; ++i)
         {
             ui->tableWidget->insertRow(currentRow);
             QTableWidgetItem *item = new QTableWidgetItem();
             item->setText(QFileInfo(fileNames[i]).fileName());
             ui->tableWidget->setItem(currentRow, 0, item);
             ++currentRow;
         }
         ui->tableWidget->setColumnWidth(0, this->width());
         if (ui->tableWidget->currentRow() < 0){ui->tableWidget-
>setCurrentItem(ui->tableWidget->item(0, 0));}
}
```

12. Вы, должно быть, заметили, что при развернутом на полный экран видео нет возможности работать с «горячими клавишами» действий. Так, например, вы не можете поставить видео на паузу при полноэкранном просмотре – давайте исправим сей недостаток.

В описании класса *VideoWidget* добавим сигнал:

```
signals:
    void signal_keyPress(QKeyEvent *event);
```

Вызовем этот сигнал в обработчике *keyPressEvent* класса *VideoWidget*:

```
    emit signal_keyPress(event);
```

Для класса MainWindow создадим обработчик *keyPressEvent*, а также в классе объявим соответствующий слот:

```
    void keyPressEvent(QKeyEvent *event);
```

В конструкторе класса MainWindow свяжем сигнал и слот:

```
        connect(videoWidget, SIGNAL(signal_keyPress(QKeyEvent*)), this,
SLOT(keyPressEvent(QKeyEvent*)));
```

Протестируйте работоспособность приложения, проверьте функционирование всех действий при полноэкранном просмотре.

Самостоятельная работа (п. 13-14)

13. Теперь сделаем так, чтобы работали кнопочки *Назад, Вперед*. Тут следует рассмотреть два варианта: когда какой-то файл уже проигрывается, либо, когда еще ничего не запущено (рис. 6.7).

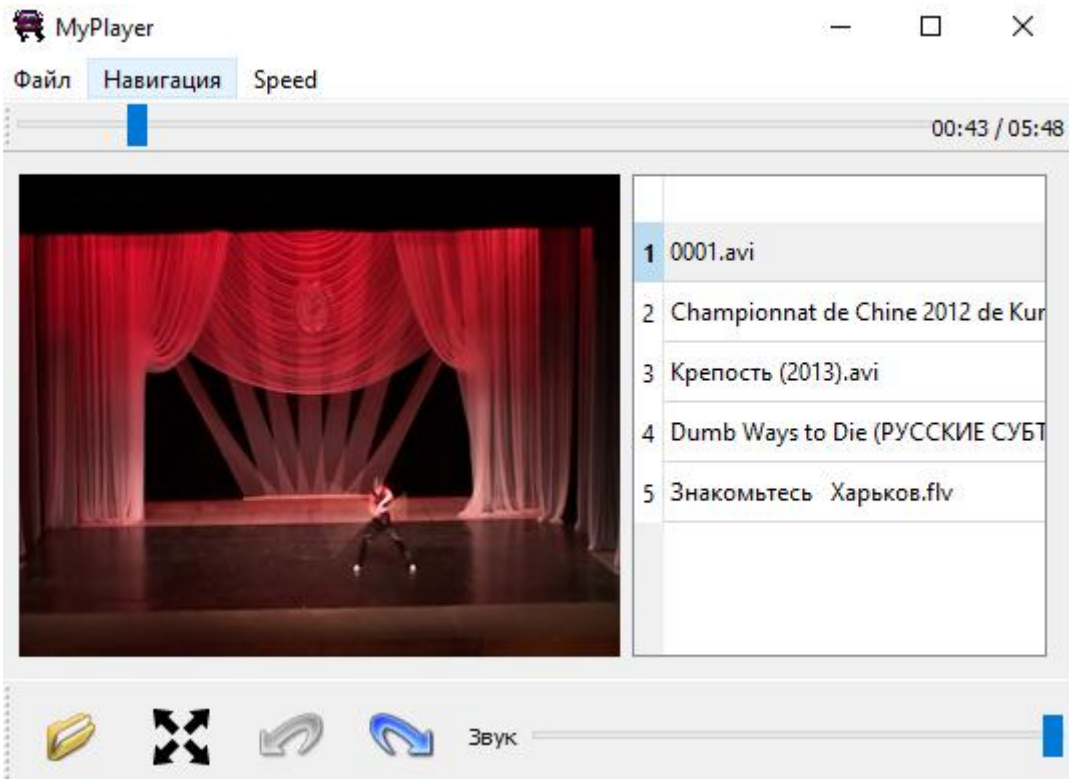


Рисунок 6.7 – Навигация по плейлисту

14. Неплохо было бы ориентироваться по плейлисту также и с помощью мышки. Для этого выберем Table Widget на форме, нажмем правую кнопку мыши и выберем *Перейти к слоту* (рис. 6.8), а также не забудем подключить:

```
#include <QTableWidgetItem>
```

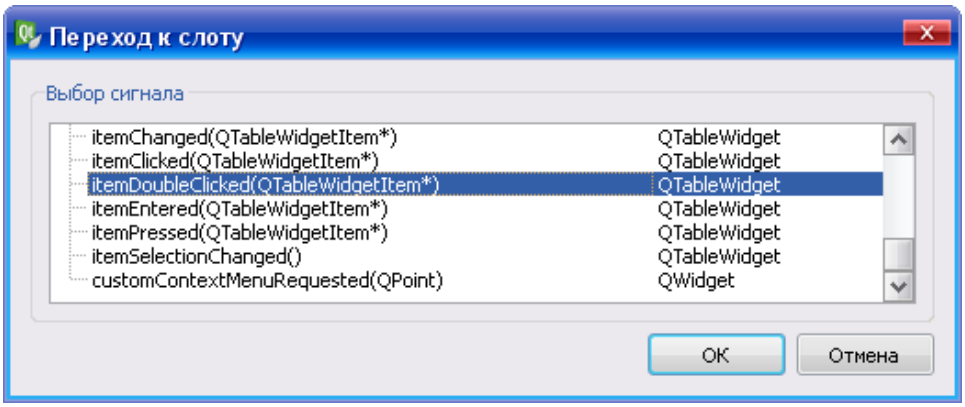


Рисунок 6.8 – Создание реакции на двойной клик мышью

Оценим результат нашей работы (рис. 6.9), протестировав приложение.

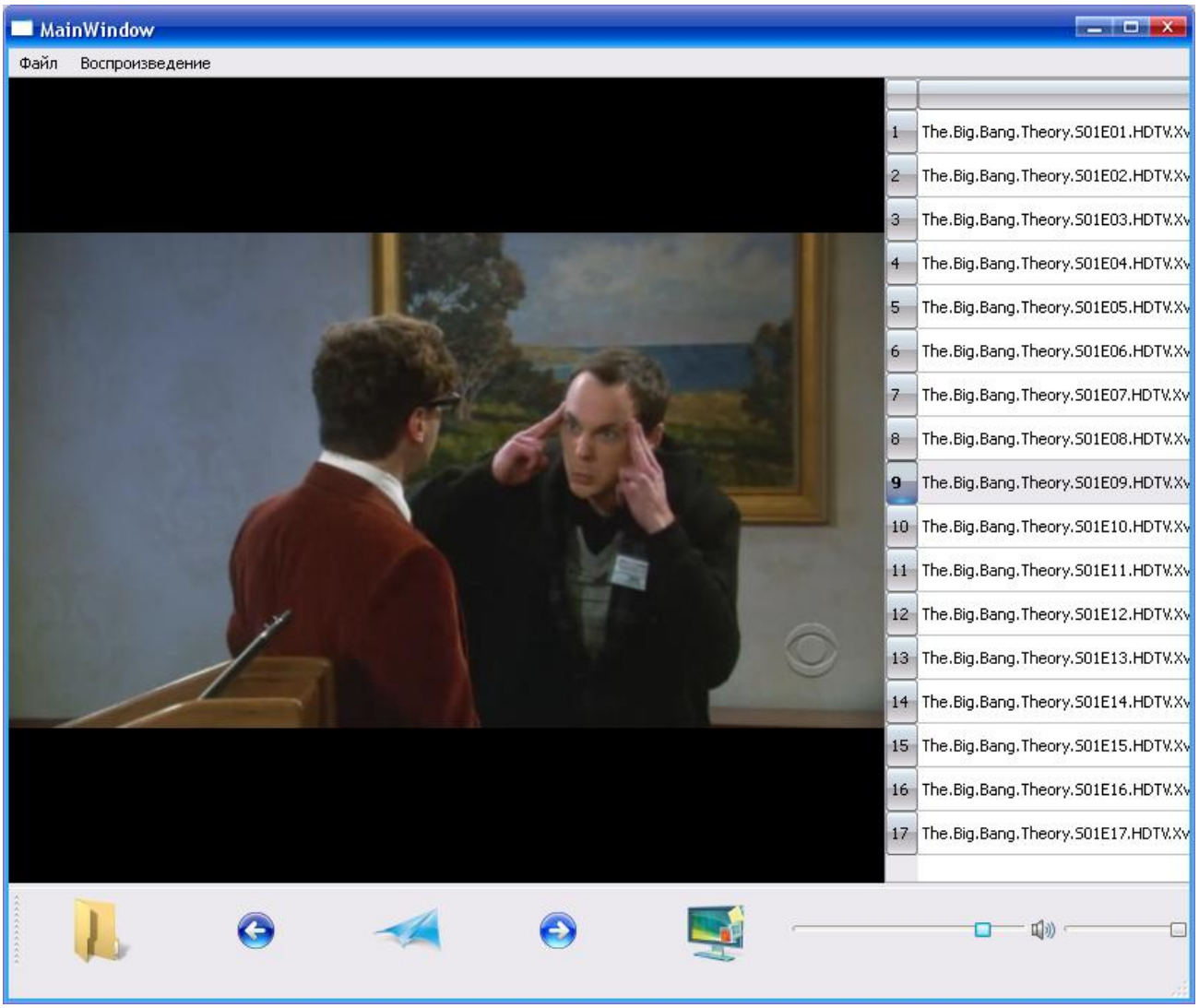


Рисунок 6.9 – Почти полноценный медиаплеер

7 РАБОТА С 2D-ГРАФИКОЙ В QT

Цель работы *– изучить основы работы с 2D-графикой с помощью Qt посредством написания простейшего просмотрщика картинок.*

Хотелось ли вам иметь быстрый и красивый, а главное, удобный просмотрщик графики? А, как известно: хочешь, чтобы программа была хорошей – напиши ее своими руками.

Всем надоел стандартный просмотрщик Windows! Вместо него мы сделаем прекрасный и ультрасовременный Viewer – QViewer!

Дальнейшая последовательность действий почти аналогична той, что мы проделывали при создании видеоплеера. Посему приведу сокращенный вариант.

1. Запустите *Qt Creator*. Снова воспользуемся мастером.

Выберите *Файл - Новый файл или проект…*, а там *Приложение Qt Widgets*.

Дадим ему имя *QViwer*, укажем месторасположение проекта и нажмем кнопку *Далее* (три раза) и *Завершить*.

2. Займемся созданием и редактированием действий.

2.1. Создайте стандартное меню *Файл* с пунктами *Открыть, Выход*.

Между *Открыть* и *Выход* добавьте разделитель.

Дадим им имена action_Open и action_Quit соответственно, а также добавим к ним горячие комбинации клавиш: Ctrl+O, Ctrl+Q. Для action_Quit см. рис. 6.1 (да-да, это ссылка на лаб. работу № 6, но вы можете не листать странички, если помните, что делать).

Теперь приступим к action_Open, зададим иконку (предварительно создав и подключив файл ресурсов), бросим ее на панель инструментов, нажмем правой копкой мыши на action_Open в редакторе действий и выберем: Перейти к слоту…, далее (см. рис. 6.2), нажимаем ОК.

```
    QString last_open_dir="";
    QString File_Name = QFileDialog::getOpenFileName(this, tr("Open Image"),
last_open_dir, tr("Image Files (*.png *.jpg *.bmp)"));
    if (!File_Name.isNull())
    {
        QImage image(File_Name);
        if (image.isNull())
        {
        QMessageBox::information(this, tr("Image Viewer"),tr("Cannot load
%1.").arg(File_Name));
        return;
        }
    imageLabel->setPixmap(QPixmap::fromImage(image));
    imageLabel->adjustSize();
```

Во-первых, обратите внимание на `last_open_dir` в первой строчке, там сразу пропишите пути к каталогу, где хранятся рисунки, чтобы каждый раз при тестировании не искать их по каталогам.

Напомним также о необходимости использования двойного слэша при написании адреса, например: "D:\\Photo_2\\Fall\\".

Если же в названии будут присутствовать русские символы, обратитесь к лаб. работе №5.

Во-вторых, следует подключить (в заголовочном файле):

```
#include <QImage>
#include <QtGui>
#include <QFileDialog>
```

а также описать две переменные:

```
    QLabel *imageLabel;
    QScrollArea *scrollArea;
```

2.2. Теперь поработаем с конструктором:

```
    ui->setupUi(this);
    imageLabel = new QLabel;
    imageLabel->setBackgroundRole(QPalette::Base);
    imageLabel->setSizePolicy(QSizePolicy::Ignored, QSizePolicy::Ignored);
    imageLabel->setScaledContents(true);
    scrollArea = new QScrollArea;
    scrollArea->setBackgroundRole(QPalette::Dark);
    scrollArea->setWidget(imageLabel);
    setCentralWidget(scrollArea);
   //resize(1600, 1066); //если захотите менять размеры
```

и давайте сразу загрузим туда картинку, так сказать по умолчанию: для этого выберем подходящий рисунок, переместим его в папку с исполняемым файлом и загрузим.

Подсказка: `QString File_Name =qApp->applicationDirPath()+"/Red_Sky.jpg";`

Еще одна подсказка: картинки вы можете найти в «LabPract/Qt/Pictures».

2.3. Приложение должно быть привлекательным: измените заголовок окна на «Самый лучший просмотрщик картинок, созданный САПРовцем» (либо на любой другой по вашему усмотрению), и добавьте иконку к окошку (предварительно подключив ее в файле ресурсов).

2.4. Создайте меню Просмотр, а в нем пункты: *Истинный размер*, *По размеру окна*, *На весь экран*, *Предыдущий*, *Следующий*. Придайте им соответствующие иконки, переименуйте действия в action_Real, action_Stretch, action_Full, action_Back и action_Forward соответственно, а также задайте им горячие клавиши: Ctrl+A, Ctrl+B, Ctrl+F, ←,→.

Разместите их на панели инструментов.

2.5. Работа с панелью инструментов:

– переместите панель инструментов вниз – так будет удобнее;

– выберите в **Инспекторе объектов** *Панель инструментов* либо просто щелкните по ней на форме, в **Окне свойств** найдите iconSize и установите значения 96 и 48 для ширины и высоты, соответственно (либо другие значения – в зависимости от выбранного набора иконок).

3. **Истинный размер** – реализуйте самостоятельно.

4. **По размеру окна** – нажатие на эту кнопку растянет картинку до размеров окна нашего приложения и привяжет к границам окна, при этом возможны искажения картинки. Повторное нажатие отвязывает картинку.

```
    if (scrollArea->widgetResizable())
    {
        scrollArea->setWidgetResizable(false);
        menuBar()->show();
        ui->mainToolBar->show();
    }
    else
    {
        scrollArea->setWidgetResizable(true);
        menuBar()->hide();
       // ui->mainToolBar->hide();
    }
```

Также мы прячем одно из имеющихся в наличии меню. Если же спрятать оба (и меню, и панель инструментов), то, по неведомой нам причине, горячие клавиши перестанут работать, и мы не сможем вернуться в предыдущее состояние.

5. **На весь экран**:

```
        if (this->isFullScreen())
        {
           if ( _maximized ) this->showMaximized();
           else this->showNormal();
           menuBar()->show();
        }
        else
        {
            _maximized = this->isMaximized();
            this->showFullScreen();
            menuBar()->hide();
        }
```

Следует описать еще одну переменную, но вы догадываетесь, где и как…

Это интересно: *действия «По размеру окна» и «На весь экран» очень занятно комбинируются – проверьте.*

6. **Туда и обратно.**

Попробуем научиться перемещаться по каталогам с картинками – для этого напишем код обработчиков соответствующих событий:

```
void MainWindow::on_action_Back_triggered()
{
    if (current_index_ - 1 >= 0)
    {
        --current_index_;
       imageLabel->setPixmap(list_[current_index_].filePath());
    }
}
void MainWindow::on_action_Forward_triggered()
{
    if (current_index_ + 1 < list_.count())
    {
```

```
        ++current_index_;
       imageLabel->setPixmap(list_[current_index_].filePath());
    }
}
```

Как видно, следует добавить парочку переменных (в заголовочный файл):

```
    int current_index_;
    QFileInfoList list_;
```

Возникает закономерный вопрос: «Где определить эти переменные?»

Для того, чтобы изваять список с картинками, нужна следующая процедура:

```
QFileInfoList MainWindow::getListFiles(QString dirPath) const
{
    QDir dir(dirPath);
    QStringList filters;
    filters << "*.jpg" << "*.jpeg" << "*.png" << ".bmp" << "tiff" << "ico";
    dir.setNameFilters(filters);
    return dir.entryInfoList(filters, QDir::Files);
}
```

Не забудем ее описать в классе:

```
    QFileInfoList getListFiles(QString dirPath) const;
```

А вызывать ее надо после диалога открытия файла:

```
    last_open_dir = File_Name.left(File_Name.lastIndexOf('/'));
     list_ = getListFiles(last_open_dir);
     current_index_ = list_.indexOf(QFileInfo(File_Name));
```

Напомню, что изначально кнопки «вперед-назад» должны быть неактивными.

7. **Работа с принтером.**

```
void MainWindow::print()
{
    Q_ASSERT(imageLabel->pixmap());
#if !defined(QT_NO_PRINTER) && !defined(QT_NO_PRINTDIALOG)
    QPrintDialog dialog(&printer, this);
    if (dialog.exec()) {
        QPainter painter(&printer);
        QRect rect = painter.viewport();
        QSize size = imageLabel->pixmap()->size();
        size.scale(rect.size(), Qt::KeepAspectRatio);
        painter.setViewport(rect.x(), rect.y(), size.width(), size.height());
        painter.setWindow(imageLabel->pixmap()->rect());
        painter.drawPixmap(0, 0, *imageLabel->pixmap());
    }
#endif
}
```

Подключите необходимые библиотеки.

8. **Больше-меньше.**

Добавьте лупы () на экран для увеличения и уменьшения размеров картинки. Реализуйте реакцию на нажатие этих кнопок.

В этом вам поможет:

```
void MainWindow::scaleImage(double factor)
```

```
{
    Q_ASSERT(imageLabel->pixmap());
    scaleFactor *= factor;
    imageLabel->resize(scaleFactor * imageLabel->pixmap()->size());

    adjustScrollBar(scrollArea->horizontalScrollBar(), factor);
    adjustScrollBar(scrollArea->verticalScrollBar(), factor);

    zoomInAct->setEnabled(scaleFactor < 3.0);
    zoomOutAct->setEnabled(scaleFactor > 0.333);
}
```

Опишите в классе новые переменные.

Самостоятельная работа

9. **Зеркальное отображение.**

Расширьте функционал вашего приложения, добавив возможность зеркально отобразить картинку.

10. **Инверсия цвета.** Добавьте возможность инвертировать цвета RGB.

Для работы с цветовой палитрой в Qt предусмотрен ряд констант (табл. 7.1).

Таблица 7.1 – Константы цветов в Qt

Цвет	Константа	Значение	16-ричный код	Описание
	Qt::white	3	#ffffff	Белый
	Qt::black	2	#000000	Черный
	Qt::red	7	#ff0000	Красный
	Qt::darkRed	13	#800000	Темно-красный
	Qt::green	8	#00ff00	Зеленый
	Qt::darkGreen	14	#008000	Темно-зеленый
	Qt::blue	9	#0000ff	Синий
	Qt::darkBlue	15	#000080	Темно-синий
	Qt::cyan	10	#00ffff	Сине-зеленый
	Qt::darkCyan	16	#008080	Темно-сине-зеленый
	Qt::magenta	11	#ff00ff	Пурпурный
	Qt::darkMagenta	17	#800080	Темно-пурпурный
	Qt::yellow	12	#ffff00	Желтый
	Qt::darkYellow	18	#808000	Темно-желтый
	Qt::gray	5	#a0a0a4	Серый
	Qt::darkGray	4	#808080	Темно-серый
	Qt::lightGray	6	#c0c0c0	Светло-серый
	Qt::transparent	19	QColor(0, 0, 0, 0)	Прозрачный черный цвет
	Qt::color0	0		0 pixel value (for bitmaps)
	Qt::color1	1		1 pixel value (for bitmaps)

8 РАБОТА С БАЗАМИ ДАННЫХ В QT

Цель работы *– научиться создавать базы данных и работать с ними в Qt.*

Теоретическое введение

База данных представляет собой систему хранения записей, организованных в виде таблиц. База данных может содержать от одной до нескольких сотен таблиц, которые связаны между собой. Таблица состоит из набора строк и столбцов. Столбцы таблицы имеют имена, и за каждым столбцом закреплен тип и/или область значений. Строки таблицы баз данных называются записями, а ячейки, на которые делится запись, – полями. *Первичный ключ* – это уникальный идентификатор записи, который может представлять собой не только один столбец, но и целую комбинацию столбцов.

Пользователь может выполнять множество разных операций с таблицами: добавлять, изменять и удалять записи, вести поиск и т. д. Для составления подобного рода запросов был разработан язык SQL (Structured Query Language – язык структурированных запросов), который дает возможность не только осуществлять запросы и изменять данные, но и создавать новые базы данных.

В библиотеке Qt имеются драйверы для работы со следующими СУБД:

- QDB2 – IBM DB2 версии не ниже 7.1;
- QIBASE – Borland InterBase;
- QMYSQL – MySQL;
- QOCI – Oracle;
- QODBC – ODBC (в том числе Microsoft SQL Server);
- QPSQL – PostgreSQL;
- QSQLITE – SQLite версии не ниже 3;
- QSQLITE2 – SQLite версии 2;
- QTDS – Sybase Adaptive Server.

Также есть возможность написать свой собственный драйвер для работы с БД, если имеющиеся вас не устраивают.

В Qt Open Source Edition отсутствует поддержка коммерческих СУБД Oracle, Sybase и DB2, т.к. драйверы для них распространяются под лицензией, не совместимой с GPL.

SQLite – компактная встраиваемая система управления реляционной базой данных.

Исходный код библиотеки передан в общественное достояние.

SQLite не обладает клиент-серверной архитектурой. То есть движок БД не является отдельно работающим процессом, с которым взаимодействует программа. SQLite представляет собой библиотеку, с которой компонуется ваша программа, и, таким образом, движок становится составной частью программы.

То есть это СУБД, не требующая сервера БД и собственно клиента.

8.1 Подготовка к подключению БД

1. Запустите *Qt Creator* (…\Qt\Tools\QtCreator\bin\qtcreator.exe).

Воспользуйтесь мастером. Выберите *Файл → Новый файл или проект…*, а там *Приложение → Проект Qt Widget* (рис. 8.1).

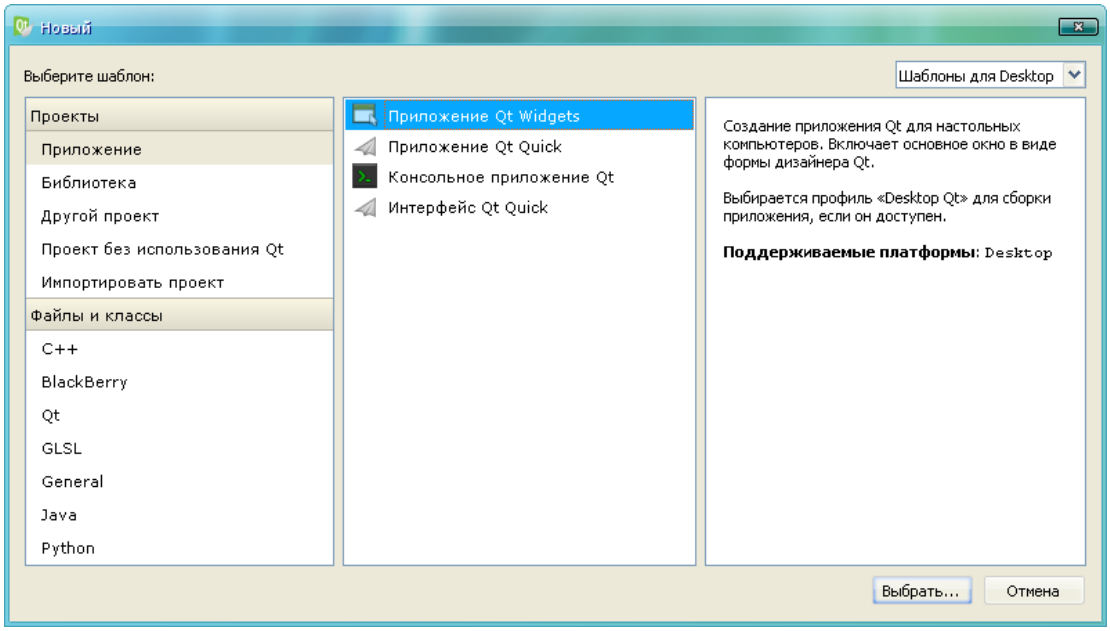


Рисунок 8.1 – Создание проекта с помощью мастера

2. Дадим ему некое осмысленное имя, укажем месторасположение проекта (рис. 8.2), нажмем *Далее* (три раза) и *Завершить*.

> ***Это важно:*** *прописывая путь к директории проекта, избегайте символов кириллицы, иначе проект не будет компилироваться.*

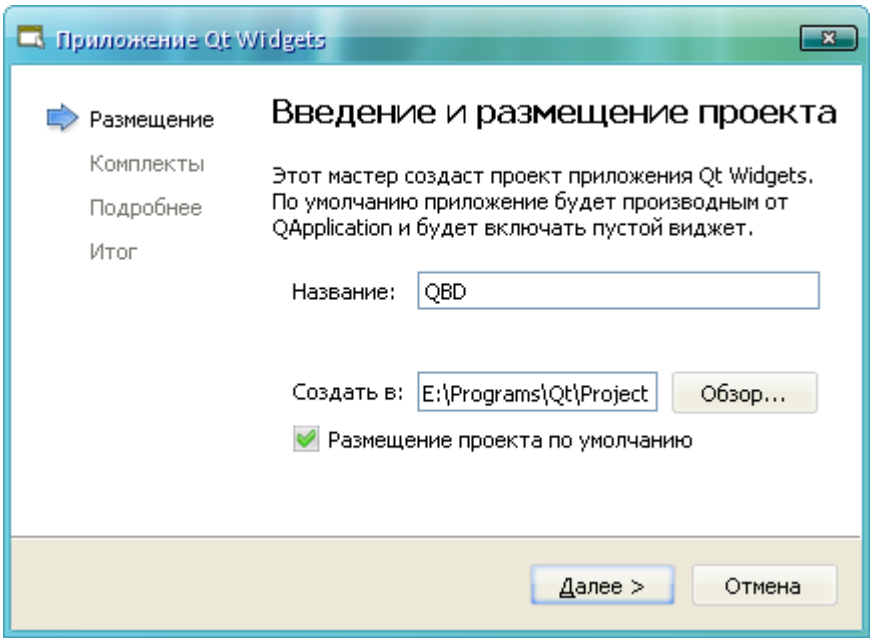


Рисунок 8.2 – Путь и имя проекта

3. По проекту удобно перемещаться с помощью панели навигации (рис. 8.3).

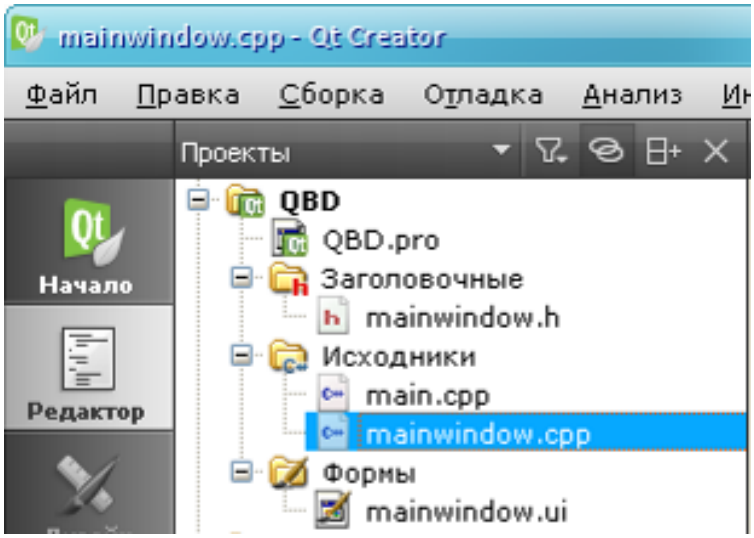


Рисунок 8.3 – Панель навигации

4. Подключим SQL.

Для использования баз данных Qt предоставляет отдельный модуль Qt Sql. Чтобы подключить его, необходимо дописать в файле проекта (с расширением .pro):

```
QT           += sql widgets
```

а также, для того чтобы работать с классами этого модуля, необходимо включить их в заголовочный файл главного окна (mainwindow.h):

```
#include <QtSql>
#include <QSqlDatabase>
#include <QSqlError>
#include <QSqlQuery>

#include <QMessageBox> //для вывода трассировочных сообщений
```

Запустите и протестируйте приложение.

8.2 Подключение к базе данных и выполнение SQL-запросов

1. Установление соединения с БД.

Добавим в класс главного окна переменные и функцию (файл mainwindow.h, секция private):

```
// база данных
QSqlDatabase db;
// модель данных
QSqlTableModel *model;
// функция для подключения к базе данных
bool createConnection();
```

Создадим функцию в файле mainwindow.cpp для реализации подключения к БД.

```
bool MainWindow::createConnection()
{
    return true;
}
```

Подключим ее в конструкторе (после `ui->setupUi(this);`):

```
createConnection();
```

Весь дальнейший код пишите внутри функции createConnection().

Для подключения к базе данных надо указать название SQL-драйвера, например:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
```

Затем можно указать имя сервера, название базы данных, имя пользователя и пароль:

```
//    db.setHostName("localhost"); // или, например, "my1.server.ru"
//    db.setDatabaseName("mydb1");
//    db.setUserName("root");
//    db.setPassword("mypassword");
```

Все эти параметры можем опустить.

Вместо этого напишем:

```
db.setDatabaseName(":memory:");
```

Это важно. *Предопределённое имя ":memory:" позволяет размещать временную базу данных в оперативной памяти.*

После того, как все параметры подключения заданы, можно *открыть соединение* (метод open()). Если подключение установить не удалось, то неплохо бы узнать описание ошибки и сообщить его пользователю:

```
if (!db.open()) {
    QMessageBox::critical(0, qApp->tr("Cannot open database"),
        qApp->tr("Unable to establish a database connection.\n"
                 "This example needs SQLite support. Please read "
                 "the Qt SQL driver documentation for information how "
                 "to build it.\n\n"
                 "Click Cancel to exit."), QMessageBox::Cancel);
    return false;
}
```

2. Первичное заполнение БД.

Для исполнения команд SQL после установления соединения можно использовать класс QSqlQuery. Запросы (команды) оформляются в виде обычной строки, которая передается в метод QSqiQuery::exec().

Это важно: *класс QSqlQuery может быть использован для исполнения* ***DML (Data Manipulation Language)*** *выражений, таких, как SELECT, INSERT, UPDATE и DELETE, и* ***DDL (Data Definition Language)*** *выражений, таких, как CREATE TABLE.*

Если подключение установлено, то можно выполнить любой SQL-запрос, например, создать таблицу:

```
if (db.tables().empty()) //проверка на наличие таблиц
{
    //QMessageBox::critical(0, tr("Error"), tr("Database is empty"));
    //создание таблицы:
    QSqlQuery query;
    query.exec("CREATE TABLE person (id integer PRIMARY KEY NOT NULL, "
```

```
                    "firstname varchar(20), lastname varchar(20))");
            //заполнение таблицы:
            query.exec("INSERT INTO person VALUES(1, 'Alina', 'Bichenko')");
            query.exec("INSERT INTO person VALUES(2, 'Kate', 'Mikova')");
            query.exec("INSERT INTO person VALUES(3, 'Olga', 'Skalskaya')");
            query.exec("INSERT INTO person VALUES(4, 'Alexander', 'Panchenko')");
            query.exec("INSERT INTO person VALUES(5, 'Victorya', 'Silivina')");
            query.exec("INSERT INTO person VALUES(6, 'July', 'Tokmakova')");
        }
```

Данные можно добавлять и по-иному:

```
        query.prepare("INSERT INTO person (id, firstname, lastname, ball) "
                      "VALUES (?, ?, ?, ?)");
        query.addBindValue(7);
        query.addBindValue("Bart");
        query.addBindValue("Simpson");
        query.addBindValue(1);
        query.exec();
```

Можно просто использовать подставляемые аргументы, которые предоставляет QString:

```
        query.exec(QString("INSERT INTO person (id, firstname, lastname, ball)
VALUES (%1, '%2', '%3', %4);").arg("8").arg("Бильбо").arg("Бэггинс").arg("5"));
```

4. Вывод содержимого таблицы на экран.

Перейдите в режим визуального редактирования (дважды щелкнув по mainwindow.ui).

Перетащите элемент Table View на форму (рис. 8.4).

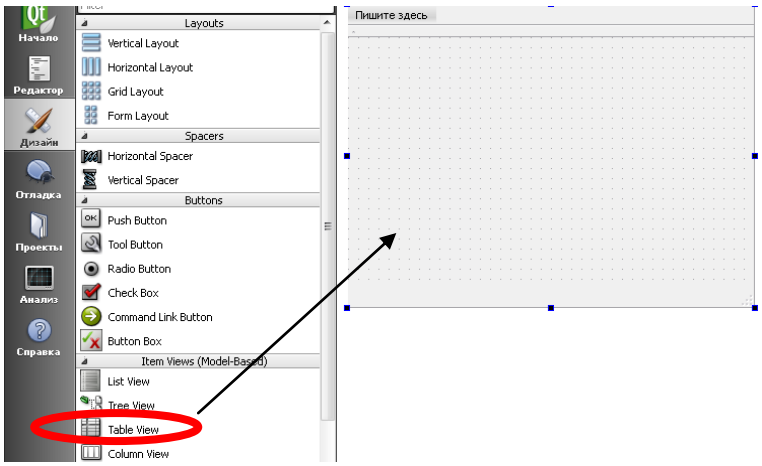
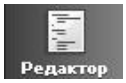


Рисунок 8.4 – Редактирование главного окна

Вернитесь в mainwindow.cpp (нажав на 

либо 

).

Свяжем модель данных с таблицей на форме:

```
        // объявляем модель данных
        model = new QSqlTableModel(this);
        // указываем таблицу из БД
        model->setTable("person");
        // заносим данные в модель
        // если удачно
        if(model->select())
        {
```

```
    // передаем данные из модели в tableView
    ui->tableView->setModel(model);
    // устанавливаем высоту строки по тексту
    ui->tableView->resizeRowsToContents();
    // шапка для первой колонки
    model->setHeaderData(0, Qt::Horizontal, tr("ID"));
    // шапка для второй колонки
    model->setHeaderData(1, Qt::Horizontal, tr("Имя"));
    // передача управления элементу tableView
    ui->tableView->setFocus();
}
```

Запустите и протестируйте приложение.

5. Компоновка и отображение.

Используя (в режиме дизайна) компоновку  , добейтесь фиксированного расположения элементов на экране.

Перейдите к свойствам элемента tableView и найдите horizontalHeaderStretchLastSection и поставьте “птичку” – это позволит автоматически растянуть колонки по всей длине tableView.

Измените отображения названий всех столбцов, например:

```
model->setHeaderData(2, Qt::Horizontal, QObject::tr("Фамилия"));
```

Если перейти к свойствам элемента tableView и активировать SortingEnable – это позволит сортировать столбец по клику на его заголовке.

Запустите и протестируйте приложение.

6. Дополнительная информация.

Класс *QSqlTableModel* предоставляет следующие стратегии редактирования (устанавливаемые с помощью *setEditStrategy*()):

– **OnRowChange** –записывает данные, как только пользователь перейдет к другой строке таблицы.

– **OnFieldChange** – осуществляет запись после того, как пользователь перейдет к другой ячейке таблицы.

– **OnManualSubmit** – записывает данные по вызову слота *submitAll*(). Если вызывается слот *revertAll*(), то данные возвращаются в исходное состояние.

В программном коде это выглядит вот так:

```
model->setEditStrategy(QSqlTableModel::OnFieldChange);
```

Если при выполнении запроса возникла ошибка, то метод lastError() позволяет вывести на экран её описание:

```
if (!query.isActive()) QMessageBox::warning(this, tr("Database Error"),
query.lastError().text());
```

7. **Самостоятельная работа.**

7.1. Сделайте так, чтобы в БД заносились данные о студентах, которые выполняют эту работу, т. е. о вас.

7.2. Измените отображаемые названия для столбцов таблицы на корректные, например: Имя, Фамилия и т. д.

7.3. Сделайте столбец с первичным ключом невидимым. Для этого следует вбить ui, нажать на «точку» – появится список доступных элементов. В нем выберем tableView, снова нажмем на точку и постараемся вспомнить, как по-анлийски будет: «Скрыть колонку».

7.3. Добавьте в таблицу столбец «Средний балл». Отсортируйте таблицу по этому показателю.

7.4. Измените размеры окна, добавьте кнопку «Выход», реализуйте эту возможность.

7.5. Добавьте столбец «Дата рождения».

7.6. Создайте оболочку для работы с БД (рис. 8.5):

– должна появиться возможность перехода к следующей (Next), предыдущей (Previous), первой (First) и последней (Last) записям таблицы (см. п. 8);

– название таблицы должно быть отображено в заголовке окна;

– все поля должны быть отображены на форме;

– должна появиться возможность прямо с формы добавить нового студента.

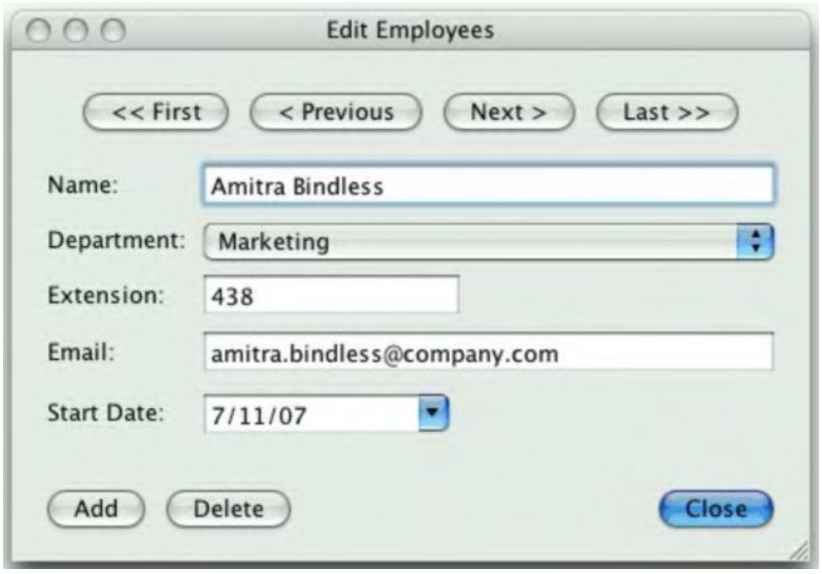


Рисунок 8.5 – Пример для работы с другой таблицой

8. Навигация по записям в таблице.

8.1. Mapper.

Для того, чтобы перемещаться по записям в таблице, необходимо воспользоваться классом QDataWidgetMapper, создав его описание в классе (файл mainwindow.h, секция private):

```
    QDataWidgetMapper *mapper;
```

там же его необходимо подключить:

```
#include <QDataWidgetMapper>
```

и упомянуть его в конструкторе (mainwindow.cpp, функция `createConnection()`), связав с моделью:

```
mapper = new QDataWidgetMapper(this);
mapper->setModel(model);
```

Перейдите в режим визуального редактирования (дважды щелкнув по mainwindow.ui).

Разбейте компоновку  и уменьшите таблицу.

Перетащите элементы Line Edit и Label на форму. С помощью клавиши Ctrl выделите оба виджета и скомпонуйте их по горизонтали  . С помощью окна свойств (рис. 8.6) измените имя Line Edit с lineEdit на nameEdit.

Измените свойство text у Label на «Имя».

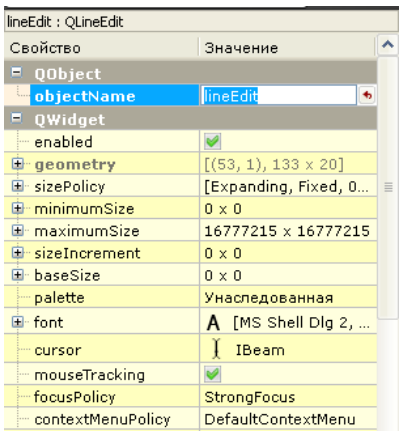


Рисунок 8.6 – Инспектор свойств

Вернитесь в mainwindow.cpp, допишите:

```
mapper->addMapping(ui->nameEdit, model->fieldIndex("firstname"));
mapper->toFirst();
```

Обратите внимание – в качестве примера связь установлена лишь с одним полем «Имя», связь с другими полями установите самостоятельно.

Запустите и протестируйте приложение.

8.2. Кнопки навигации.

Перетащите элемент PushButton на форму, задайте ему имя next_Button, измените отображаемый текст на клавише «Вперед», либо «Следующий», либо «>>». Нажмите на кнопку правой кнопкой мыши, выберите «Перейти к слоту…» (рис. 8.7).

В открывшемся диалоговом окне выберите сигнал clicked() и нажмите *OK*.

В появившейся функции напишите:

```
mapper->toNext();
```

Для остальных кнопок – аналогично: toPrevious(), toFirst(), toLast().

Это важно: *кнопки следует именовать корректно, т.е. НЕ pushButton1, pushButton2… а previous_Button, first_Button…*

Для кнопки удаления:

```
model->removeRow(mapper->currentIndex());
model->select();
```

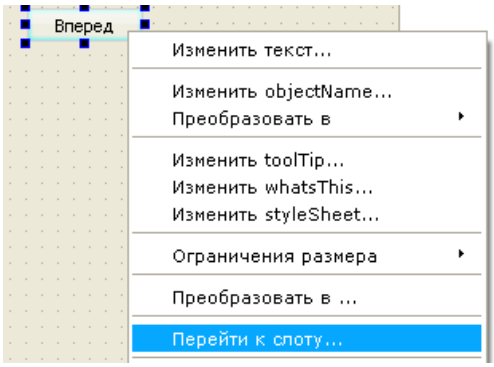


Рисунок 8.7 – Создание реакции на нажатие клавиши

Как создавать другие SQL-запросы?

Для заполнения также можно использовать следующий код:

```
QSqlQuery query;
query.prepare("INSERT INTO person (id, firstname, lastname) "
              "VALUES (:id, :firstname, :lastname);");
// query.bindValue(":id", "7"); //id будет обновляться самостоятельно
query.bindValue(":firstname", "Garry");
query.bindValue(":lastname", "Potter");
query.exec();
model->select(); mapper->toLast();
```

Этот SQL-запрос можно заменить операциями над моделью:

```
model->insertRows(0,1);
model->setData(model->index(0, 1), "Tyrion");
model->setData(model->index(0, 2), "Lannister");
model->setData(model->index(0, 3), 5);
model->submitAll();
model->select(); // если закомментировать - добавит в верх таблицы
mapper->toLast();
```

Предположим, вы хотите сделать такой запрос:

```
SELECT * FROM person WHERE ball >= 1 ORDER BY id DESC
```

Это можно сделать так:

```
query.exec("SELECT * FROM person WHERE ball >= 1 ORDER BY ball DESC");
```

Или же достаточно задать фильтр и условие сортировки:

```
    model.setTable("person");           // Имя таблицы базы данных.
    model.setFilter("ball >= 1");   // Условие WHERE.
    model.setSort(0, Qt::DescendingOrder); // Сортировка по убыванию id.
    model.select();                        // Получить данные.
```

9. Сделайте так, чтобы таблица сохранялась на компьютере, а не хранилась в оперативной памяти.

P.S. Также рекомендуется просмотреть примеры: sqlbrowser, sqlwidgetmapper…

8.3 Реляционные БД в Qt

Реляционная база данных – база данных, основанная на реляционной модели данных. Слово «реляционный» происходит от англ. relation («отношение», «зависимость», «связь»).

1. Подключение класса для реляционных связей.

Для обеспечения реляционных связей в Qt служит класс QSqlRelationalTableModel.

Соответственно, заменим используемый нами ранее класс QSqlTableModel на QSqlRelationalTableModel:

– в заголовочном файле:

```
    // модель данных
    QSqlRelationalTableModel *model;
```

– в исполняемом файле:

```
            // объявляем модель данных
            model = new QSqlRelationalTableModel(this);
```

2. Добавление поля для связи.

Самостоятельная работа: напишите SQL-запрос, который добавляет новое поле «group_id» (номер группы) в таблицу «person».

3. Создание новой таблицы и заполнение ее данными.

Самостоятельная работа: напишите SQL-запрос, который создает новую таблицу «group_list», состоящей из двух полей («group_id», «group_name»), вставьте в нее номер вашей группы.

Попробуйте вывести на экран содержимое этой таблицы (это для проверки корректности функционирования вашего SQL-запроса, после того как вы сделаете это – верните все в исходное состояние). Для этого следует заменить

```
    model->setTable("person");
```

на

```
    model->setTable("group_list");
```

> ***Это интересно:*** *можете попробовать создать таблицу с именем «group» – посмотрите, что у Вас получится.*

4. Добавление связей.

```
            model->setRelation(5, QSqlRelation("group_list", "group_id",
"group_name"));
      //5 - колонка внешнего ключа, "group_list" - таблица с которой связываются,
"group_id" - колонка первичного ключа, "group_name" - подставляемое поле
```

Обратите особое внимание на номер колонки внешнего ключа (нумерация колонок начинается с нуля).

Протестируйте приложение.

5. Добавление нового окна для отображения таблицы.

Предположим, что у нас есть необходимость просмотреть и отредактировать таблицу «group_list», для этого необходимо создать новое окно, разместить на нем элемент Table View и связать его с новой моделью.

Реализовывать все это мы будем через меню и панель инструментов.

5.1. Создание меню.

Щелкните дважды на форме mainwindow.ui – откроется визуальный редактор (рис. 8.8). Найдите слова *Пишите здесь* и последуйте этому указанию.

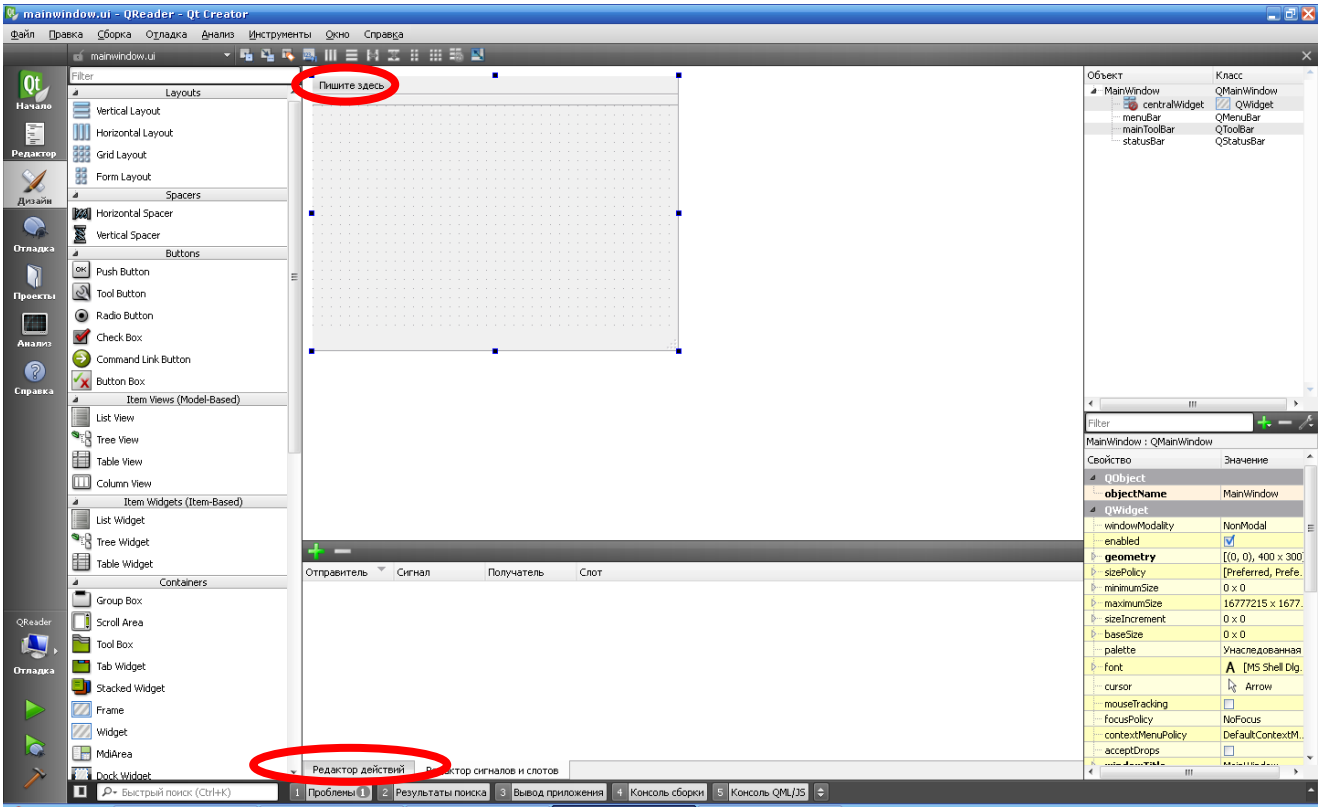


Рисунок 8.8 – Редактирование главного окна

Создайте стандартное меню *Авторизация* с пунктами *Авторизация*, *Выход*.

Между пунктами меню *Авторизация* и *Выход* добавьте разделитель.

Создайте еще одно меню *Администрирование* с именами ваших таблиц (например, *Таблица Группы*).

5.2. Редактирование действий.

Если вы внимательно посмотрите вниз экрана (рис. 8.8, выделено красным внизу), то обнаружите, что там появились так называемые *Действия* (*Action*) – реакции на нажатие соответствующих пунктов меню.

Давайте модифицируем их, заменив названия на более осмысленные, например, action_Login, action_Quit и action_tbl_St, action_tbl_Gr, а также добавим к ним горячие комбинации клавиш. Делается это просто: в окне редактирования действия, чтоб попасть в него – дважды щелкните на самом действии (рис. 8.9), нажмите: Ctrl+L, Ctrl+Q для соответствующих действий по авторизации и выходу.

Это важно: *вводите наименования действий внимательно, например, не стоит случайно ставить в нем пробел в конце (это не так просто заметить).*

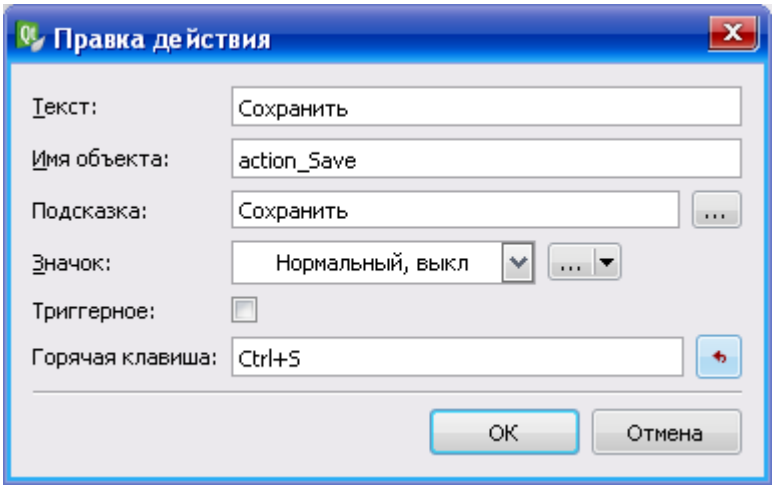


Рисунок 8.9 – Окно редактирования действия

5.3. Для того чтобы сделать меню более привлекательным, добавим к каждому действию свою иконку, а чтоб иконкам было где жить, создадим файл ресурсов (рис. 8.10, 8.11).

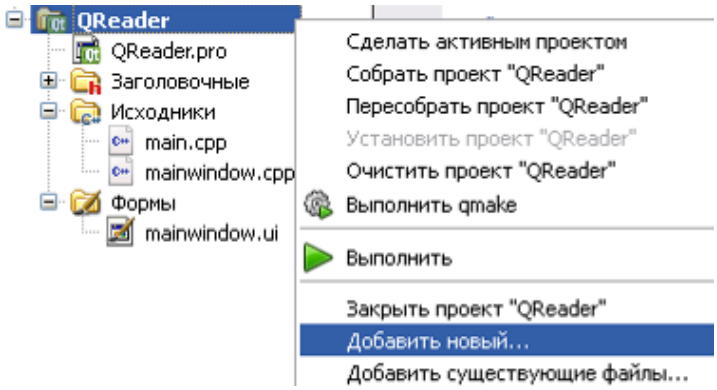


Рисунок 8.10 – Добавление нового файла в проект

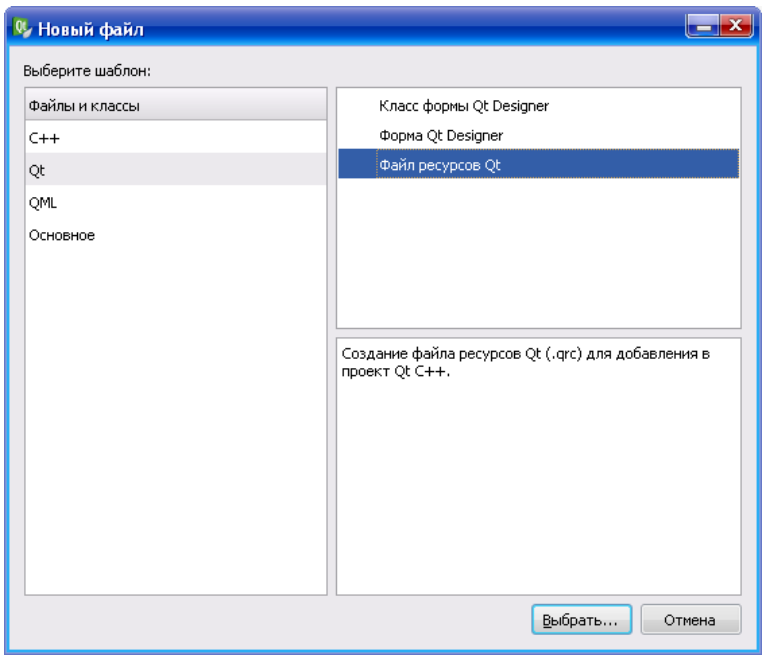


Рисунок 8.11 – Создание файла ресурсов

Нас попросят ввести его имя (рекомендую использовать имя проекта, но это остается на вашем усмотрении), а также размещение ресурсного файла (в принципе его можно разместить, где угодно, но опять же лучше расположить его в директории проекта, что услужливо рекомендует программа). Нажмите кнопку *Далее* и затем *Завершить*.

Теперь дважды щелкните на QBD.qrc (если вы так назвали файл) на боковой панели. Затем нажмите на кнопку *Добавить* и выберите *Добавить префикс* (рис. 8.12). Можете поменять название префикса на более осмысленное.

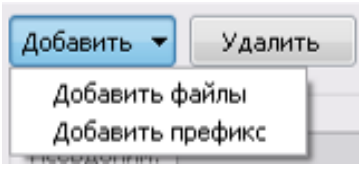


Рисунок 8.12 – Добавление префиксов и файлов в ресурсный файл

Сейчас нам следует найти (или самим создать) файлы, которые мы будем использовать в качестве иконок. Размеры иконок должны быть 32x32. Рекомендуется взять файлы из «<Путь установки Qt>\<№ версии Qt>\ \mingw492_32\examples\widgets\mainwindows\application\images». Добавьте их в ресурсный файл.

Лучше переместить файлы изображений в каталог с программой и поместить их, например, в созданную вами папку «images».

Скомпилируйте проект.

Снова вернемся в окно редактирования формы и зададим иконки для всех действий. Сделать это можно через Окно редактирования действия (рис. 8.9), либо через окно редактирования свойств (рис. 8.13). Таким образом, задайте иконки для всех пунктов меню (рис. 8.14).

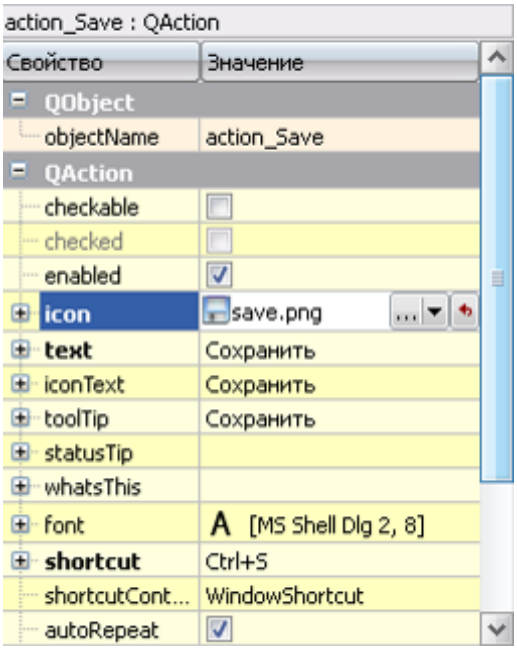


Рисунок 8.13 – Окно редактирования свойств действия

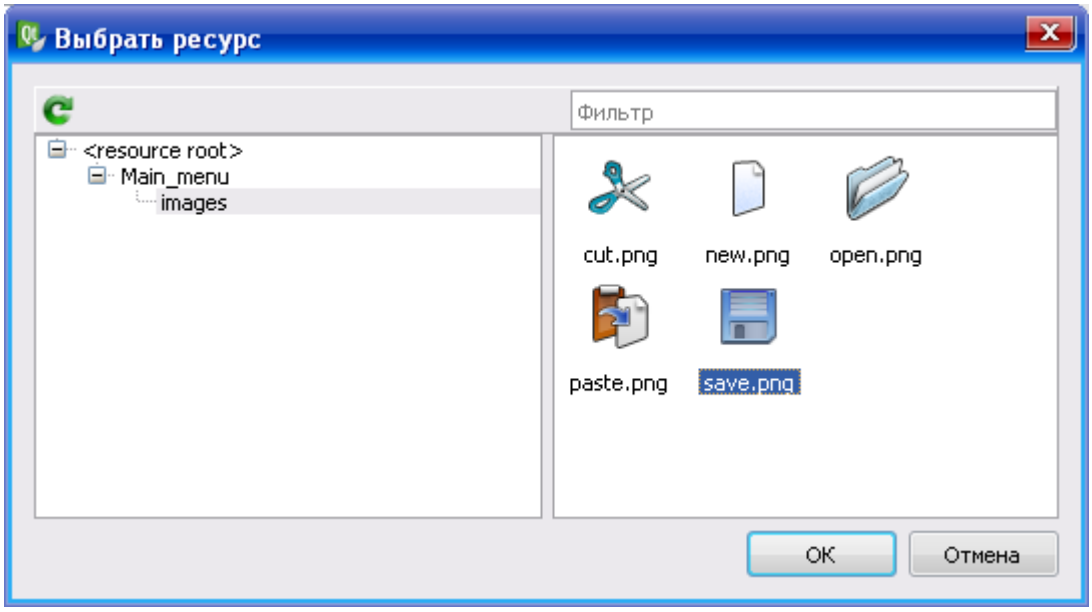


Рисунок 8.14 – Выбор иконок для пунктов меню

5.4. Рядом с **редактором действий** есть **редактор сигналов и слотов** (рис. 8.15), перейдем в него и последовательно выберем: action_Quit, triggered(), MainWindow, close(). Теперь по нажатию на пункт меню *Выход* главное окно приложения закроется.

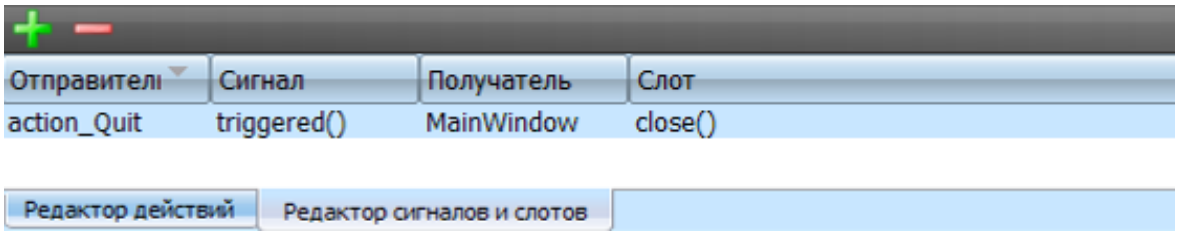


Рисунок 8.15 – Редактирование действия action_Quit

5.5. Приложение должно быть красивым: измените заголовок окна на «Самое лучшее клиентское приложение к базе данных» (либо на любой другой по вашему усмотрению) и добавьте иконку к окошку (предварительно подключив ее в файле ресурсов).

См. свойство WindowTitle.

5.6. Панель инструментов.

Когда вы попробуете добавить что-либо на панель инструментов, у вас, скорее всего, возникнут некоторые сложности, так как контекстное меню не позволяет что-либо добавить, кроме как объект под названием Разделитель.

Все достаточно просто: выделите желаемое действие (в редакторе действий) и перетащите его на панель инструментов – оно появится на панели инструментов (рис. 8.16). Ничего более прописывать не надо: к этим кнопкам автоматически подключаются уже созданные вами процедуры. Осталось лишь добавить разделители, расположив их так же, как и в меню.

Выберите в **Инспекторе объектов** *Панель инструментов* либо просто щелкните по ней на форме. В **Окне свойств** найдите iconSize и установите значения 96 и 48 для ширины и высоты соответственно (рис. 8.17).

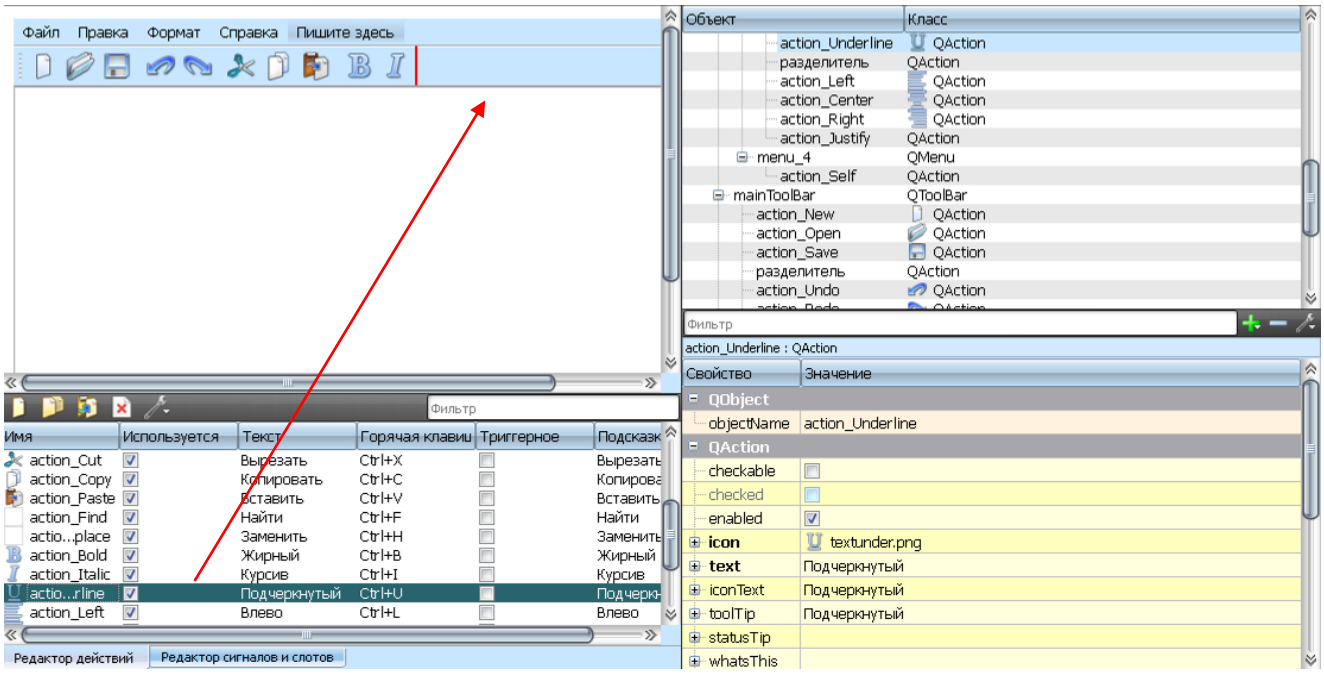


Рисунок 8.16 – Добавление действий на панель инструментов

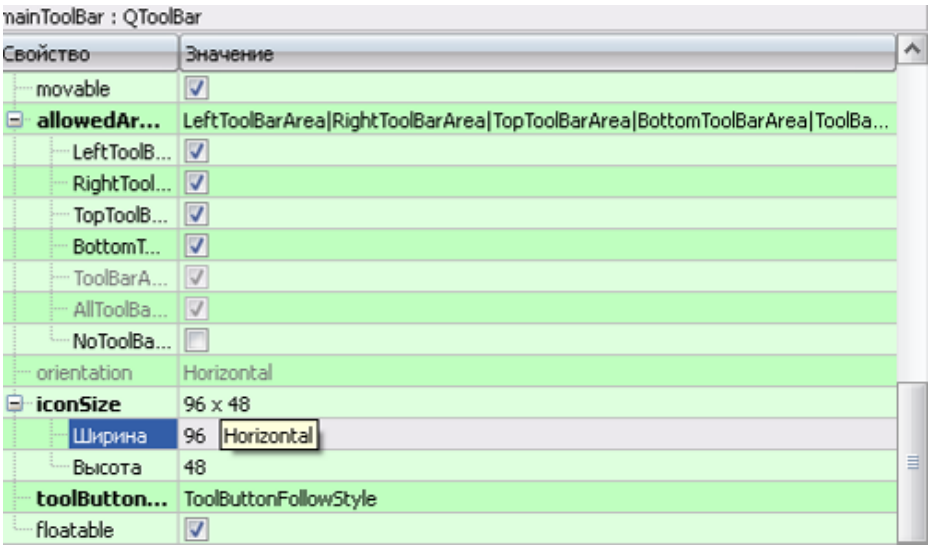


Рисунок 8.17 – Задание размеров иконок

5.7. Подключение нового окна приложения для отображения таблицы с номерами групп.

Для того, чтобы подключить новую форму к программе, следует: щелкнуть правой кнопкой мыши на корневой папке в дереве проектов (см. рис. 8.9), затем выбрать *Добавить новый…*, в появившемся окне выбрать, соответственно, *Qt* и *Класс формы Qt Designer* (рис. 8.18).

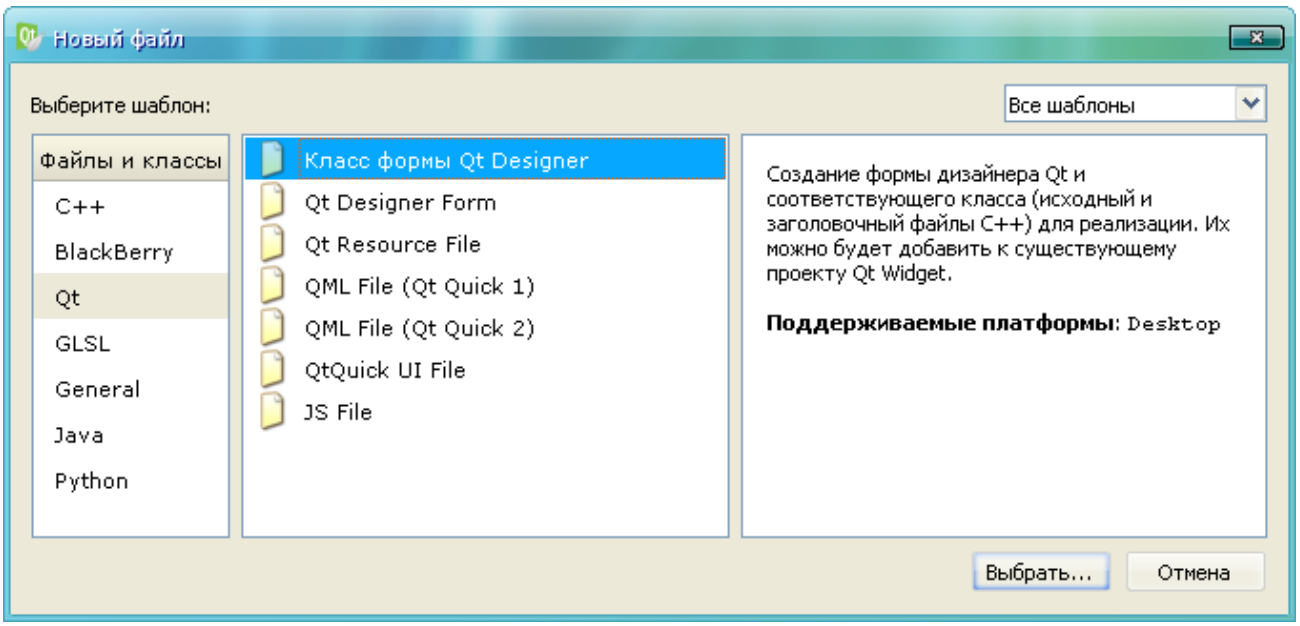


Рисунок 8.18 – Подключение новой формы

Затем задайте имя класса: «*Tbl_Gr_Dialog*», соответственно, файл формы будет носить имя «*tbl_gr_dialog.ui*». Такие же имена, но с другим расширением будут иметь заголовочный файл и файл с исходными кодами.

Нажмем правой копкой мыши на *action_tbl_Gr* в редакторе действий и выберем: *Перейти к слоту…*, далее (рис. 8.19), нажмем *ОК*.

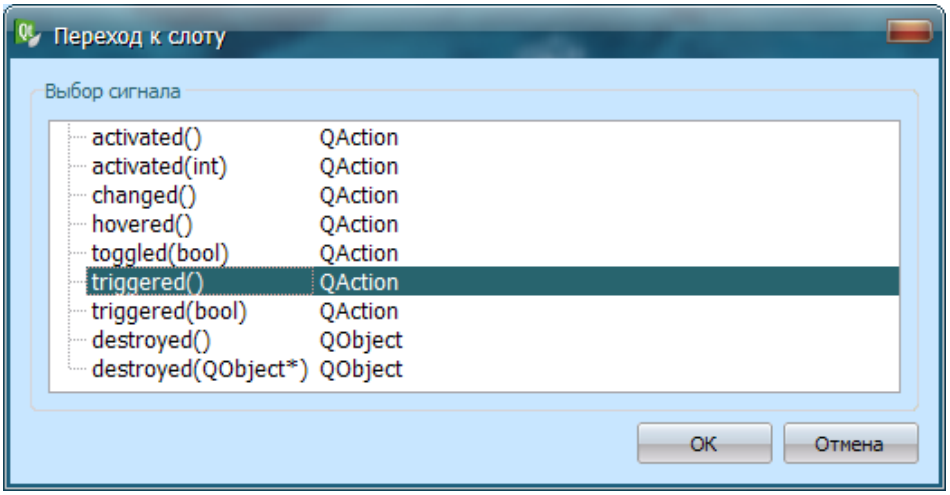


Рисунок 8.19 – Переход к слоту действия

Само подключение реализуется так (в файле mainwindow.cpp):

```
#include "tbl_gr_dialog.h"
```

В *void MainWindow::on_action_tbl_gr_triggered()*:

```
Tbl_Gr_Dialog form_group;
    form_group.exec();
```

там же необходимо дописать следующий код, чтобы данные в основной таблице обновились:

```
    model->select();
```

Есть еще одно магическое выражение, которое стоит добавить туда же:

```
model->relationModel(5)->select();      //5 - колонка внешнего ключа,
```

Оно обновляет данные, выводимые в выпадающем списке.

5.8. Наполнение формы.

Добавьте на форму элемент Table View и свяжите его с таблицей «*group_list*», скомпонуйте форму, дайте ей заголовок, установите иконку для окна.

Это интересно: *если вы хотите, чтобы изменения не сразу сохранялись в БД, а накапливались, пока не будет нажата кнопка «Сохранить изменения», то можно создать два новых действия (action_Save, action_Undo) и прописать данную реализацию в конструкторе:*

```
connect(ui->action_Save,SIGNAL(triggered()),model,SLOT(submitAll()));
connect(ui->action_Undo,SIGNAL(triggered()),model,SLOT(revertAll()));
```

В таком случае вам необходимо будет убрать `submitAll()` *по тексту программы, дабы сей код мог нормально функционировать. Эта заметка не является руководством к действию, а лишь информацией к сведению – лучше оставить программный код в таком виде, какой он есть.*

6. Чтобы на нашей главной таблице мы могли выбирать номера групп из имеющегося списка групп, следует добавить всего одну строчку в конструктор:

```
ui->tableView->setItemDelegate(new QSqlRelationalDelegate(ui->tableView));
```

Дабы было из чего выбирать, дополните таблицу «group_list».

7. Для реализации операции соединения реляционной алгебры используются константы InnerJoin и LeftJoin:

```
model->setJoinMode(QSqlRelationalTableModel::LeftJoin);
```

По умолчанию установлен режим InnerJoin, что не позволяет отобразить записи с нулевыми внешними ключами (т. е. вы не увидите данные о студентах, у которых нет номера группы). LeftJoin – отобразит все поля.

8. **Самостоятельная работа:**

– предоставьте возможность добавить и удалить группы;

– реализуйте авторизацию (ввод логина и пароля) посредством отдельного окна, запускаемого перед стартом главного окна приложения (которое запуститься только при введении верного пароля); заметим, что QLineEdit также поддерживает режимы ввода (echoMode): NoEcho и Password.

8.4 Делегаты. Сортировка, контроль ввода и поиск по БД

1. Сортировка.

Помимо выполнения классических SQL-запросов на сортировку данных, последовательность отображаемых данных можно изменить посредством виджета.

Делается это вполне элементарно – перейдите в режим дизайна и в инспекторе свойств поставьте галочку напротив значения sortingEnabled для экземпляра класса QTableView (рис. 8.20).

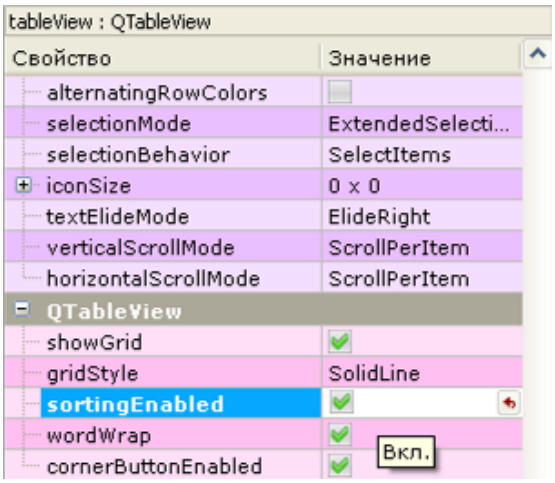


Рисунок 8.20 – Установка возможности сортировки по имени столбца

Теперь, щелкнув по заголовку окна, вы сможете отсортировать данные.

2. Контроль диапазона ввода.

В режиме дизайна добавьте Double Spin Box (из раздела Input Widgets) для поля, отвечающего за средний балл, если вы этого еще не сделали раньше.

В инспекторе свойств поменяйте, например, максимальное значение и шаг – измененные параметры при этом подкрасятся жирным шрифтом (рис. 8.21).

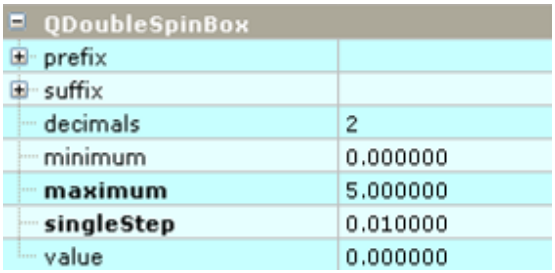


Рисунок 8.21 – Контроль диапазона ввода для Double Spin Box

Для экземпляров класса QLineEdit длина текста может быть ограничена с помощью maxLength(). Для текста можно задать условия, используя validator() или inputMask(), либо их оба.

Сначала валидатор необходимо описать в классе (в файле mainwindow.h):

```
    QRegExpValidator *exp;
```

Затем в конструкторе установить для определенного поля (mainwindow.cpp):

```
    QRegExp regExp("[A-Za-zA-Яа-яЁё'-]{20}");
        exp= new QRegExpValidator(regExp,this);
        ui->nameEdit->setValidator(exp);
```

В приведенном выше примере маска устанавливается для поля с именем и позволяет вводить лишь приведенные выше символы, включая букву «ё», апостроф и дефис для сложных имен. Совокупность символов *{20}* – показывает

ограничение на количество вводимых символов, его можно задать также через инспектор свойств (поле maxLenght).

При изменении текста испускается сигнал textChanged(); при изменении текста с помощью setText() – textEdited(); при перемещении курсора – cursorPositionChanged(); а при нажатии клавиш Return или Enter – returnPressed().

После окончания редактирования (редактор теряет фокус или нажата клавиша Return/Enter) испускается сигнал editingFinished().

Обратите внимание на то, что если установлено условие для текста (validator()), то сигналы returnPressed()/editingFinished() испускаются только в случае, если условие для текста возвращает QValidator::Acceptable.

Следует помнить, что для **приведения типов** используются методы:

– .toString() в строку;
– .toDouble() в вещественное;
– .toInt() в целое.

3. Модальность.

В графическом интерфейсе пользователя **модальным** называется окно, которое блокирует работу пользователя с родительским приложением до тех пор, пока пользователь это окно не закроет.

Это интересно: *прямо сейчас окна являются модальными, однако, если установить дополнительному окну свойство модальности, то оно перестанет им быть (рис. 8.22).*

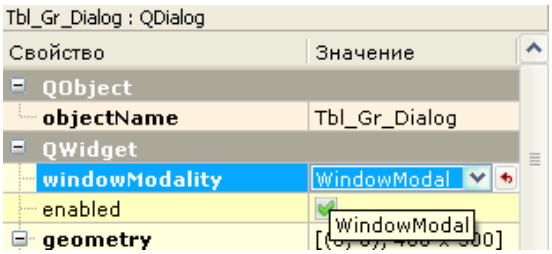


Рисунок 8.22 – Установка модальности дополнительного окна

4. Подсказки.

Различают три типа помощи:

– воздушная подсказка;
– подсказка "Что это";
– система помощи (Online Help).

4.1. Воздушная подсказка.

Работая с программами, вы, наверное, заметили, что при задержке указателя мыши над кнопками панелей инструментов автоматически появляется небольшое текстовое окошко, поясняющее назначение кнопки. Такое окно называется воздушной подсказкой (*tooltip*) и, как правило, содержит только одну строку

текста. Можно использовать подсказки, имеющие больше одной строки, но такие подсказки нужно делать как можно короче.

Присутствие воздушной подсказки в приложении является необязательным, но лучше все-таки предоставлять ее, поскольку она помогает пользователям быстрее сориентироваться среди множества кнопок. Несомненный плюс этого типа подсказки в том, что, не останавливая своей работы с приложением, пользователь получает информацию об элементах приложения.

Чаще всего такие подсказки применяются для кнопок панелей инструментов, но их можно с успехом использовать для любых виджетов. При этом не забывайте о том, что воздушные подсказки не следует применять в тех случаях, когда объяснение излишне. Например, будет нелогично, если для кнопки Cancel (Отмена) повторить ее надпись в воздушной подсказке. Для того, чтобы установить подсказку в виджете, нужно вызвать метод setToolTip().

```
    ui->nameEdit->setToolTip("Имя");
```

4.2. Подсказка "Что это".

Иногда требуется отобразить больше информации о виджете, чем способна предоставить воздушная подсказка. Подсказка What's this (Что это) является промежуточной между воздушной подсказкой и системой помощи (рис. 8.23). По своей функциональности эта подсказка очень похожа на воздушную, но с той разницей, что она не появляется автоматически при задержке указателя мыши. Для ее отображения нужно войти в специальный режим, нажав кнопку ?, находящуюся на панели инструментов или в области заголовка окна. Также можно воспользоваться стандартной комбинацией клавиш Shift + Fl.

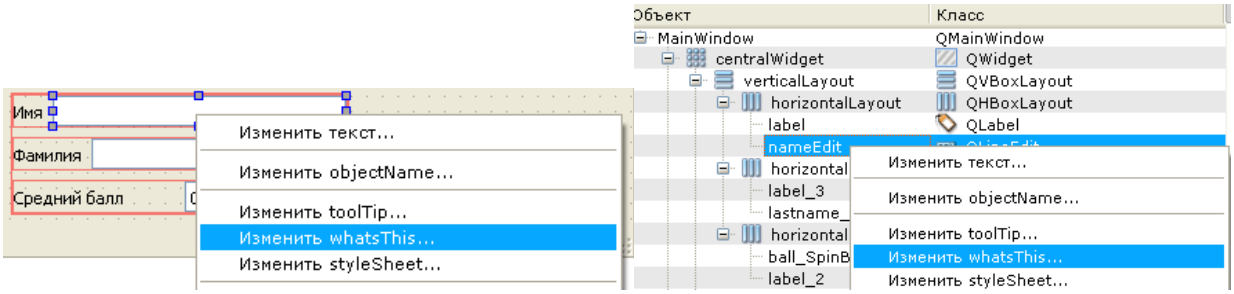


Рисунок 8.23 – Установка подсказки «Что это»

Как видно, есть небольшая проблема – кнопка «?» отсутствует в области заголовка окна у главной формы, чтобы его установить вместо кнопок «Свернуть / Развернуть» нужно поработать с флагами (в файле main.cpp):

```
    w.setWindowFlags(Qt::WindowCloseButtonHint|Qt::WindowContextHelpButtonHint);
```

5. Автодополнения.

```
        QStringList wordList;
        wordList<<"Тони"<<"Стив"<<"Наташа"<<"Халк"<<"Ник"<<"Фил"<<"Хэппи";
        QCompleter *completer = new QCompleter(wordList, this);
        completer->setCompletionMode(QCompleter::PopupCompletion);
        //completer->setCompletionMode(QCompleter::UnfilteredPopupCompletion);
        //completer->setCaseSensitivity(Qt::CaseSensitive);
//регистрочувствительность
        completer->setModelSorting(QCompleter::UnsortedModel);
        ui->nameEdit->setCompleter(completer);
```

Чтобы данные в поле подгружались из базы данных:

```
QCompleter *completer = new QCompleter(model, this);
completer->setCompletionColumn(1); //здесь 1 - № столбца с именами
```

6. Делегаты.

У Qt есть свои средства для отображения и редактирования элементов данных в модели – QItemDelegate. Использование QItemDelegate позволяет отображать и предоставлять механизмы редактирования, которые будут настроены и разработаны независимо от модели и представления.

Архитектура модель-представление-контроллер

Архитектура модель-представление-контроллер (Model-View-Controller, MVC) является шаблоном проектирования.

MVC состоит из трех типов объектов. **Модель** – объект приложения, **представление** – его экранное представление и **контроллер** – определяет реакцию пользовательского интерфейса на пользовательский ввод. До MVC при разработке пользовательского интерфейса эти объекты совмещались вместе. MVC разделяет их – для увеличения гибкости и возможности повторного использования.

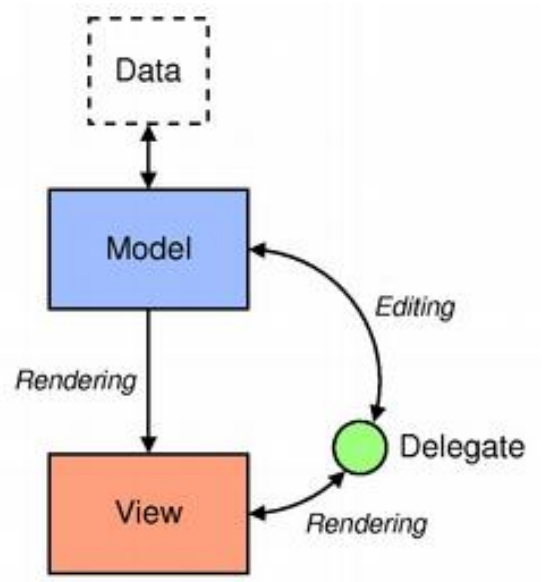


Рисунок 8.24 – Архитектура модель-представление

Если *объединить объекты представления и контроллера*, то в результате получится *архитектура модель-представление* (рис. 8.24).

Это все еще отделяет способ хранения данных от способа их представления пользователю, но обеспечивает простую структуру, основанную на тех же принципах. Такое разделение дает возможность показать пользователю одни и те же данные в различных представлениях и реализовать новые типы представлений без изменения базовой структуры данных. Чтобы обеспечить гибкость управления пользовательским вводом, мы представляем концепцию **делегата** (delegate).

Преимущество наличия делегата в этой структуре состоит в том, что это дает возможность для настройки представления и редактирования элементов данных.

Рассмотрим подробно процесс создания делегата для отображения и редактирования данных. Для этого нам понадобится создать новый класс, для чего следует щелкнуть правой кнопкой мыши на корневой папке в дереве проектов (см. рис. 8.9), затем выбрать «Добавить новый…», в появившемся окне выбрать, соответственно, «C++» и «Класс C++», затем ввести данные, как на рис. 8.25.

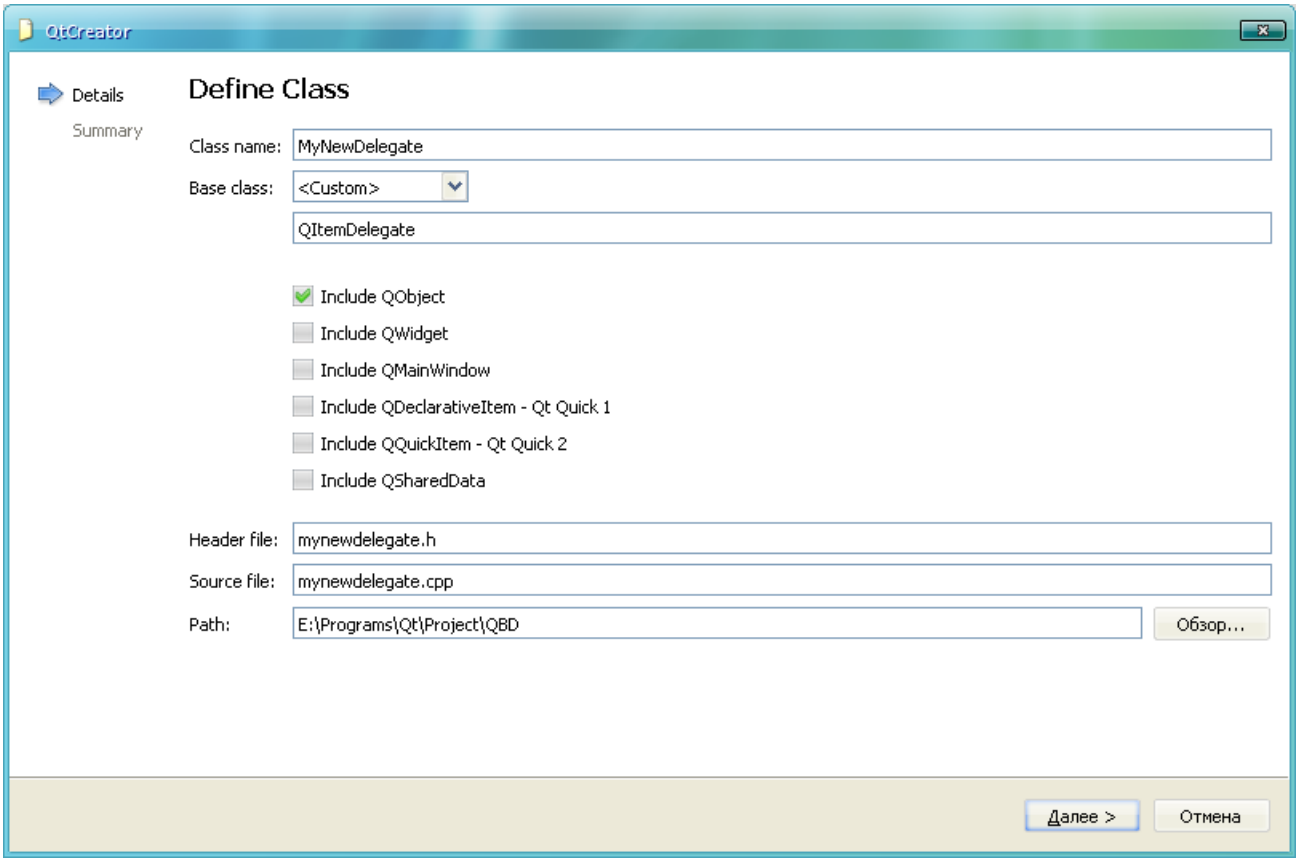


Рисунок 8.25 – Создание нового класса-делегата

Перепишем код заголовочного файла (mynewdelegate.h):

```
#ifndef MYNEWDELEGATE_H
#define MYNEWDELEGATE_H

#include <QItemDelegate>
#include <QObject>
#include <QModelIndex>
#include <QSize>
#include <QSpinBox>

class MyNewDelegate : public QItemDelegate
{
public:
    explicit MyNewDelegate(QObject *parent = 0);
    QWidget *createEditor(QWidget *parent,
                          const QStyleOptionViewItem &option,
                          const QModelIndex &index) const;

    void setEditorData(QWidget *editor,
                       const QModelIndex &index) const;

    void setModelData(QWidget *editor,
                      QAbstractItemModel *model,
                      const QModelIndex &index) const;

    void updateEditorGeometry(QWidget *editor,
                              const QStyleOptionViewItem &option,
                              const QModelIndex &index) const;

signals:
```

```
public slots:

};

#endif // MYNEWDELEGATE_H
```

и файла с исходными кодами (mynewdelegate.cpp):

```
#include "mynewdelegate.h"

MyNewDelegate::MyNewDelegate(QObject *parent) :
    QItemDelegate(parent)
{
}

QWidget *MyNewDelegate::createEditor(QWidget *parent,
                                     const QStyleOptionViewItem &option,
                                     const QModelIndex &index) const
{
    // создаем элемент
    QSpinBox *editor = new QSpinBox(parent);
    // указываем, что нижняя граница у нас будет "0" (ноль)
    editor->setMinimum(0);
    // а верхняя 100(сто)
    editor->setMaximum(100);
    // возвращаем элемент
    return editor;
}
void MyNewDelegate::setEditorData(QWidget *editor,
                                  const QModelIndex &index) const
{
    // перехватываем значения ячейки таблицы
    int value =index.model()->data(index, Qt::EditRole).toInt();
    // создаем spinBox для ввода данных, без проверки типов данных
    QSpinBox *spinbox = static_cast<QSpinBox*>(editor);
    // записывает данные из ячейки в spinBox
    spinbox->setValue(value);
}
void MyNewDelegate::setModelData(QWidget *editor,
                                 QAbstractItemModel *model,
                                 const QModelIndex &index) const
{
    // создаем элемент
    QSpinBox *spinbox = static_cast<QSpinBox*>(editor);

    // Эта функция интерпретирует
    // текст в spinbox. Если значение изменилось с момента
    // последней активации, то она будет излучать сигналы.
    spinbox->interpretText();
    // сохраняем значения счетчика в переменную
    int value = spinbox->value();
    // меняем значение по координатам на новое из счетчика
    model->setData(index, value);
}
void MyNewDelegate::updateEditorGeometry(QWidget *editor,
                                         const QStyleOptionViewItem &option,
                                         const QModelIndex &index) const
{
    // устанавливаем расположение
    editor->setGeometry(option.rect);
}
```

Опишем наш новый класс в заголовочном файле главного окна приложения (в файле mainwindow.h):

```
    MyNewDelegate *mDelegate;
```

необходимо добавить:

```
#include "mynewdelegate.h"
```

а также подключить его в конструкторе (в файле mainwindow.cpp):

```
                // наш класс
                mDelegate = new MyNewDelegate(this);
                // указываем на замену стандартного представления новым.
                ui->tableView->setItemDelegate(mDelegate);
```

Протестируйте приложение. Что изменилось?

Усложним код:

```
QWidget *MyNewDelegate::createEditor(QWidget *parent,
                                     const QStyleOptionViewItem &option,
                                     const QModelIndex &index) const

{


//    if (index.column() == 0) { // для первого столбца используем QDateTimeEdit
//            QDateTimeEdit *editor = new QDateTimeEdit(parent);
//            editor->setDisplayFormat("dd.MM.yyyy"); // формат даты
//            editor->setCalendarPopup(true);
//            editor->setDate(QDate::currentDate()); // устанавливаем дату по
умолчанию
//            return editor;

    if (index.column() == 3) { // для столбца с баллами
            // создаем элемент
            QSpinBox *editor = new QSpinBox(parent);
            // указываем, что нижняя граница у нас будет "0" (ноль)
            editor->setMinimum(0);
            // а верхняя 5(пять)
            editor->setMaximum(5);
            // возвращаем элемент
            return editor;
       }else if (index.column() == 4) { // для связи
       }
    else{ // для всех остальных полей устанавливаем QLineEdit
           QLineEdit *editor = new QLineEdit(parent);
           QString currentText = index.model()->data(index,
Qt::DisplayRole).toString();
           editor->setText(currentText);
           return editor;
       }
}
void MyNewDelegate::setEditorData(QWidget *editor,
                                  const QModelIndex &index) const
{
//    if (index.column() == 0) {
//            QDateTimeEdit *dateEditor = qobject_cast<QDateTimeEdit *>(editor);
//            if (dateEditor) {
//                dateEditor->setDate(QDate::fromString(index.model()->data(index,
Qt::EditRole).toString(),"d.MM.yyyy"));
//            }
    if (index.column() == 3) // для столбца с баллами
    {
        // создаем spinBox для ввода данных, без проверки типов данных
```

```
        QSpinBox *spinbox = static_cast<QSpinBox*>(editor);
            if (spinbox) {
                // перехватываем значения ячейки таблицы
                int value =index.model()->data(index, Qt::EditRole).toInt();
                // записывает данные из ячейки в spinBox
                spinbox->setValue(value);
            }
        }else{
            QLineEdit *textEditor = qobject_cast<QLineEdit *>(editor);
            if (textEditor) {
                textEditor->setText(index.model()->data(index,
Qt::EditRole).toString());
            }
}
}
void MyNewDelegate::setModelData(QWidget *editor,
                                 QAbstractItemModel *model,
                                 const QModelIndex &index) const
{
//    if (index.column() == 0) {
//           QDateTimeEdit *dateEditor = qobject_cast<QDateTimeEdit *>(editor);
//           if (dateEditor) {
//               model->setData(index, dateEditor->date().toString("d.MM.yyyy"));
// дату приводим к формату ДД.ММ.ГГГГ
//           }
//       }
         if (index.column() == 3)
         {
             // создаем элемент
             QSpinBox *spinbox = static_cast<QSpinBox*>(editor);
           if (spinbox) {
               // Эта функция интерпретирует
               // текст в spinbox. Если значение изменилось с момента
               // последней активации, то она будет излучать сигналы.
               spinbox->interpretText();
               // сохраняем значения счетчика в переменную
               int value = spinbox->value();
               // меняем значение по координатам, на новое, из счетчика
               model->setData(index, value);
           }
       }else{
           QLineEdit *textEditor = qobject_cast<QLineEdit *>(editor);
           if (textEditor) {
               model->setData(index, textEditor->text());
           }
       }
}
```

Самостоятельная работа:

– реализуйте контроль ввода для всех редактируемых полей.

7. Строка статуса.

```
            // напишем текст в statusBar, вторым параметром можно указывать время,
            // которое текст будет видимым на экране, если время не указать,
            // то надпись будет отражаться постоянно
// ui->statusBar->showMessage("База данных успешно загружена", 3000); //3 секунды
            ui->statusBar->showMessage("Улыбнитесь!");
```

8. Маска.

inputMask : QString – данное свойство содержит маску ввода QLineEdit.

Если маска не установлена, то inputMask() возвращает пустую строку.

Условие для текста можно использовать вместо маски или вместе с ней; см. п. 2 – setValidator(). Сброс маски и возвращение обычной работы QLineEdit осуществляется с помощью передачи пустой строки ("") или с помощью вызова setInputMask() без аргументов.

Примеры:

Маска	Описание
`000.000.000.000;_`	IP-адрес; пробелы обозначаются символом _.
`HH:HH:HH:HH:HH:HH;_`	MAC-адрес
`0000-00-00`	Дата ISO; пробелы обозначаются символом `пробел`
`>AAAAA-AAAAA-AAAAA-AAAAA-AAAAA;#`	Лицензионный номер; пробелы обозначаются символом - и все (алфавитные) символы приводятся к верхнему регистру.

9. Поиск.

Добавьте на главное окно приложения экземпляр класса QLineEdit и реализуйте реакцию на изменение текста в нем:

```
model->setFilter(QString("firstname LIKE '%%1%'").arg(arg1));
```

Самостоятельная работа:

– добавьте функцию автодополнения к полю поиска;

– реализуйте поиск по нескольким полям.

Контрольные вопросы

1. С какими СУБД может работать Qt?
2. В чем особенность СУБД SQLITE?
3. Что такое файл-серверная, клиент-серверная и встраиваемая СУБД?
4. Что означает слово «:memory:», как оно используется в программе?
5. Чем DDL отличается от DML?
6. Зачем нужен маппер?
7. Что такое реляционная БД?
8. Как устанавливается связь между таблицами в реляционной БД?
9. Что такое INNER JOIN?
10. Что такое LEFT JOIN?
11. Что такое «submitAll»? Какие бывают режимы редактирования модели?
12. Зачем необходим класс-делегат?

Тест

Укажите элемент технологии модель-представление, отвечающий за рисование каждого элемента в отдельности, а также за его редактирование:

– выделение;

– модель;

– редактор;

– представление;

– делегат.

ПРИЛОЖЕНИЕ

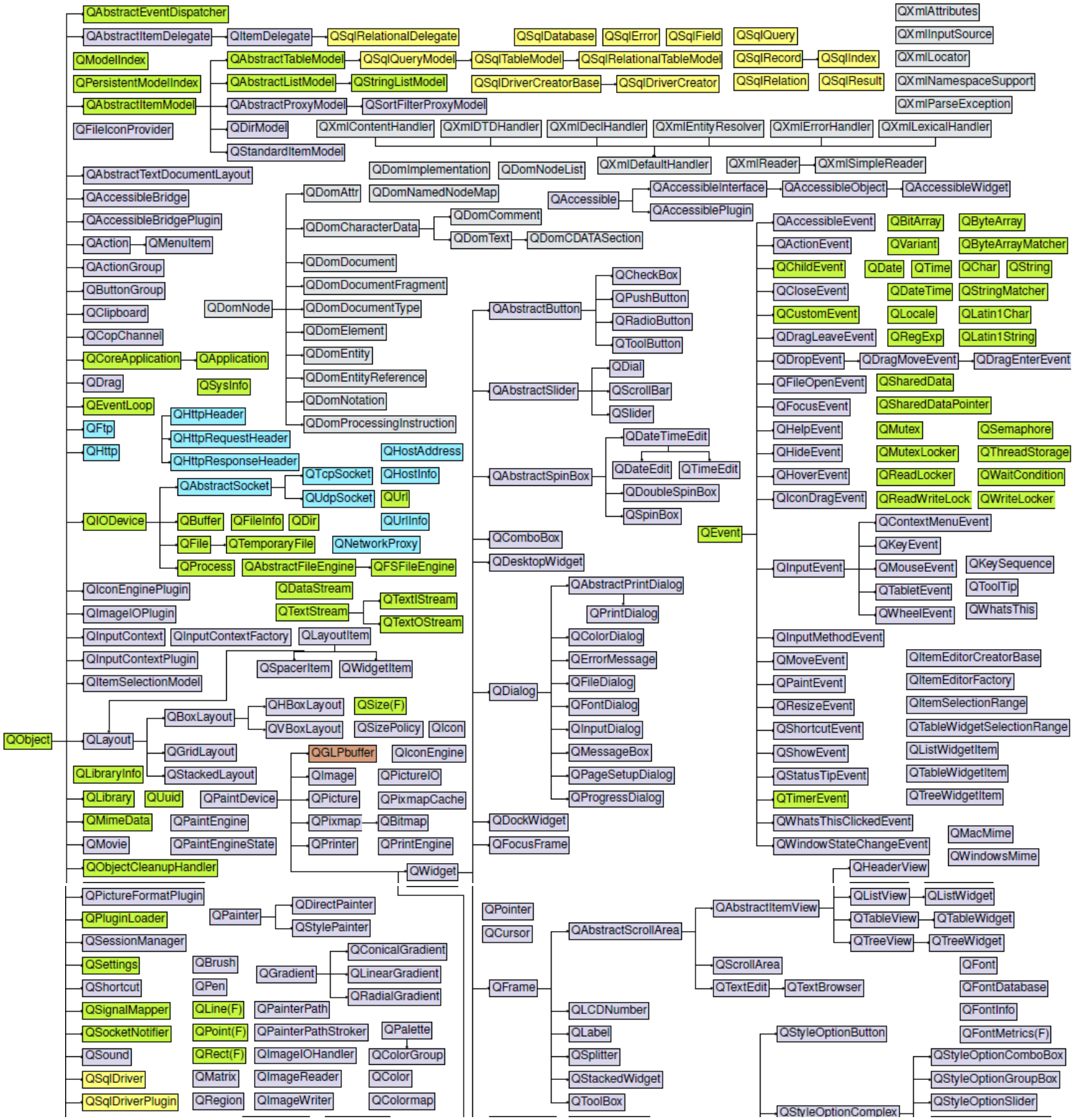


Рисунок П.1 – Обзор библиотеки классов Qt 4

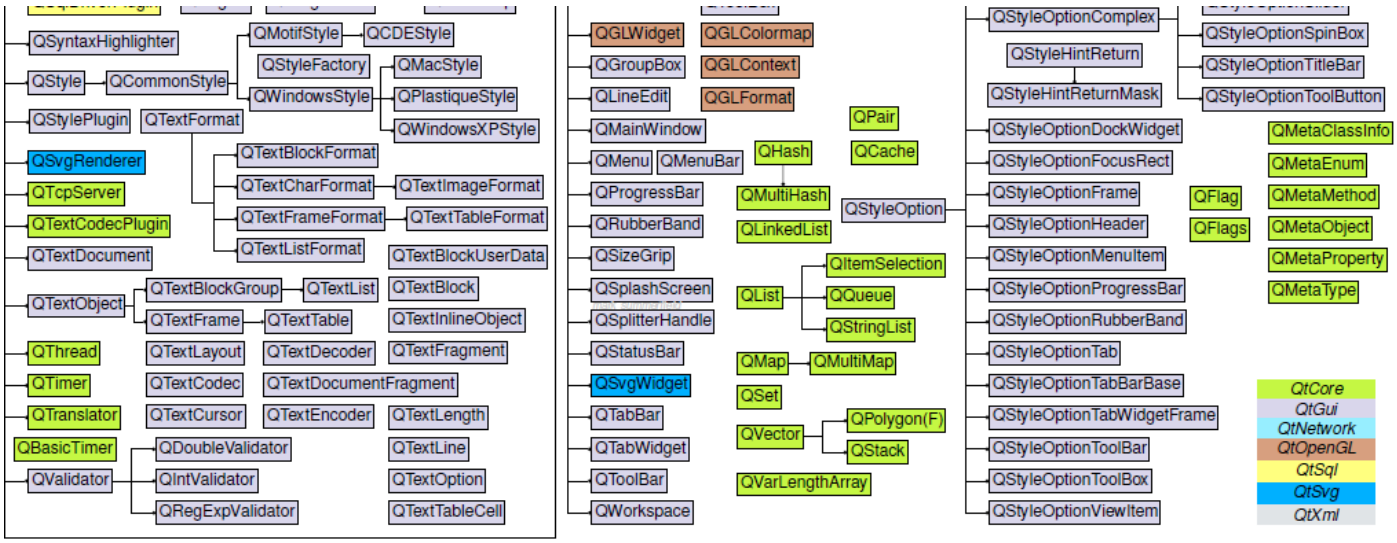


Рисунок П.1, лист 2

Кратко перечислим основные классы библиотеки Qt 4.

Модуль QtCore – ядро библиотеки Qt (невизуальные элементы):

QObject – родитель для большинства других классов библиотеки Qt;

QBitArray – битовый массив;

QByteArray – массив байтов;

QBuffer – класс, позволяющий работать с QByteArray как с потоком ввода-вывода;

QChar – 16-разрядный символ Unicode;

QCoreApplication – консольное приложение;

QDataStream – обеспечивает платформонезависимую сериализацию объектов;

QDate – работа с календарными датами;

QDateTime – дата и время;

QDir – работа с каталогами файловой системы;

QEvent – базовый класс для событий;

QFile – работа с файлами;

QFileSystemWatcher – позволяет отслеживать изменения в каталогах и файлах;

QFlags – набор битовых флагов;

QIODevice – базовый класс для логических устройств ввода-вывода;

QLatin1Char – 8-разрядный символ ASCII/Latin1;

QLatin1String – строка 8-разрядных символов в кодировке Latin1;

QLibrary – динамически загружаемая библиотека;

QLine – двумерный вектор, заданный целочисленными координатами своих концов;

QLineF – двумерный вектор с вещественными координатами;

QLinkedList – класс-шаблон для работы со связанными списками;

QList – класс-шаблон для работы со списками элементов;

QMap и QHash – словарь (массив элементов с доступом по ключу);

QMutex, QMutexLocker, QSemaphore и QWaitCondition – мьютекс, семафор и условие возобновления потока;

QReadLocker, QWriteLocker, QReadWriteLock – блокировки чтения/записи;

QPair – класс-шаблон для работы с парами элементов;

QPluginLoader – загрузка плагинов во время выполнения приложения;

QPoint – точка на плоскости, заданная своими целочисленными координатами;

QPointF – точка на плоскости с вещественными координатами;

QPointer – класс-шаблон для работы с "безопасными" указателями, автоматически обнуляемыми при уничтожении объекта, на который они указывают;

QProcess – класс для запуска сторонних процессов и взаимодействия с ними;

QPointer – класс-шаблон для работы с очередями;

QRegExp – регулярные выражения;

QSet – множество элементов;

QSettings – сохранение и загрузка настроек приложения (в том числе работа с реестром Windows и ini-файлами);

QSize – два целых числа – размер прямоугольника;

QSizeF – два вещественных числа – размер прямоугольника;

QSocketNotifier – слежение за активностью файлового дескриптора;

QStack – стек;

QString – строка Unicode;

QStringList – список строк;

QTemporaryFile – работа с временными файлами;

QTextCodec – текстовый кодек для перекодировки строк;

QTextStream – строковый поток ввода-вывода;

QThread, QThreadStorage – многопоточное программирование;

QTime – работа с временем;

QTimer – таймер;

QUrl – унифицированный указатель информационного ресурса (uniform resource locator);

QUuid – универсальный уникальный идентификатор (universally unique identifier);

QVarLengthArray – массив переменной длины;

QVariant – вариантный тип, позволяющий работать практически со всеми объектами библиотеки Qt;

QVector – класс-шаблон для работы с динамическими массивами.

Модуль QtGui – элементы интерфейса пользователя:

QWidget – базовый класс для всех визуальных элементов;

QAbstractButton – базовый класс для представления кнопок;

QAbstractScrollArea – область с полосами прокрутки;

QAbstractSlider – ползунок для ввода значения с помощью мыши;

QAbstractSpinBox – поле с кнопками инкремента/декремента для ввода числового значения;

QDialog – диалог;

QFrame – фрейм;

QLabel – текстовая метка или пиктограмма;

QLineEdit – однострочное поле для ввода строки текста;

QCheckBox – независимый элемент с двумя (или тремя) состояниями и текстовой меткой;

QComboBox – поле ввода с кнопкой выбора и раскрывающимся списком;

QDateEdit – поле для ввода даты;

QTimeEdit – поле для ввода времени;

QDateTimeEdit – поле для ввода даты и времени;

QDial – элемент ввода, по внешнему виду напоминающий поворотный регулятор громкости или спидометр;

QSpinBox – поле для ввода целого числа с кнопками инкремента/декремента;

QDoubleSpinBox – поле ввода вещественных значений с кнопками инкремента/декремента;

QFocusFrame – позволяет получить доступ к области, выходящей за пределы прямоугольника, занимаемого данным элементом;

QFontComboBox – поле ввода с кнопкой и раскрывающимся списком для выбора шрифта;

QLCDNumber – элемент для отображения числового значения, по внешнему виду напоминающий жидкокристаллический индикатор;

QMenu – вертикальное меню;

QPushButton – кнопка;

QRadioButton – зависимый переключатель (радиокнопка);

QScrollArea – элемент, который позволяет отображать свой дочерний виджет внутри области с полосами прокрутки;

QScrollBar – вертикальная или горизонтальная полоса прокрутки;

QSizeGrip – маркер в углу окна, который можно "зацепить" указателем мыши, чтобы изменить размеры окна;

QSlider – вертикальный или горизонтальный ползунок;

QToolBar – панель инструментов;

QToolButton – кнопка на панели инструментов;

QProgressBar – индикатор состояния длительного процесса;

QTabBar – многостраничный диалог с закладками;

QTabWidget – набор элементов, отображаемых на страницах диалога с закладками;

QToolBox – одна колонка элементов, отображаемых на странице диалога с закладками;

QDesktopWidget, QDesktopServices, QSystemTrayIcon – обеспечивают доступ к рабочему столу и системному лотку на панели задач;

QClipboard – работа с буфером обмена;

QFontDatabase – сведения о всех доступных шрифтах;

QSessionManager – менеджер сеансов.

Модуль QtNetwork – классы для работы с сетью:

QFtp – клиентская часть протокола FTP;

QHttp – работа с протоколом HTTP;

QHostAddress – работа с IP-адресами;

QUdpSocket – UDP-сокет;

QTcpSocket – TCP-сокет;

QTcpServer – сервер TCP;

QUrl – работа с URL;

QSslSocket – серверный и клиентский SSL-сокет.

Модуль QtSql – работа с базами данных:

QSql – пространство имен, в котором определены различные объекты, используемые при работе с модулем QtSql;

QSqlDatabase – соединение с базой данных;

QSqlDriver – абстрактный базовый класс для доступа к базам данных;

QSqlDriverCreator – шаблон класса для доступа к базе данных, с помощью которого создаются классы для работы с конкретной СУБД;

QSqlError – информация об ошибке при работе с базами данных;

QSqlField – поле таблицы БД;

QSqlIndex – индекс БД;

QSqlQuery – запрос на языке SQL;

QSqlQueryModel – модель данных, полученных в результате выполнения запроса;

QSqlRecord – одна запись в таблице БД;

QSqlResult – абстрактный интерфейс для доступа к данным;

QSqlTableModel – модель данных, доступных для редактирования, соответствующая одной таблице БД;

QSqlRelationalTableModel – модель данных, доступных для редактирования, соответствующая одной таблице БД (с поддержкой внешних ключей).

Модуль QtOpenGL – поддержка OpenGL для работы с трехмерной графикой.

Модуль QtScript – интерпретатор скриптового языка.

Модуль QtSvg – поддержка векторной графики SVG.

Модуль QtXml – работа с XML (DOM и SAX).

Модуль QtXmlPatterns – для работы с XQuery.

Модуль QtDesigner – классы для расширения Qt Designer.

Модуль QtUiTools – работа с формами, разработанными в Qt Designer.

Модуль QtAssistant, QtHelp – система помощи.

Модуль Qt3Support – поддержка старых программ, разработанных для Qt 3.

Модуль QtTest – тестирование приложений.

Модуль QtWebKit – для работы с Web-страницами.

Модуль Phonon – средства мультимедиа.

Модуль QtTest – средства тестирования.

Следующие два модуля доступны только в коммерческой версии библиотеки Qt 4 для Windows.

Модуль QAxContainer – работа с элементами ActiveX.

Модуль QAxServer – классы для разработки серверных компонентов ActiveX.

Следующий модуль имеется только в библиотеках Qt 4 для UNIX/ Linux/FreeBSD.

Модуль QtDBus – взаимодействие между процессами с использованием протокола D-Bus.

Таблица П.1 – Комбинации клавиш для редактора Qt Creator

Сочетание клавиш	Описание
Ctrl+]	Перейти к концу блока
Ctrl+U	Выделить блок
Ctrl+Shift+U	Снять выделение блока
Ctrl+I	Выровнять блок
Ctrl+<	Свернуть блок
Ctrl+>	Развернуть блок
Ctrl+/	Закомментировать блок
Ctrl+Shift+↑	Переместить строку вверх
Ctrl+Shift+↓	Переместить строку вниз
Shift+Del	Удалить строку

Во встроенном редакторе реализовано «умное» дополнение кода, вызываемое комбинацией клавиш Ctrl+<Пробел>.

Таблица П.2 – Символы формата маски

Символ	Значение
A	Требуется алфавитный символ ASCII. A-Z, a-z.
a	Разрешен, но не обязателен алфавитный симовл ASCII.
N	Требуется алфавитный символ или цифра ASCII. A-Z, a-z, 0-9.
n	Разрешен, но не обязателен алфавитный символ или цифра ASCII.
X	Требуется любой символ.
x	Разрешен, но не обязателен любой символ.
9	Требуется цифра ASCII. 0-9.
0	Разрешена, но не обязательна цифра ASCII.
D	Требуется цифра ASCII, не равная нулю. 1-9.
d	Разрешена, но не обязательна цифра ASCII, не равная нулю (1-9).
#	Разрешена, но не обязательна цифра или знак плюс/минус ASCII.
H	Требуется шестнадцатеричный символ. A-F, a-f, 0-9.
h	Разрешен, но не обязателен шестнадцатеричный символ.
B	Требуется двоичный символ. 0-1.
b	Разрешен, но не обязателен двоичный символ.
>	Все следующие алфавитные символы переводятся в верхний регистр.
<	Все следующие алфавитные символы переводятся в нижний регистр.
!	Изменение регистра отключается.
\	Используйте \ для того, чтобы отменить действие приведенных выше знаков как специальных символов и использовать их в качестве разделителей

Реентерабельность и потокобезопасность

В документации термины «реентерабельность» и «потокобезопасность» используются при обозначении классов и функций для указания того, как они могут быть использованы в многопоточных приложениях.

Потокобезопасная функция может быть вызвана одновременно из разных потоков, даже когда вызовы используют разделяемые данные, поскольку все обращения к разделяемым данным упорядочены.

Реентерабельная функция также может быть вызвана одновременно из нескольких потоков, но только, если каждый вызов использует свои собственные данные.

Таким образом, потокобезопасная функция всегда реентерабельна, но реентерабельная функция не всегда потокобезопасна.

В более широком смысле класс называется реентерабельным, если его функции-члены могут быть безопасно вызваны из нескольких потоков, пока каждый поток использует свой отдельный экземпляр класса. Класс является потокобезопасным, если его функции-члены могут быть безопасно вызваны из нескольких потоков, даже если все потоки используют один и тот же экземпляр класса.

Замечание. Классы Qt документируются как потокобезопасные, если они предназначены для работы в многопоточных приложениях. Если функция не помечена как потокобезопасная или реентерабельная, то она не должна использоваться в разных потоках. Если класс не помечен как потокобезопасный или реентерабельный, то к конкретному экземпляру класса не должно быть доступа из разных потоков.

Реентерабельность

Классы C++ часто реентерабельны просто потому, что они имеют доступ только к данным своих членов. Любой поток может вызвать функцию-член экземпляра реентерабельного класса, в то время как ни один другой поток не может вызвать функцию-член того же самого экземпляра класса в то же самое время. Например, указанный ниже класс Counter является реентерабельным:

```
class Counter
 {
 public:
     Counter() { n = 0; }

     void increment() { ++n; }
     void decrement() { --n; }
     int value() const { return n; }

 private:
     int n;
 };
```

Данный класс не является потокобезопасным, поскольку, если несколько потоков попытаются изменить поле n, результат будет не определен. Это так,

потому что операторы ++ и -- не всегда атомарны. В действительности, они обычно расширяются до трех машинных инструкций:

- загрузка значения переменной в регистр;
- увеличение или уменьшение значения регистра;
- сохранение значения регистра обратно в основной памяти.

Потоки A и B одновременно могут загрузить старое значение переменной, увеличить ее значение в регистре и сохранить значение переменной в памяти, но переменная будет увеличена только однажды!

Потокобезопасность

Поток A должен выполнить шаги 1, 2, 3 без прерывания (атомарно) прежде, чем поток B сможет выполнить те же шаги; или наоборот. Самый легкий способ создания потокобезопасного класса состоит в том, чтобы защитить весь доступ к членам данных с помощью QMutex:

```
class Counter
{
public:
    Counter() { n = 0; }

    void increment() { QMutexLocker locker(&mutex); ++n; }
    void decrement() { QMutexLocker locker(&mutex); --n; }
    int value() const { QMutexLocker locker(&mutex); return n; }

private:
    mutable QMutex mutex;
    int n;
};
```

Класс QMutexLocker автоматически запирает мьютекс в своем конструкторе и отпирает его в деструкторе, вызываемом при завершении функции. Запирание мьютекса гарантирует, что обращения из разных потоков будут упорядочены. Член данных mutex объявлен как mutable, потому что позволяет запереть и отпереть мьютекс в функции value(), которая является константной.

Замечания по классам Qt

Большинство классов Qt реентерабельны, но не потокобезопасны, поскольку такая реализация потребовала бы дополнительных издержек на многократные блокировки и разблокировки QMutex. Например, QString реентерабелен, но не потокобезопасен. Вы можете смело обращаться к различным экземплярам класса QString из нескольких потоков одновременно, но вы не сможете спокойно получить доступ к одному и тому же экземпляру QString из нескольких потоков одновременно (если вы самостоятельно не обеспечиваете защиту от доступа с помощью QMutex).

Некоторые классы и функции Qt потокобезопасны. Это, главным образом, связанные с потоками классы (например, QMutex) и фундаментальные функции (например, QCoreApplication::postEvent()).