Министерство образования Республики Беларусь
Учреждение образования «Витебский государственный университет имени П.М. Машерова»
Кафедра прикладной математики и механики

С.В. Сергеенко, М.Г. Семенов

Программирование: библиотека Qt

Методические рекомендации по выполнению лабораторных работ

Витебск
ВГУ имени П.М. Машерова
2016

УДК 004.42(075.8)
ББК 32.973я73
С32

Печатается по решению научно-методического совета учреждения образования «Витебский государственный университет имени П.М. Машерова». Протокол № 2 от 24.12.2015 г.

Авторы: старшие преподаватели кафедры прикладной математики и механики ВГУ имени П.М. Машерова **С.В. Сергеенко, М.Г. Семенов**

Р е ц е н з е н т :
доцент кафедры автоматизации технологических процессов и производства УО «ВГТУ», кандидат технических наук, доцент *В.Е. Казаков*

Сергеенко, С.В.

С32 Программирование: библиотека Qt : методические рекомендации по выполнению лабораторных работ / С.В. Сергеенко, М.Г. Семенов. – Витебск : ВГУ имени П.М. Машерова, 2016. – 38 с.

В методических рекомендациях описаны особенности применения инструментов библиотеки Qt при разработке приложений с графическим пользовательским интерфейсом. Даны краткие теоретические сведения, приведены описания лабораторных работ со списком индивидуальных заданий. Кроме того для основных тем рассмотрены различные примеры, отражающие основные принципы проектирования оконных приложений.

Предназначается для студентов специальностей «Прикладная информатика», «Прикладная математика» (дисциплина «Программирование»).

УДК 004.42(075.8)
ББК 32.973я73

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Данные методические рекомендации посвящены разработке оконных приложений с графическим пользовательским интерфейсом. В базовых курсах «Методы алгоритмизации и программирования» (специальности «Прикладная информатика») и «Программирование» (специальности «Прикладная математика», 1 семестр), которые изучаются студентами до библиотеки Qt, рассматривается объектно-ориентированный язык C++. Классически разработка графического интерфейса для оконных windows-приложений осуществляется посредством функций WinAPI. Тем не менее, в настоящее время существует множество альтернативных подходов, базирующихся на использовании сторонних библиотек. Одной из них является библиотека Qt. Эта библиотека активно применяется многими известными IT-компаниями (например, Sennheiser, BlackBerry, Jolla, Thales, ABB, PitneyBowes и др.)

В рамках рекомендаций рассматриваются такие темы, как общая структура Qt-проекта, механизм сигналов и слотов, классы основных визуальных компонентов и менеджеров компоновок, способы реализации меню и панели инструментов, фреймворк Interview, принципы разработки одно- и многодокументных интерфейсов, реализация технологии Drag’n’Drop, работа с буфером обмена и базами данных. По каждой из приведенных выше тем описаны лабораторные работы и приводятся индивидуальные задания к ним. Наряду с теоретическим материалом и лабораторными работами, в данных методических рекомендациях приводится множество практических примеров с комментариями. Стоит отметить, что реализация данных примеров предусматривает использование последней на момент написания рекомендаций версии библиотеки Qt – пятой.

Материал соответствует отдельным темам рабочих программ курса «Программирование» (специальности «Прикладная математика» и «Прикладная информатика»).

1 ПЕРВЫЕ ШАГИ В QT

1.1 Установка и настройка библиотеки Qt и IDE“Qt Creator”

Наряду с коммерческими вариантами Qt на официальном сайте библиотеки (http://www.qt.io/ru) предлагается также и открытая лицензия (LGPL). Скачать программу установки для использования открытой лицензии IDE “Qt Creator” можно на странице http://www.qt.io/ru/download-open-source/. Здесь доступны как несколько вариантов установщиков. Предусмотрены варианты установки среды Qt Creator,которая будет использовать компилятор IDE “Visual Studio” версии 2010 или выше. Однако рекомендуем обратить внимание на версию, использующую компилятор MinGW (на момент написания пособия последняя версия – Qt 5.5.1 for Windows 32-bit (MinGW 4.9.2)).

Процесс установки не должен вызвать особых вопросов. Сначала необходимо создать учетную запись или зайти, используя уже существующую. Далее выбрать компоненты установки. На этом этапе необходимо убедиться, что в качестве одного из элементов установки указан компилятор MinGW. После необходимо ознакомиться с лицензионным соглашением и завершить установку.

1.2 Структура проекта Qt. Общие рекомендации по разработке проектов с использованием библиотеки Qt

В проект Qt входят файл конфигурации проекта (*.pro), исходные файлы (*.cpp) и заголовочные файлы (*.h). В начале файла конфигурации производиться подключение необходимых модулей. Для подключения модуля используется конструкция `QT +=` (например, `QT += core gui widgets`). В рамках данного пособия мы будем подробно говорить о следующих модулях Qt:

- **QtCore** – ядро Qt. Базовый модуль, который содержит класс QObject, контейнерные классы, классы для ввода и вывода, классы моделей интервью, классы для работы с датой и временем и др., но не содержит классов относящихся к интерфейсу пользователя.
- **QtGui** – модуль базовых классов для программирования пользовательского интерфейса. Содержит класс QGuiApplication, который поддерживает механизм цикла событий, позволяет получить доступ к буферу обмена, позволяет изменять форму курсора мыши и т.д.
- **QtWidgets** – модуль содержащий классы, которые являются “деталями” при реализации пользовательского интерфейса. В его состав входит класс QWidget – базовый класс всех элементов управления

библиотеки Qt, классы автоматического размещения элементов, классы меню, классы диалоговых окон и окон сообщений, классы для рисования, а также классы всех стандартных элементов управления.

- **QtSql** – модуль отвечающий за поддержку работы с базами данных.

Кроме перечисленных выше модулей библиотека Qt поддерживает также: QtNetwork для работы с сетями, QtOpenGL для работы с графикой OpenGL, QtTest содержащий вспомогательные классы для тестирования кода, QtXML отвечающий за поддержку XML и многие другие.

Далее в файле конфигурации проекта можно задать следующие параметры (в виде ПАРАМЕТР = ЗНАЧЕНИЕ):

TARGET – имя приложения. Если данное поле не заполнено, то название программы будет соответствовать имени проектного файла;

TEMPLATE – указывает тип проекта. Например: app – приложение, lib – библиотека.

FORMS –список файлов с расширением ui. Эти файлы создаются программой Qt Designer и содержат описание интерфейса пользователя в формате XML.

Кроме того, в файле проекта указывается список заголовочных файлов (ключевое слово HEADERS) и файлов исходного кода (ключевое слово SOURCES) в виде КЛЮЧЕВОЕ_СЛОВО += ИМЯ_ФАЙЛА1 ИМЯ_ФАЙЛА2 и т.д. При добавлении в проект файлов, QtCreator автоматически вносит изменения в файл проекта. Например, при добавлении в пустой проект HelloWorldфайла `main.cpp`, в файле HelloWorld.proпоявится строка `SOURCES += main.cpp`.

Также в файле проекта допустимо устанавливать следующие параметры:

LIBS – список библиотек, которые должны быть подключены для создания исполняемого модуля;

CONFIG – настройки компилятора;

DESTDIR – директория, в которую будет помещен исполняемый файл;

INCLUDEPATH – каталог, содержащий заголовочные файлы;

и другие.

При разработке проекта файлы классов лучше всего разбивать на две отдельные части. Часть определения класса помещается в заголовочный файл *.h, а реализация класса – в файл с расширением .cpp. В заголовочном файле с определением класса необходимо указывать директиву препроцессора `#ifndef`или `#pragmaonce`. Например,

```
#ifndef _MyClass_h_
#define _MyClass_h_
class MyClass {
    ...
```

```
};
#endif //_MyClass_h_
```

Такой подход применяется во избежание конфликтов в случае, когда заголовочный файл включается в исходные файлы более одного раза. По традиции заголовочный файл, как правило, носит имя содержащегося в нем класса.

Для ускорения компиляции, при использовании в заголовочных файлах указателей на типы данных рекомендуется предварительно объявлять эти типы, а не напрямую включать их определения посредством директивы #include.

В случае, когда предполагается использование механизма сигналов и слотов или приведение типов при помощи функции `qobject_cast<T>()`, рекомендуется в начале определения класса указать макрос`Q_OBJECT`. Например,

```
classMyClass : publicQObject {
    Q_OBJECT
    public:
    MyClass();
    ...
};
```

Основную программу рекомендуется размещать в отдельном файле. В этом файле должна быть реализована функция `main`(). В связи с этим, в качестве имени для такого файла зачастую выбирают `main.cpp`.

1.3 Первая программа на Qt

Приступим к написанию первой программы на Qt. Создайте новый проект («Другой проект» –> «Empty qmake project»). Проект назовите HelloWorld, выберите компилятор и завершите создание проекта. В файл HelloWorld.pro запишите следующий код:

```
QT += core gui widgets

TARGET = HelloWorld
TEMPLATE = app
```

В контекстном меню проекта нажмите «Добавить новый» и выберите C++ Source File. Назовите его main. В проекте должен появиться каталог исходных файлов, содержащий main.cpp. Обратите внимание на то, что в файл HelloWorld.pro автоматически добавились следующие строки:

```
SOURCES += \
    main.cpp
```

Это означается, что файл main.cpp, находящийся в корне каталога проекта, включен в проект. Теперь наполните файл main.cpp следующим содержимым:

```
#include<QtWidgets>

int main(int argc, char** argv){
    QApplication app(argc, argv);
    QLabel* lbl = new QLabel("Hello World!");
    lbl->show();
    return app.exec();
}
```

В первой строке функции `main` создается объект класса `QApplication`, который осуществляет контроль и управление приложением. Любая использующая Qt-программа с графическим интерфейсом должна создавать только один объект этого класса, и он должен быть создан до использования операций, связанных с пользовательским интерфейсом.

Затем создается объект класса `QLabel`. После создания элементы управления Qt по умолчанию невидимы, и для их отображения необходимо вызвать метод `show()`. Объект класса `QLabel` является основным управляющим элементом приложения, что позволяет завершить работу приложения при закрытии окна элемента.

Наконец, в последней строке программы приложение запускается вызовом `QApplication::exec()`. С его запуском приводится в действие цикл обработки событий, который передает получаемые от системы события на обработку соответствующим объектам. Он продолжается до тех пор, пока либо не будет вызван статический метод `QCoreApplication::exit()`, либо не закроется окно последнего элемента управления. По завершению работы приложения метод `QApplication::exec()` возвращает значение целого типа, содержащее код, информирующий о его завершении.

2 ПРОЕКТИРОВАНИЕ ГРАФИЧЕСКОГО ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

2.1 Механизм сигналов и слотов

Во время своей работы, приложение реагирует на различные события. Для того чтобы реализовать необходимую “реакцию” на определенное событие могут быть использованы различные подходы. В библиотеке Qt таким подходом является механизм сигналов и слотов. Сигналами в Qt являются методы, которые предназначены для пересылки сообщений. Слоты – это методы, которые описывают “реакцию” приложения на сигналы. Для использования механизма сигналов и слотов в описании класса, сразу на следующей строке после ключевого слова class, должен находится макрос `Q_OBJECT`, а сам класс должен быть унаследован от `QObject`. После макроса `Q_OBJECT` не должно стоять точки с запятой.

Сигналы описываются как пустые методы класса после ключевого слова “signals”. Например,

```
signals:
class MySignalClass : public QObject {
    Q_OBJECT
    ...
    signals:
    void mySignal();
    ...
};
```

Для подачи сигнала используется ключевое слово `emit`. Например, `emitmySignal()`.

Слоты в отличие от сигналов могут быть определены как `public`, `private` или `protected`. Соответственно, перед каждой группой слотов необходимо указать: `privateslots:`, `protectedslots:` или `publicslots:`. Например,

```
publicslots:
class MySlotClass : public QObject {
    Q_OBJECT
    …
    public slots:
    void mySlot()
    {
        …
    }
};
```

Каждый сигнал может быть связан с произвольным количеством слотов, а каждый слот – с произвольным количеством сигналов. Для связи сигнала со слотом используется метод `connect()` класса `QObject`.

```
QObject::connect(const QObject*   sender,
                 const char* signal,
                 const QObject* receiver,
                 const char*   slot,
    Qt::ConnectionType type = Qt::AutoConnection
    );
```

`sender` – указатель на объект, отправляющий сигнал;

`signal` – это сигнал, с которым осуществляется соединение. Прототип (имя и аргументы) метода сигнала должен быть заключен в специальный макрос `SIGNAL(method());`

`receiver` – указатель на объект, который имеет слот для обработки сигнала;

`slot` – слот, который вызывается при получении сигнала. Прототип слота должен быть заключен в специальном макросе `SLOT(method());`

`type` – управляет режимом обработки.

Пример: `QObject::connect(MySignalClass, SIGNAL(mySignal()), MySlotClass, SLOT(mySlot()));`

Если вызов метода `connect()` происходит из потомка класса `QObject`, тогда `QObject::` можно опустить. Если сигнал или слот находится в классе, из которого происходит вызов `connect()`, то можно опустить первый или третий параметр соответственно. Например, если указанное выше соединение осуществляется в классе `MySlotClass`, то вызов `connect()` можно записать в следующем виде: `connect(MySignalClass, SIGNAL(mySignal()), SLOT(mySlot()));`.

В некоторых ситуациях возникает необходимость в разъединение сигналов и слотов, для этих целей может быть использован статический метод `disconnect()` класса `QObject`, набор параметров которого аналогичен параметрам метода `connect()`.

2.2 Класс QWidget

При проектировании графического пользовательского интерфейса с использованием библиотеки Qt основным “строительным материалом” являются виджеты. За работу с виджетами отвечает класс `QWidget`. Он унаследован от класса `QObject`и является предком для таких элементов графического интерфейса как: главное и диалоговые окна приложения, кнопки, флажки, переключатели, слайдеры, полосы прокрутки, меню,

выпадающие списки и многие другие. Наследование класса QObject позволяет использовать механизм сигналов и слотов, а также организовывать объектные иерархии. Элементы расположенные на виджете считаются его потомками. В иерархии такого рода виджеты, которые не являются потомками, называются виджетами верхнего уровня и обладают собственными окнами.

Конструктор класса QWidget выглядит следующим образом:

```
QWidget(QWidget* pwgt = 0, Qt::WindowFlags f = 0)
```

Первый параметр задает предка создаваемого виджета в объектной иерархии. Второй параметр отвечает за внешний вид окна. Туда могут быть переданы объединенные с типом окна побитовой операцией "|". Для задания типа окна используются следующие константы: Qt::Window, Qt::Tool, Qt::ToolTip, Qt::Popup, Qt::SplashScreen; а в качестве модификаторов: Qt::WindowSystemMenuHint, Qt::FramelessWindowHint, Qt::WindowMinimizeButtonHintи другие. Как можно заметить, для получения виджета верхнего уровня мы можем использовать конструктор без параметров.

Основные методы класса QWidget:

- setWindowTitle() – устанавливает заголовок окна;
- size(), height() и width() – возвращают размеры виджета;
- resize()– позволяет установить или изменить размеры виджета;
- pos()– возвращает позицию окна;
- move()– перемещает окно;
- setGeometry() – объединяет методы move и resize.

2.3 Размещение элементов. Менеджеры компоновки

Для построения гибкого и легко масштабируемого графического интерфейса удобно использовать менеджеры компоновки, которые управляют размещением виджетов на окне. Корневым в иерархии классов отвечающих за работу менеджеров компоновок является QLayout. От него унаследованы классы QBoxLayoutи QGridLayout. QBoxLayout является предком для классов QHBoxLayout и QVBoxLayout, которые отвечают за горизонтальное и вертикальное размещение виджетов соответственно. Класс QGridLayoutв свою очередь обеспечивает размещение виджетов в виде таблицы.

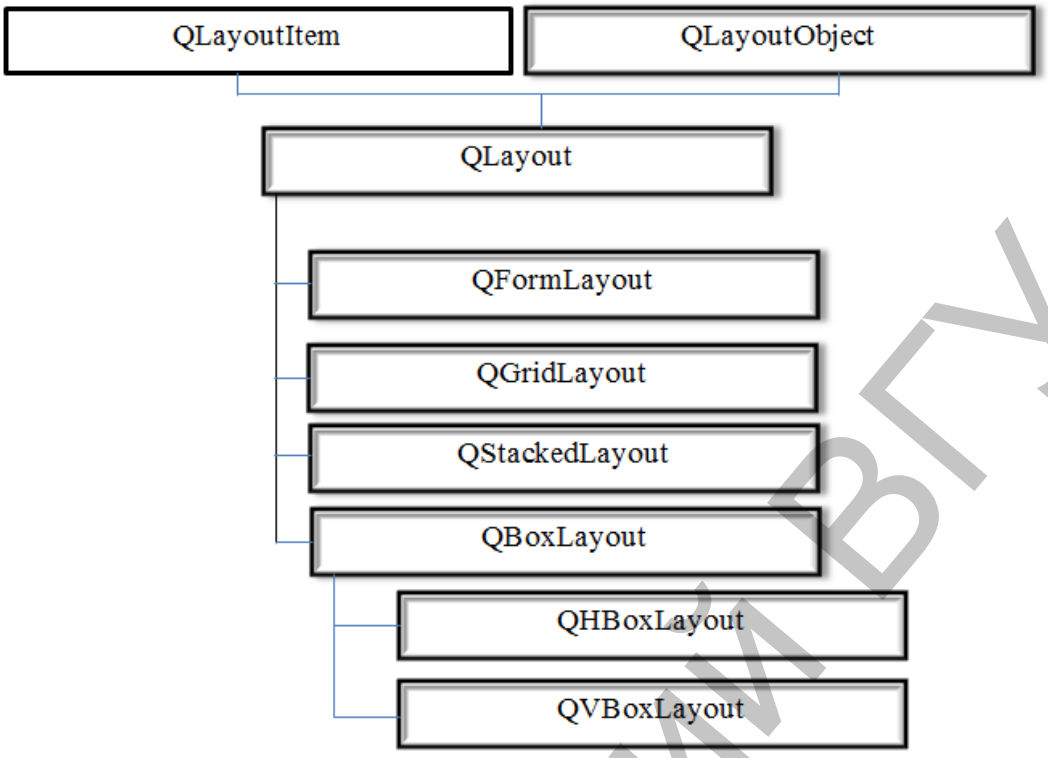

Рис 2.3.1 Иерархия классов менеджеров компоновок.

Некоторые общие методы менеджеров компоновок:

- `setSpacing()` устанавливает расстояние между виджетами;
- `setMargin()` устанавливает отступ виджетов от границы области;
- `addWidget()` добавляет виджет в компановку;
- `addLayout()` позволяет встраивать другие менеджеры компоновок.

При использовании класса `QBoxLayout` или его потомков, для указания способа размещения первым параметром в конструктор можно передать одно из следующих значений:

- `LeftToRight` для горизонтального размещения слева направо;
- `RightToLeft` для горизонтального размещения справа налево;
- `TopToBottom` для вертикального размещения сверху вниз;
- `BottomToTop` для вертикального размещения снизу вверх.

В табличном размещении (`QGridLayout`) при добавлении элемента указывается его позиция (номер строки и столбца). Для этого применяется метод `addWidget(QWidget * widget, int fromRow, int fromColumn, int rowSpan, int columnSpan, Qt::Alignment alignment = 0)`. Параметры `fromRow` и `fromColumn` содержат информацию о том, куда необходимо добавить элемент, параметры `rowSpan` и `columnSpan` – о количестве строк и столбцов, занимаемых данным элементом, а параметр `alignment` – о выравнивании.

2.4 Стандартные визуальные компоненты

Можно выделить следующие типы визуальных компонентов: элементы отображения, кнопки, флажки, переключатели, элементы настройки, элементы ввода и элементы выбора.

Элементы отображения. Основным классом для создания элемента отображения является `QLabel`. Отображаемая этим классом информация может быть текстовой, графической или анимацией. Для задания содержимого определены методы `setText()`, `setPixmap()` и `setMovie()`.

Кнопки, флажки и переключатели. Базовым классом для всех видов кнопок является класс `QAbstractButton`. От него наследуются следующие классы:

- `QPushButton` – кнопка, на которую можно нажать.
- `QCheckBox` – флажок, который имеет два состояния: выбран или не выбран.
- `QRadioButton` – переключатель, который также как и предыдущий имеет два состояния. Однако переключатели обычно формируются группами по два и более и, при выборе одного из них, остальные автоматически входят в состояние “не выбран”.
- `QToolButton` – кнопка быстрого доступа. Обычно используется для добавления на панель инструментов.

Установить текст на кнопке можно либо передав его в конструктор, либо с помощью метода `setText()`. Кроме текста кнопке можно задать растровое изображение при помощи метода `setIcon()`.

Класс `QAbstractButton` предоставляет следующие сигналы:

- `pressed()` – отправляется при нажатии на кнопку;
- `released()` –отправляется, когда “отпускаем” кнопку;
- `clicked()` – отправляется после того, как на кнопку нажали и отпустили;
- `toggled()` – отправляется при изменении состояния кнопки (только для кнопок имеющих статус переключателя).

Для класса `QPushButton` определен метод `setCheckable()`, который позволяет создать так называемый выключатель (togglebutton).

Для класса `QCheckBox` существует два основных вида флажков – обычный флажок (normalcheckbox) и флажок с неопределенным состоянием (tristatecheckbox). Состояние задается с помощью метода `setChecked()`. Флажки с неопределенным состоянием задаются при помощи метода `setTristate()`, а третье состояние задается передачей константы `Qt::PartiallyChecked` в метод `setCheckState()`.

Как уже оговаривалось выше, переключатели QRadioButton группируются по два и более. Для подобного рода группировки удобно использовать класс QGroupBox.

Полосы прокрутки, слайдеры, установщики. Базовым для данных элементов настройки является класс QAbstractSlider. Для изменения ориентации следует применить слот setOrientation() и передать в него константу Qt::Horizontal или Qt::Vertical. Для установки диапазона значений используется метод setRange(), передав через запятую минимальное и максимальное значение, или методами setMinimum() и setMaximum(). Величина единичного шага задается методом setSingleStep(), а страничного – методом setPageStep(). Для получения значения слайдера применяется метод value(), а для установки значения – setValue(). Сигналы valueChanged() и sliderMoved() отправляются при изменении значения слайдера. От QAbstractSliderунаследованы классы QSlider (ползунок), QScrollBar (полоса прокрутки) и QDial (установщик).

Класс QSlider позволяет отображать метки (шкалы), которые дают пользователю визуальное представление о значении слайдера. Позиция меток задается методом setTickPosition() с помощью следующих констант: NoTicks (без меток), TicksAbove (сверху), TicksBelow (снизу) и TicksBothSides (сверху и снизу). Шаг для прорисовки меток устанавливается методом setTickInterval(). Применение класса QDial схоже с QSlider, но визуально первый представляет собой круглый регулятор.

Элементы ввода. Обычное однострочное поле ввода определяется классом QLineEdit, а многострочное – QPlainTextEdit. Получить текст из поля ввода можно методом text(), установить – методом setText(). При изменении текст отправляется сигнал textChanged().

Поля ввода со стрелками, предназначенными для увеличения или уменьшения значения, будем называть счетчиками. Абстрактным классом для счетчиков является QAbstractSpinBox. От него унаследованы классы QSpinBox (счетчик для целых значений), QDoubleSpinBox (счетчик для значений типа double) и QDateTimeEdit (счетчик для даты). Для всех видов счетчиков реализованы методы stepUp() и stepDown(), которые эмулируют нажатие на кнопки стрелок.

Элементы выбора. К элементам выбора относят простые и выпадающие списки. Простой список реализован в классе QListWidget. Элементы списка могут содержать текст и растровые изображения. Для добавления элемента в список можно воспользоваться методом addItem(). Элементы списка представлены классом

QListWidgetItem. В список можно добавить сразу несколько текстовых элементов, передав объект класса QStringList, содержащий список строк, в метод insertItems(). Для вставки одного элемента (текстового или объекта класса QListWidgetItem) в произвольное место определен метод insertItem(). Стоит отметить, что элементами списка могут служить и другие виджеты. Для этой цели в классе QListWidget определены методы setItemWidget() и itemWidget().

2.5 Класс действия QAction. Панель инструментов. Меню. Главное окно приложения

Зачастую, при проектировании современного графического интерфейса, многие операции (например, копирование, вставка, сохранение и др.) доступны одновременно через главное меню, панель инструментов, контекстное меню, комбинации горячих клавиш и т.д. Для упрощения разработки подобных приложений в библиотеке Qt существует класс QAction. Этот класс хранит следующую информацию о действии:

- название операции (установка – метод setText);
- текст всплывающей подсказки (setToolTip);
- текст подсказки “Что это” (setWhatsThis);
- “горячие” клавиши (setShortcut);
- ассоциированные пиктограммы (setIcon);
- текст подсказки в строке состояния (setStatusTip).

Панель инструментов предназначена для быстрого доступа к необходимому действию. Класс панели инструментов в Qt – QToolBar. Установка панели инструментов осуществляется при помощи метода addToolBar(). Добавить действие на панель инструментов можно при помощи метода addAction(), а виджет – методом addWidget().

За создание меню в Qt отвечает класс QMenu. Основные методы:

- addAction – добавить действие;
- addSeparator – добавить разделитель;
- addMenu – добавить подменю.

После создания меню, его можно поместить стандартную панель QMenuBar (метод addMenu).

Класс QMainWindow позволяет создавать главное окно приложения. Он содержит такие элементы как: главное меню, секции для панели инструментов, строку состояния, главную рабочую область. Указатель на виджет главного меню можно получить с помощью метода menuBar(), а на рабочую область – метод centralWidget(). Установить основной виджет рабочей области можно методом setCentralWidget().

2.6 Лабораторная работа № 1 «Графический интерфейс»

Разработайте приложение, которое средствами библиотеки Qt предоставляет пользователю интерфейс и решает задачу вашего варианта. Для выбора действия пользователю должны быть предоставлены пункты меню и кнопки панели инструментов. Эти элементы управления должны определяться объектами класса `QAction`.

Решение задачи оформить в виде отдельной функции, которая все необходимые ей данные получает в качестве параметров и не использует функций библиотеки Qt. Получение данных введённых пользователем и представление результата следует (но не обязательно) оформить в виде отдельных подпрограмм.

Функции Windows API не использовать и заголовочный файл windows.h не подключать.

Варианты заданий.

1. Даны три целых числа: *A*, *B* и *C*. Найдите среди них
   - наибольшее значение,
   - наименьшее значение.
2. Даны целые числа *A* и *B*. Не используя операторов «/», «%» необходимо вычислить
   - неполное частное,
   - остаток от их деления.
3. Даны комплексные числа *A* + *B***i** и *C* + *D***i**. Необходимо найти их
   - сумму,
   - разность.
4. Даны целые числа *A* и *B*. Необходимо вычислить их
   - наибольший общий делитель,
   - наименьшее общее кратное.
5. Даны целые числа *A*, *B* и *C*, проверить, можно ли из них составить арифметическую прогрессию. Если да, составьте из них
   - возрастающую,
   - убывающую прогрессию.
6. Даны дроби *A*/*B* и *C*/*D* . Необходимо найти их
   - произведение,
   - частное.
7. Даны вектора (*A*, *B*) и (*C*, *D*). Необходимо найти их
   - сумму,
   - разность.
8. Даны три целых числа: *A*, *B* и *C*. Найдите среди них

• количество положительных значений,
• количество отрицательных значений.

9. Даны целые числа *A* и *B*. Необходимо
• не используя оператор “*” вычислить их произведение,
• не используя функции “pow” вычислить A^B.

10. Даны целые числа *A, B* и *C*, проверить, можно ли из них составить геометрическую прогрессию. Если да, составьте из них
• возрастающую прогрессию,
• убывающую прогрессию.

11. Даны дроби *A/B* и *C/D* . Необходимо найти их
• сумму,
• разность.

12. Даны комплексные числа *A* + *B***i** и *C* + *D***i**. Необходимо найти их
• произведение,
• частное.

13. Даны три положительных числа A, B и C. Проверьте, существует ли треугольник, у которого длины сторон равны данным числам. Если такой треугольник существует, то найдите его
• площадь,
• периметр.

2.7 Пример

Решим следующую задачу:

Даны целые числа A и B. Необходимо вычислить их
• *сумму,*
• *разность,*
• *произведение,*
• *частное.*

Создайте пустой проект с именем Lab1-example. В файл проекта Lab1-example.pro добавьте следующий код:

```
QT += core gui widgets

TARGET = Lab1-example
TEMPLATE = app
```

Основной функционал реализуем в классе Calculator. Добавьте к проекту заголовочный файл Calculator.h. В этот файл запишите следующий код:

```
#ifndef CALCULATOR
#define CALCULATOR

#include <QMainWindow>
//Перечисление используемых классов
class QLineEdit;
class QComboBox;
class QLabel;
class QPushButton;
class QWidget;
// =====================================================
class Calculator : public QMainWindow {
    Q_OBJECT

private:
//Основной виджет приложения
    QWidget *centralWidget;
//Поля для ввода чисел
    QLineEdit* pleFirstNum;
    QLineEdit* pleSecondNum;
//Выпадающий список для выбора операции
QComboBox* pcbAction;
//Кнопка «Выполнить»
QPushButton* ppbCalc;
//Статический текс с результатом вычислений
QLabel* plResult;

public:
Calculator(QWidget* pwgt = 0);
//Функции выполняющие вычисления
double add(double a, double b);
    double sub(double a, double b);
    double mul(double a, double b);
    double div(double a, double b);

publicslots:
//Слот нажатия на кнопку
voidslotCalcClicked();
};
```

#endif //CALCULATOR

Добавьте к проекту файл Calculator.cpp, который будет содержать реализацию методов класса Calculator.

```
#include <Calculator.h>
#include <QtWidgets>
//Конструктор
Calculator::Calculator(QWidget* pwgt) :
QMainWindow(pwgt)
{
//Инициализация полей ввода
    pleFirstNum = new QLineEdit();
    pleFirstNum->setMaximumWidth(30);
    pleSecondNum = new QLineEdit();
    pleSecondNum->setMaximumWidth(30);
//Инициализация выпадающего списка действий
    pcbAction = new QComboBox();
    pcbAction->setMaximumWidth(30);
    pcbAction->addItems(QStringList()
    << "+" << "-" << "/" << "*");
//Инициализация кнопки
    ppbCalc = new QPushButton("=");
    ppbCalc->setMaximumWidth(30);
    connect(ppbCalc, SIGNAL(clicked(bool)),
SLOT(slotCalcClicked()));
//Инициализация надписи для результата
    plResult = new QLabel("0");
    plResult->setMaximumWidth(60);
//Настройка размещения и установка центрального
виджета
QVBoxLayout* phbxLayout = new QVBoxLayout(this);
    phbxLayout->addWidget(pleFirstNum);
    phbxLayout->addWidget(pcbAction);
    phbxLayout->addWidget(pleSecondNum);
    phbxLayout->addWidget(ppbCalc);
    phbxLayout->addWidget(plResult);
    phbxLayout->setContentsMargins(100, 100, 100,
100);
    centralWidget = new QWidget();
    centralWidget->setLayout(phbxLayout);
    setCentralWidget(centralWidget);
}
//Методы для вычисления результата
```

```
double Calculator::add(double a, double b)
{
    return a + b;
}

double Calculator::sub(double a, double b)
{
    return a - b;
}

double Calculator::mul(double a, double b)
{
    return a * b;
}

double Calculator::div(double a, double b)
{
    return a / b;
}
//Слот нажатия кнопки
voidCalculator::slotCalcClicked()
{
//Считывание информации из полей ввода
double a = pleFirstNum->text().toDouble();
    double b = pleSecondNum->text().toDouble();
    double result = 0;
//Получение выбранной операции
    QString act = pcbAction->currentText();
//Вызов соответствующего метода
    if (act == "+") result = add(a, b);
    else if (act == "-") result = sub(a, b);
    else if (act == "/") result = div(a, b);
    else if (act == "*") result = mul(a, b);
    else {
        plResult->setText("err");
        return;
    }
//Отображение результата
    plResult->setText((new QString)->setNum(result));
}
```

Теперь создадим файл `main.cpp`, содержащий функцию `main()`.

```
#include <QtWidgets>
```

```
#include "Calculator.h"

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

    Calculator calculator;
    calculator.setWindowTitle("Calculator");
    calculator.resize(300,300);
    calculator.show();
    return app.exec();
}
```

Запустите приложение и проверьте его работу.

Теперь реализуем поддержку действий, меню и панели инструментов. Для этого добавим в файл Calculator.h предварительное описание следующих классов:

```
class QAction;
class QMenu;
classQToolBar;
```

Кроме того, добавим описание следующих элементов:

```
//Описание объектов действий
QAction* pactAdd;
QAction* pactSub;
    QAction* pactMul;
    QAction* pactDiv;
//Описание объектов пунктов меню
QMenu* pmnuAdd;
QMenu* pmnuSub;
    QMenu* pmnuMul;
    QMenu* pmnuDiv;
    QMenu* pmnuActions;
//Описание объекта панели инструментов
    QToolBar* ptbTools;
```

и описание слотов для работы с действиями:

```
void slotAdd();
    void slotSub();
    void slotMul();
void slotDiv();
```

Теперь в реализацию конструктора класса Calculator добавим инициализацию действий, меню и панели инструментов.

```
//Сумма
    pactAdd = new QAction(this);
    pactAdd->setText("&+");
    pactAdd->setShortcut(QKeySequence("CTRL++"));
    pactAdd->setToolTip("Сумма");
    connect(pactAdd, SIGNAL(triggered()),
SLOT(slotAdd()));
//Разность
    pactSub = new QAction(this);
    pactSub->setText("&-");
    pactSub->setShortcut(QKeySequence("CTRL+-"));
    pactSub->setToolTip("Разность");
    connect(pactSub, SIGNAL(triggered()),
SLOT(slotSub()));
//Произведение
    pactMul = new QAction(this);
    pactMul->setText("&*");
    pactMul->setShortcut(QKeySequence("CTRL+*"));
    pactMul->setToolTip("Произведение");
    connect(pactMul, SIGNAL(triggered()),
SLOT(slotMul()));
//Частное
    pactDiv = new QAction(this);
    pactDiv->setText("&/");
    pactDiv->setShortcut(QKeySequence("CTRL+/"));
    pactDiv->setToolTip("Частное");
    connect(pactDiv, SIGNAL(triggered()),
SLOT(slotDiv()));
//Создание меню
    pmnuActions = new QMenu("&Действия");
    pmnuActions->addAction(pactAdd);
    pmnuActions->addAction(pactSub);
    pmnuActions->addAction(pactMul);
    pmnuActions->addAction(pactDiv);
    menuBar()->addMenu(pmnuActions);
//Создание панели инструментов
    ptbTools = new QToolBar();
    ptbTools->addAction(pactAdd);
    ptbTools->addAction(pactSub);
    ptbTools->addAction(pactMul);
ptbTools->addAction(pactDiv);
```

```
    addToolBar(Qt::LeftToolBarArea, ptbTools);
```

Нам осталось только реализовать слоты для работы действий. Для этого мы воспользуемся уже реализованным слотом `slotCalcClicked()`, перед вызовом которого будем менять значение выпадающего списка `pcbAction` на соответствующее необходимому действию.

```
void Calculator::slotAdd()
{
    pcbAction->setCurrentText("+");
    slotCalcClicked();
}
void Calculator::slotSub()
{
    pcbAction->setCurrentText("-");
    slotCalcClicked();
}
void Calculator::slotDiv()
{
    pcbAction->setCurrentText("/");
    slotCalcClicked();
}
void Calculator::slotMul()
{
    pcbAction->setCurrentText("*");
    slotCalcClicked();
}
```

3 РАЗРАБОТКА ОДНО- И МНОГОДОКУМЕНТНОГО ПРИЛОЖЕНИЯ

3.1 Фреймворк Interview

Разделение способов хранения данных и их представления позволяет использовать одинаковые виджеты для представления данных из различных источников, а также отображать одни и те же данные разными способами одновременно. Кроме того этот подход позволяет изменять представление и модель данных в значительной степени независимо: необходимо лишь сохранять интерфейс взаимодействия.

Для разделения модели и представления данных библиотекой Qt предоставляется Interview Framework. Эта часть библиотеки реализует измененный шаблон проектирования MVC (модель-представление-контролёр).

Этот фреймворк (каркаса) включает в себя три основных разновидности классов:

– Классы модели наследуются от класса QAbstractItemModel, служат для предоставления остальным частям фреймворка интерфейса доступа к данных из некоторого источника.

– Классы представления наследуются от класса QAbstractItemView, служат для отображения данных пользователю. Данные получают от модели. Для указания на конкретные данные используются индексы модели.

– Классы делегаты наследуются от QAbstractItemDelegate, используются стандартными представлениями для отображения и редактирования элементов данных.

Библиотека Qt предоставляет ряд классов моделей и представления.

Модели:

– QStandardItemModel – общая модель хранения произвольных данных;

– QFileSystemModel – модель файловой системы; она позволяет получать атрибуты файлов и каталогов, а также содержимое каталогов;

– QStringListModel – модель предоставляющая представлению строки;

– QSortFilterProxyModel – модель которая предоставляет возможность сортировки и отбора данных из другой модели;

– а также классы для работы с базами данных.

– Абстрактные классы для разных типов моделей.

– QAbstractTableModel – абстрактный класс, позволяющий реализовать табличную модель данных;

– QAbstractListModel – абстрактный класс, позволяющий реализовать модель данных в виде списка.

– Представления:

– QTableView – табличное представление данных;

– QTreeView – древовидное представление данных;

– QListView – представление данных в виде списка;

– QComboBox может использовать модели для получения списка элементов.

Виджеты QListWidget, QTreeWidget и QTableWidget являются вариантами соответствующих представлений, содержащими внутри себя модели и источники данных. Эти виджеты предоставляют доступ к внутренним моделям.

3.2 Лабораторная работа № 2 «Однодокументный интерфейс»

Будем называть словом последовательность букв и цифр. Будем называть путём последовательность слов, разделённых пробелами.

Разработайте приложение с реализованным на Qt графическим пользовательским интерфейсом, которое предоставляет пользователю следующие возможности:

- Загружать список путей из файла. Каждый путь списка записывается в файле в отдельной строке. Пустые строки следует игнорировать.
- Отображать его сжато в виде дерева. Каждому узлу дерева соответствует некоторый путь. Корень дерева соответствует пустому пути и не отображается. Путь соответствующий не корневому узлу равен пути предка, к которому через пробел дописали справа название узла. Названия дочерних элементов одного предка должны быть уникальными.
- Добавлять к дереву путей новые узлы.
- В строке состояния должно быть отображено:
  - постоянно количество узлов дерева без потомков (такие узлы называются листьями)
  - при выборе элемента дерева временно полный путь до него.
- Доступ к действиям должен быть предоставлен через пункты меню и панель инструментов.

На рисунке 3.2.1 показан пример внешнего вида приложения после

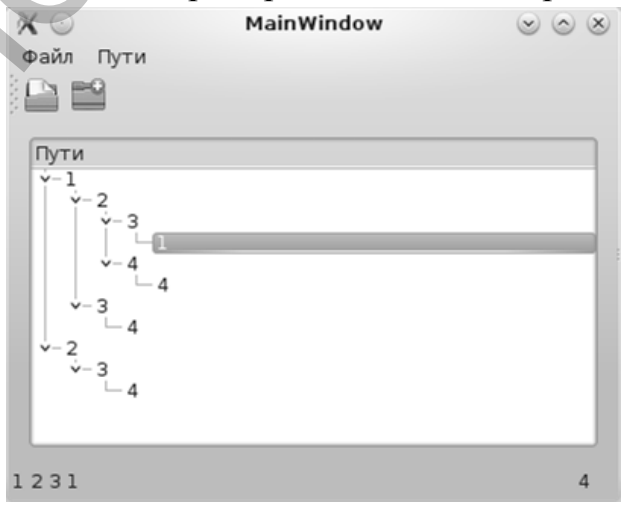

Рис. 3.2.1 Вид окна приложения

открытия файла следующего содержания:

1 2 3
2 3 4
1 3 4
1 2 4 4
1 2 3 1

3.3 Лабораторная работа № 3 «Многодокументный интерфейс»

Ознакомьтесь с документацией по классам QMdiArea, QMdiSubWindow, QSignalMapper и с примером «MDI Example» (см. соответствующий раздел документации библиотеки Qt и проект examples/mainwindows/mdi в каталоге установки библиотеки Qt).

Измените разработанное в предыдущей лабораторной работе приложение так, чтобы оно предоставляло многодокументный интерфейс. У внутренних окон нет меню, панели инструментов и строки состояния, все действия осуществляются через соответствующие элементы главного окна. Приложение должно позволять работать с несколькими окнами.

4 ВЗАИМОДЕЙСТВИЕ СО СРЕДОЙ ИСПОЛНЕНИЯ. МЕХАНИЗМЫ ОБМЕНА ИНФОРМАЦИЕЙ

4.1 Технология Drag'n'Drop

При работе с технологией Drag'n'Drop следует рассматривать следующие события:

- void dragEnterEvent(QDragEnterEvent *)
  курсор мыши попадает внутрь данного виджета при перетаскивании
- void dragLeaveEvent(QDragLeaveEvent *)
  курсор мыши покидает границы данного виджета при перетаскивании
- void dragMoveEvent(QDragMoveEvent *)
  курсор мыши перемещается внутри границ виджета при перетаскивании
- void dropEvent(QDropEvent *)
  была отпущена кнопка мыши, когда курсор находился в границах данного виджета при перетаскивании
- void mouseMoveEvent(QMouseEvent *)
  курсор мыши перемещается внутри границ виджета
- void mousePressEvent(QMouseEvent *)
  была нажата кнопка мыши, когда курсор находился в границах данного виджета

Перетаскивание обычно начинается в следующих случаях:

- при нажатии ЛКМ на объекте;
- при нажатии ЛКМ на объекте и удержании её в течении некоторого времени (QApplication::startDragTime());
- при нажатии ЛКМ на объекте и перемещении курсора мыши с нажатой клавишей на некоторое расстояние (QApplication::startDragDistance()).

Для того что бы выполнить перетаскивание на стороне источника необходимо:

- Создать объект класса QMimeData в динамически распределяемой памяти (с использованием оператора new). Удаляется библиотекой Qt.
- Задать созданному объекту передаваемые данные и указать их тип (можно регистрировать свои типы данных).
- Создать объект класса QDrag в динамически распределяемой памяти (с использованием оператора new). Удаляется библиотекой Qt.

Указать созданному объекту на объект класса QMimeData, содержащий перетаскиваемые данные.

- Задать прочие свойства перетаскивания (отображаемый курсор, отображаемое изображение перетаскиваемого объекта, позиция курсора относительно изображения).
- Вызвать метод QDrag::exec, указав ему допустимые действия.
- Выполнить нужные операции в соответствии с возвращённым значением выбранного действия (только те которые надо предпринять на стороне источника).

Возможные действия описываются значениями из перечисления Qt::DropAction:

- Qt::CopyAction – создание копии перетаскиваемого объекта
- Qt::MoveAction – перемещение перетаскиваемого объекта
- Qt::LinkAction – создание ссылки перетаскиваемого объекта
- Qt::IgnoreAction – отсутствие действия

Важные методы в классе QDropEvent:

- acceptProposedAction – установить предложенное действие в качестве действия предпринимаемого при “бросании” и принять событие
- dropAction – предпринимаемое действие при “бросании” объекта
- mimeData – данные перетаскиваемого объекта
- possibleActions – возможные действия
- proposedAction – предложенное действие
- setDropAction – установить действие предпринимаемое при “бросании” объекта
- source – виджет-источник перетаскиваемого объекта, если объект перетаскивается в рамках одной программы
- accept – принять событие
- ignore – игнорировать событие (игнорируемое событие будет передаваться родительскому виджету)

Для того чтобы виджет получал события о перетаскивании над нимнекоторого объекта необходимо выполнить setAcceptDrops(true). Для получения более подробной информации по этой теме изучите документацию по следующим методам и классам:

- QAbstractItemView::setDragEnabled
- QWidget::setAcceptDrops
- QAbstractItemView::setDropIndicatorShown
- QWidget::dragEnterEvent
- QWidget::dropEvent
- QDragEnterEvent
- QDropEvent
- QMimeData

- QUrl

Продемонстрируем работу с этой технологией в Qt на примере приложения, которое моделирует ханойские башни, представляемые виджетом с вертикальным размещением элементов. Каждый диск изображается меткой обозначающей его размер. Интерфейс программы позволяет снимать с башни верхний диск и перемещать его на другую башню так, чтобы под ним не оказался диск меньшего размера.

Файл tower.h содержит определение класса Tower, который отвечает за работу отдельной башни.

```
#ifndef TOWER_H
#define TOWER_H
#include <QDragEnterEvent>
#include <QSpacerItem>
#include <QVBoxLayout>
#include <QStack>
#include <QWidget>
/* Класс Tower отвечает за отображение одной из башен
   и дисков на ней. Кроме того, этот класс отвечает
за
   возможность перетаскивания дисков */
class Tower : public QWidget
{
Q_OBJECT
public:
    explicit Tower(QWidget *parent = 0);
/* Помещает, если допустимо, диск указанного размера
       на вершину башни*/
    bool addDisk(int size);
protected:
/* Обработка события нажатия кнопки мыши.
       Определяет начало перетаскивания */
    void mousePressEvent(QMouseEvent *);
    /* Обработка события внесения перетаскиваемого
    объекта в виджет */
void dragEnterEvent(QDragEnterEvent *event);
    /* Обработка события бросания в виджет */
    void dropEvent(QDropEvent *event);
private:
    QVBoxLayout *verticalLayout;
    QSpacerItem *verticalSpacer;
    QStack<int> diskSizes;
    bool canPut(int size);
    bool isAcceptable(const QDropEvent *event);
```

```
    int eventData(const QDropEvent *event, bool *ok);
};
#endif // TOWER_H
```

Файл tower.cpp содержит определения методов класса Tower.

```
#include "tower.h"
#include <QWidget>
#include <QLabel>
#include <QMimeData>
#include <QPixmap>
#include <QDrag>
Tower::Tower(QWidget *parent) :
    QWidget(parent)
{
    resize(400,300);
    setAcceptDrops(true);
    verticalLayout = new QVBoxLayout(this);
    verticalSpacer = new QSpacerItem(20, 289,
                                     QSizePolicy::Minimum,

QSizePolicy::Expanding);
    verticalLayout->addItem(verticalSpacer);
}
bool Tower::addDisk(int size)
{
    if (!canPut(size))
        return false;
    QString name = QString::number(size);
    QLabel *disk = new QLabel(name, this);
    disk->setAlignment(Qt::AlignCenter);
    verticalLayout->insertWidget(1, disk);
    diskSizes.push(size);
    return true;
}
void Tower::mousePressEvent(QMouseEvent *)
{
    if (diskSizes.empty())
        return;
    QWidget *item = verticalLayout->itemAt(1)-
>widget();
    QLabel *disk = static_cast<QLabel *>(item);
    QMimeData *diskData = new QMimeData;
    diskData->setText(disk->text());
    int diskWidth = disk->width();
```

```
    int diskHeight = disk->height();
    QPixmap diskPixmap(diskWidth, diskHeight);
    disk->render(&diskPixmap);
    QDrag *drag = new QDrag(this);
    drag->setMimeData(diskData);
    drag->setPixmap(diskPixmap);
    drag->setHotSpot(QPoint(diskWidth/2,
diskHeight/2));
    if (drag->exec() == Qt::MoveAction) {
        verticalLayout->removeWidget(disk);
        diskSizes.pop();
        delete disk;
    }
}
void Tower::dragEnterEvent(QDragEnterEvent *event)
{
    if (isAcceptable(event))
        event->acceptProposedAction();
}
void Tower::dropEvent(QDropEvent *event)
{
    bool ok;
    int diskSize = eventData(event, &ok);
    if (ok && addDisk(diskSize))
        event->acceptProposedAction();
}
bool Tower::canPut(int size)
{
    return diskSizes.empty() || diskSizes.top() >
size;
}
bool Tower::isAcceptable(const QDropEvent *event)
{
    bool ok;
    int size = eventData(event, &ok);
    if (!ok)
        return false;
    return canPut(size);
}
int Tower::eventData(const QDropEvent *event, bool
*ok)
{
    *ok = false;
    QObject *source = event->source();
```

```
    if (source == 0 || source == this)
        return 0;
    const QMimeData *mimeData = event->mimeData();
    if (!mimeData->hasText())
        return 0;
    return mimeData->text().toInt(ok);
}
```

Файл mainwindow.h содержит определение класса MainWindow отвечающего за основное окно программы.

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
#include <QWidget>
#include <QHBoxLayout>
#include "tower.h"
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    explicit MainWindow(QWidget *parent = 0);
private:
    QWidget *centralWidget;
    QHBoxLayout *horizontalLayout;
    Tower *tower_1;
    Tower *tower_2;
    Tower *tower_3;
};
#endif // MAINWINDOW_H
```

Файл mainwindow.cpp содержит определения методов класса MainWindow и вспомогательные функции.

```
#include "mainwindow.h"
#include <QFrame>
#include <QVBoxLayout>
#include <QHBoxLayout>
Tower * newTower(QWidget *centralWidget)
{
    QFrame *frame = new QFrame(centralWidget);
    frame->setFrameShape(QFrame::StyledPanel);
    frame->setFrameShadow(QFrame::Raised);
    Tower *tower = new Tower(frame);
    QVBoxLayout *layout = new QVBoxLayout(frame);
```

```
    layout->addWidget(tower);
    centralWidget->layout()->addWidget(frame);
    return tower;
}
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent)
{
    resize(400, 300);
    centralWidget = new QWidget(this);
    horizontalLayout = new
QHBoxLayout(centralWidget);
    tower_1 = newTower(centralWidget);
    tower_2 = newTower(centralWidget);
    tower_3 = newTower(centralWidget);
    setCentralWidget(centralWidget);
    for (int size = 3; size > 0; --size)
tower_1->addDisk(size);
}
```

Файл main.cpp содержит точку входа в программу – функцию main, которая отвечает за создание главного окна программы и запуск цикла обработки событий.

```
#include <QApplication>
#include "mainwindow.h"
int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

Файл проекта обычно имеет расширение .pro и содержит информацию о типе проекта, используемых модулях

```
QT += core gui widgets
TEMPLATE = app
TARGET = htower
SOURCES += \
    main.cpp \
    tower.cpp \
    mainwindow.cpp
HEADERS += \
    tower.h \
    mainwindow.h
```

4.2 Лабораторная работа № 4 «Drag'n'Drop»

Дополните разработанное в лабораторной работе № 3приложение так, чтобы оно позволяло

- перетаскивать узлы дерева вместе со всеми их потомками между дочерними окнами,
- отрывать файл, содержащий список путей, в новом дочернем окне с помощью перетаскивания этого файла в родительское окно.

4.3 Лабораторная работа № 5 «Буфер обмена»

Изучите документацию по классам QClipboard, QMimeData и QLabel.

Разработайте приложение, которое

1. отображает содержащиеся в буфере обмена
   - текст
   - изображение поддерживаемого формата
2. обновляет своё содержимое при изменении буфера обмена
3. позволяет помещать в буфер обмена
   - текст
   - изображение

4.4 Основы использования реляционных баз данных

Основные понятия:

- Отношение (таблица) – множество кортежей с одинаковыми наборами атрибутов.
- Кортеж (запись) – набор значений, соответствующих атрибутам (каждому атрибуту в кортеже соответствует ровно одно значение).
- Атрибут (поле) – пара, состоящая из имени и приписываемого атрибуту типа. Атрибуту могут соответствовать только те значения, которые принадлежат его типу.
- Первичный ключ отношения – некоторая часть набора атрибутов отношения, значения которой однозначно идентифицируют кортеж. То есть в отношении не может быть двух кортежей у которых значения, соответствующие атрибутам первичного ключа, одного совпадали бы с соответствующими значениями другого.
- Внешний ключ отношения – некоторая часть набора атрибутов отношения по которым кортежи должны совпадать с кортежами из другого отношения.

- Язык SQL – Structured Query Language (язык структурированных запросов). – язык позволяющий описывать отношения, связи между ними и запросы на выборку, изменение, добавление и удаление данных в реляционных базах данных.

Взаимодействие с системами управления реляционными базами данных, поддерживающими SQL предоставляется в модуле `QtSql`, который подключается добавлением следующей строки в начале *.pro файла:

```
QT += sql
```

Основные классы модуля `QtSql`:

- `QSql` – содержит различные идентификаторы используемые в модуле `QtSql`;
- `QSqlDatabase` – представляет соединение с базой данных;
- `QSqlDriver` – абстрактный базовый класс доступа к конкретным базам данных;
- `QSqlDriverCreator` – шаблонный класс для создания объектов классов наследующих от `QSqlDirever`;
- `QSqlDriverCreatorBase` – базовый класс для `QSqlDriverCreator`;
- `QSqlError` – информация об ошибке;
- `QSqlRecord` – инкапсулирует запись базы данных;
- `QSqlField` – управление полями отношений и представлений базы данных;
- `QSqlIndex` – управление индексами базы данных и их описание;
- `QSqlQuery` – средства для запуска и управления операторами SQL;
- `QSqlResult` – абстрактный интерфейс для получения данных из базы;
- `QSqlQueryModel` – модель для получения доступа только на чтение к результатам выполнения SQL запросов;
- `QSqlTableModel` – редактируемая модель для одной таблицы базы данных;
- `QSqlRelationalTableModel` – редактируемая модель с поддержкой внешних ключей для одной таблицы базы данных.

4.5 Лабораторная работа № 6 «Работа с базами данных»

Разработайте приложение, которое позволяет отображать и редактировать с учетом внешних ключей содержимое базы данных. Приложение должно иметь многодокументный интерфейс.

База данных состоит из двух отношений: «Product» (товар), «Order_» (заказ). Отношение «Product» содержит следующие атрибуты:

- code (код товара),
- name (наименование товара),
- unit (наименование единицы измерения товара),
- price (цена единицы товара).

Первичный ключ этого отношения состоит из атрибута code. Отношение «Order_» содержит следующие атрибуты:

- id (номер заказа),
- product (код товара),
- amount (заказанное количество единиц товара).

Первичный ключ этого отношения состоит из атрибута id. Внешний ключ этого, атрибут product, связан с атрибутом code отношения «Product». При отображении заказа вместо кода товара должно отображаться его наименование.

4.6 Динамические библиотеки

Динамические библиотеки представляют собой модули программы загружаемые в пространство процесса из отдельного файла. Такая загрузка может происходить как во время запуска или загрузки модуля программы, так и во время выполнения программы.

Символ – имя, связанное с некоторым стартовым адресом. В некоторых случаях в качестве имени символа может выступать число.

Для разработки и использования динамических библиотек используются:

- макрос Q_DECL_EXPORT для экспорта символа
- макрос Q_DECL_IMPORT для импорта символа
- класс QLibrary для загрузки библиотеки во время выполнения программы

Основные методы класса QLibrary

- setFileName позволяет задать имя файла, содержащего загружаемую библиотеку
- load – загрузить библиотеку
- resolve – получить адрес символа
- unload – выгрузить библиотеку

Существует отдельный тип библиотек, называемых расширениями (plugins) Qt. Эти библиотеки позволяют загружать код объекта, реализующего некоторый интерфейс. Класс этого объекта должен быть потомком QObject, так как при загрузке используется механизм мета-объектной информации.

Для разработки и использования расширений (plugins) используются:

- макрос Q_DECLARE_INTARFACE позволяет связать класс-интерфейс с уникальным строковым идентификатором (используется при разработке расширений)
- макрос Q_INTERFACES позволяет указать, какие интерфейсы реализует данный класс (используется при разработке расширений)
- макрос Q_PLUGIN_METADATA – осуществить экспорт расширения
- класс QPluginLoader для загрузки библиотеки специального вида (плагина) во время выполнения программы.

Основные методы класса QPluginLoader:

- setFileName позволяет задать имя файла, содержащего загружаемую библиотеку
- load – загрузить библиотеку
- instance – получить экземпляр плагина
- unload – выгрузить библиотеку

Для упрощения разработки и использования динамических библиотек удобно использовать заголовочный файл следующего содержания:

```
#include <QtCore / QtGlobal>
#if defined (MYSHAREDLIB_LIBRARY)
#define MYSHAREDLIB_EXPORT Q_DECL_EXPORT
#else
#define MYSHAREDLIB_EXPORT Q_DECL_IMPORT
#endif
```

Это позволит использовать одни и те же заголовочные файлы как при разработке, так и при использовании библиотеки:

```
#include "mysharedlib_global.h"
MYSHAREDLIB_EXPORT void foo();
class MYSHAREDLIB_EXPORT MyClass...
```

При этом в файле проекта библиотеки необходимо указать

```
DEFINES += MYSHAREDLIB_LIBRARY
```

4.7 Лабораторнаяработа № 7 «Динамические библиотеки»

Измените программу, разработанную в рамках задания лабораторной работы № 1, так чтобы функции, отвечающие за решение задачи, загружались из динамической библиотеки. Реализуйте два приложения с разными вариантами загрузки библиотеки:

1. Динамическая загрузка с использованием класса QLibrary.
2. Подключение при запуске программы указанных при компиляции библиотек и их функций.

СПИСОК ЛИТЕРАТУРЫ

1. Шлее, М. Qt. Профессиональное программирование на C++: руководство / М. Шлее [пер. с нем.]. – СПб.: БХВ-Петербург, 2005. – 544 с.

2. Саммерфилд, М. Qt. Профессиональное программирование / М. Саммерфилд – СПб.–М., 2011 – 552 с.

3. Боровский, А. Qt 4.7. Практическое программирование на C++ / А. Боровский – СПб., 2012 – 491 с.

4. Страуструп, Б. Язык программирования C++ / Б. Страуструп. – М.: Бином, 2005. – 1099 с.

5. Кунцевич, С.П. Языки С и C++: Практикум по программированию / С.П. Кунцевич. – Витебск, 2004. – 64 с.

Учебное издание

СЕРГЕЕНКО Сергей Владимирович
СЕМЕНОВ Максим Геннадьевич

ПРОГРАММИРОВАНИЕ: БИБЛИОТЕКА QT

Методические рекомендации
по выполнению лабораторных работ

Технический редактор *Г.В. Разбоева*
Компьютерный дизайн *Т.Е. Сафранкова*

Подписано в печать .2016. Формат 60x84 $^{1}/_{16}$. Бумага офсетная.
Усл. печ. л. 2,26. Уч.-изд. л. 1,36. Тираж экз. Заказ .

Издатель и полиграфическое исполнение – учреждение образования
«Витебский государственный университет имени П.М. Машерова».

Свидетельство о государственной регистрации в качестве издателя,
изготовителя, распространителя печатных изданий
№ 1/255 от 31.03.2014 г.

Отпечатано на ризографе учреждения образования
«Витебский государственный университет имени П.М. Машерова».
210038, г. Витебск, Московский проспект, 33.