М. Г. Городничев
Т. Д. Фатхулин
Х. А. Джабраилов

Разработка кроссплатформенного программного обеспечения

Учебное пособие по направлениям подготовки бакалавров
09.03.01 Информатика и вычислительная техника,
02.03.02 Фундаментальная информатика и информационные технологии

МИНИСТЕРСТВО ЦИФРОВОГО РАЗВИТИЯ, СВЯЗИ И МАССОВЫХ КОММУНИКАЦИЙ РОССИЙСКОЙ ФЕДЕРАЦИИ
Ордена Трудового Красного Знамени федеральное государственное бюджетное образовательное учреждение высшего образования
Московский технический университет связи и информатики

Факультет «Информационные технологии»
Кафедра «Математическая кибернетика и информационные технологии»

М. Г. Городничев, Т. Д. Фатхулин, Х. А. Джабраилов

РАЗРАБОТКА КРОССПЛАТФОРМЕННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебное пособие по направлениям подготовки бакалавров
09.03.01 Информатика и вычислительная техника,
02.03.02 Фундаментальная информатика и информационные технологии

Санкт-Петербург
Наукоемкие технологии
2023

УДК 004.432
ББК 32.973.2
Г70

Рецензент:
д-р техн. наук, проф. кафедры «Автоматизация производственных процессов»
Московского автомобильно-дорожного государственного технического университета
(МАДИ) *Дмитрий Наумович Суворов*

Г70 **Городничев М. Г.**
Разработка кроссплатформенного программного обеспечения. Учебное пособие по направлениям подготовки бакалавров 09.03.01 Информатика и вычислительная техника, 02.03.02 Фундаментальная информатика и информационные технологии / М. Г. Городничев, Т. Д. Фатхулин, Х. А. Джабраилов, МТУСИ. – СПб: Наукоемкие технологии, 2023. – 148 с.

ISBN 978-5-907804-04-3

В данном учебном пособии представлены основы создания кроссплатформенного программного обеспечения на языке программирования C++ с использованием фреймворка Qt. Рассматриваются элементы управления, применяемые в Qt. Показаны основные методы работы с датой и временем, текстовыми и бинарными файлами, аудиоплеером, собственными стилями оформления приложений, многопоточным программированием, компьютерными сетями, базами данных, а также готовыми кроссплатформенными программными продуктами, сопровождающиеся пояснениями и иллюстрациями.

Учебное пособие ориентировано на бакалавров, обучающихся по направлениям 09.03.01 – Информатика и вычислительная техника, и 02.03.02 – Фундаментальная информатика и информационные технологии.

УДК 004.432
ББК 32.973.2

ISBN 978-5-907804-04-3

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

На современном этапе развития информационных технологий существует множество различных платформ, в том числе и мобильных, на которых пользователи запускают прикладное программное обеспечение (ПО). Необходимо учитывать, что используемые архитектуры процессоров и операционные системы (ОС) могут сильно варьироваться на разных устройствах. Все это затрудняет разработчикам программного обеспечения реализацию конечного продукта для пользователя, т. к. необходимо предоставить одинаковый функционал и быстродействие на всех платформах.

Фреймворк Qt — инструмент разработки кроссплатформенного программного обеспечения. Он используется для написания ПО на таких языках программирования, как C, C++ и Python. Фреймворк Qt состоит из набора специализированных программных средств, а также дополнительных библиотек классов, позволяющих создавать многофункциональный, удобный и красивый пользовательский интерфейс. ПО, разработанное при помощи фреймворка Qt, является переносимым на различные платформы, которые поддерживаются фреймворком. Важно отметить, что кроссплатформенность ПО реализуется на уровне исходного кода. Фреймворк Qt позволяет создавать кроссплатформенное ПО для ОС Linux, Windows, Android и MacOS.

Такие популярные коммерческие программные продукты и решения, как редактор изображений Adobe Photoshop Album, мессенджер Skype, мультиплеер VLC Media Player и браузер Opera были разработаны с использованием фреймворка Qt и входящих в него библиотек классов. В настоящем учебном пособии студентам предлагается на основе разнообразных примеров разрабатываемых приложений освоить современные возможности языка программирования C++, позволяющие создавать кроссплатформенное программное обеспечение.

Учебное пособие содержит восемь лабораторных работ, которые позволяют в полной мере на практике сформировать у студентов навыки по работе с фреймворком Qt и по созданию ПО с удобным графическим пользовательским интерфейсом, позволяющим: воспроизводить аудио, использовать различные стили оформления приложений, осуществлять ввод-вывод данных для проведения математических расчетов, использовать многопоточное программирование, реализовывать сетевое взаимодействие посредством сокетов и HTTP-запросов, а также создавать базы данных (БД) и управлять ими. В заключении приведены способы создания независимо переносимого ПО.

При выполнении лабораторных работ следует использовать примеры листингов программного кода, которые приведены в Приложении. Нумерация листингов соответствует разделам учебного пособия.

1. ПРОСТЕЙШИЕ ЭЛЕМЕНТЫ УПРАВЛЕНИЯ. РАБОТА С ДАТОЙ И ВРЕМЕНЕМ. РАБОТА С ФАЙЛАМИ

1.1. Краткая теория

Согласно терминологии, используемой при работе с фреймворком Qt и в стандартах Unix, под виджетом (widget) понимается любой визуальный элемент графического интерфейса пользователя (GUI). Этимология термина происходит от словосочетания window gadget, что соответствует «элементу управления» (control) и «контейнеру» (container) согласно терминологии, используемой в ОС Windows. К виджетам можно отнести кнопки, полосы прокрутки, меню или фреймы. При этом важно отметить, что одни виджеты могут содержать в себе другие виджеты. Например, «окно приложения» обычно является виджетом, содержащим «панель меню» (реализуется посредством класса QMenuBar), несколько «панелей инструментов» (реализующихся посредством класса QToolBar), «строк состояния» (реализующихся посредством класса QStatusBar) и т. п. Большинство приложений используют в качестве окна приложения классы QMainWindow или QDialog, тем не менее стоит учитывать, что во фреймворке Qt любой виджет может являться окном.

Виджеты во фреймворке Qt генерируют сигналы в ответ на выполнение пользователем какого-то действия или изменение состояния. Например, QPushButton генерирует 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.

Каждый класс, унаследованный от QObject, содержит свои собственные встроенные таймеры. Вызов метода QObject::startTimer() производит запуск таймера. В качестве параметра ему передается интервал запуска в миллисекундах. Метод startTimer() возвращает идентификатор, необходимый для распознавания таймеров, используемых в объекте. По истечении установленного интервала запуска генерируется событие QTimerEvent, которое передается в метод timerEvent(). Вызвав метод QTimerEvent::timerId() объекта события QTimerEvent, можно узнать идентификатор таймера, инициировавшего это событие. Идентификатор можно использовать для уничтожения таймера, передав его в метод QObject::killTimer().

QIODevice — это абстрактный класс, обобщающий устройство ввода/вывода, который содержит виртуальные методы для открытия и закрытия устройства ввода/вывода, а также для чтения и записи блоков данных или отдельных символов. Класс QFile унаследован от класса QIODevice. В нем содержатся методы для работы с файлами: открытия, закрытия, чтения и записи данных. Создать объект можно, передав в конструкторе строку, содержащую имя файла.

В процессе работы с файлами иногда требуется узнать, открыт файл или нет. Для этого вызывается метод QIODevice::isOpen(), который вернет значение

true, в том случае, если файл открыт, иначе — false. Чтобы закрыть файл, нужно вызвать метод close(). С закрытием произведется запись всех данных буфера. Если требуется произвести запись данных буфера в файл без его закрытия, то вызывается метод QFile::flush(). Проверить, существует ли нужный вам файл, можно статическим методом QFile::exists(). Этот метод принимает строку, содержащую полный или относительный путь к файлу. Если файл найден, то метод возвратит значение true, в противном случае — false. Для проведения этой операции существует и нестатический метод QFile::exists(). Методы QIODevice::read() и QIODevice::write() позволяют считывать и записывать файлы блоками.

Класс QBuffer унаследован от QIODevice и представляет собой эмуляцию файлов в памяти компьютера (memory mapped files). Это позволяет записывать информацию в оперативную память и использовать объекты как обычные файлы (открывать при помощи метода open() и закрывать методом close()). При помощи методов write() и read() можно считывать и записывать блоки данных.

Класс QTextStream предназначен для чтения текстовых данных. В качестве текстовых данных могут выступать не только объекты, произведенные классами, унаследованными от QIODevice, но и переменные типов char, QChar, char*, QString, QByteArray, short, int, long, float и double. Числовые данные, передаваемые в поток, автоматически преобразуются в текст. Можно управлять форматом их преобразования, например, метод QTextStream::setRealNumberPrecision() задает количество знаков после запятой. Следует использовать этот класс для считывания и записи текстовых данных, находящихся в формате Unicode. Чтобы считать текстовый файл, необходимо создать объект типа QFile и считать данные методом QTextStream:: readLine().

1.2. Цель работы

Изучить и практически освоить работу с простейшими элементами управления в среде разработки Qt, использование классов QDate, QTime, QDateTime для работы с датой и временем, а также работу с текстовыми и бинарными файлами.

1.3. Задание

- Ознакомиться по литературе [5, 9–11] с основными элементами управления, используемыми в Qt, классами и методами, позволяющими работать с датой, временем и с текстовыми и бинарными файлами.
- Создать виджет с надписью «Hello, World», задать размер виджета, вывести его на экран. В качестве виджета использовать кнопку (объект button класса QPushButton).

- Создать текстовую метку при помощи класса QLabel, в текстовой метке записать текст «Hello, Qt!». Вывести сообщение с использованием указателей и класса QLabel.
- Повторить предыдущий пункт задания без использования указателей.
- Создать кнопку при помощи класса QPushButton с текстом «Quit». При нажатии на кнопку должно происходить завершение работы программы.
- Создать объект класса QDate, содержащий текущую дату, создать объект класса QTime, содержащий текущее время, вывести полученную информацию о дате и времени в текстовую метку с использованием объектов класса QString.
- Используя события таймера, через каждые 200 мс выводить в текстовую метку сообщение «Blink».
- Используя Qt Designer, разместить на форме объект класса QLabel (текстовую метку), в которую необходимо выводить текущее время (ежесекундно).
- Используя объекты класса QFile, скопировать содержимое одного бинарного файла в другой блоками по 2 символа, при этом осуществить проверку на возможность открытия файлов для чтения и записи.
- Повторить предыдущий пункт задания с использованием объектов класса QByteArray.
- Используя объекты класса QBuffer, осуществить вывод текстовой строки в консоль.
- Используя объекты класса QTextStream, построчно считать содержимое текстового файла и вывести его в консоль.
- Повторить предыдущий пункт задания с использованием метода readAll(). Записать текстовую информацию в файл, предварительно отформатировав текст.
- Согласно пунктам выполнения лабораторной работы, сделать необходимые снимки экрана. Изучить полученную информацию и оформить ее в соответствии с требованиями раздела «Содержание отчета».

С учетом большого объема задания, рекомендуется разбить реализацию на несколько проектов. Каждый проект будет реализовывать один пункт задания.

Рассмотрим перечень файлов, входящих в создаваемый проект. Файл с расширением *.pro — файл проекта. В нем декларируются основные параметры проекта, такие как подключаемые модули, шаблон приложения, список исходных кодов программы, а также заголовочных файлов. Файлы с расширениями *.cpp и *.h — файлы основного класса приложения. В файле *.h мы можем описать объявление переменных, функций, сигналов, слотов и др., а в main.cpp описать определение, то есть реализацию. Никто не мешает совмещать объявление и реализацию в одном файле *.cpp, но подобное разделение считается хорошим стилем программирования. Файл *.cpp по умолчанию состоит из пустой реализации конструктора и деструктора.

1.4. Выполнение

Примеры программной реализации выполнения пунктов раздела «Задание» приведены в листингах П.1.1–П.1.14, которые находятся в разделе «Приложение».

Создать виджет с надписью «Hello, World», задать размер виджета, вывести его на экран, взяв за основу пример в Листинге П.1.1 ПРИЛОЖЕНИЯ. В качестве виджета использовать кнопку (объект button класса QPushButton). После запуска программы сделайте снимок экрана (рисунок 1.1).

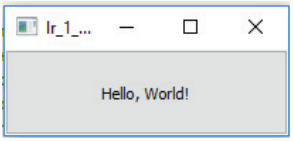

Рисунок 1.1. Главное окно приложения с кнопкой «Hello,World!»

Создать текстовую метку при помощи класса QLabel, в текстовой метке записать текст «Hello, Qt!», взяв за основу пример в Листинге П.1.2. Вывести сообщение с использованием указателей и класса QLabel. После запуска программы сделайте снимок экрана (рисунок 1.2).

Рисунок 1.2. Выполнение программы с использованием указателей

Повторить предыдущее выполнение без использования указателей, взяв за основу пример в Листинге П.1.3. После запуска программы сделайте снимок экрана (рисунок 1.3).

Рисунок 1.3. Выполнение программы без использования указателей

Создать кнопку при помощи класса QPushButton с текстом «Quit», взяв за основу пример в Листинге П.1.4. При нажатии на кнопку должно происходить завершение работы программы. Здесь связываем сигнал кнопки clicked() со слотом quit() объекта приложения QApplication. Макросы SIGNAL() и SLOT() являются частью синтаксиса. После запуска программы сделайте снимок экрана (рисунок 1.4). Нажмите на кнопку.

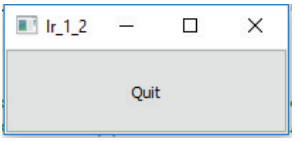

Рисунок 1.4. Главное окно приложения с кнопкой «Quit»

Создать объект класса QDate, содержащий текущую дату, создать объект класса QTime, содержащий текущее время, вывести полученную информацию о дате и времени в текстовую метку с использованием объектов класса QString, взяв за основу пример в Листинге П.1.5. После запуска программы сделайте снимок экрана (рисунок 1.5).

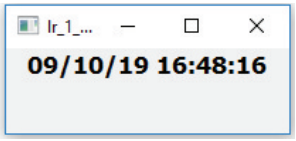

Рисунок 1.5. Вывод информации о дате и времени в текстовую метку приложения

Используя события таймера, через каждые 200 мс выводить в текстовую метку сообщение «Blink», взяв за основу пример в Листинге П.1.6. После запуска программы сделайте снимок экрана (рисунок 1.6).

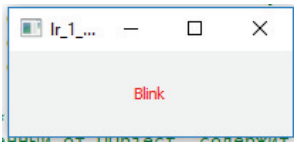

Рисунок 1.6. Вывод надписи «Blink» через каждые 200 мс

Используя Qt Designer, разместить на форме (рисунок 1.7) объект класса QLabel (текстовую метку), в которую необходимо выводить текущее время (ежесекундно). Используйте минимум файлов в своем проекте, как это показано в Листингах П.1.7–П.1.9 ПРИЛОЖЕНИЯ:
lr_1_3_3.pro — профайл;
main.cpp — основной исходный файл, с которого стартует приложение;
mainwindow.cpp — исходный код окна;
mainwindow.h — заголовочный файл основного окна приложения;
mainwindow.ui — форма основного окна приложения;

Форму создайте в QtCreator. Поместите QLabel в середину.

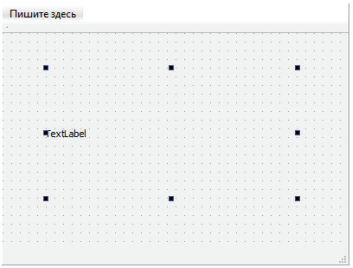

Рисунок 1.7. Форма приложения с текстовой меткой в Qt Designer

После запуска программы сделайте снимок экрана (рисунок 1.8).

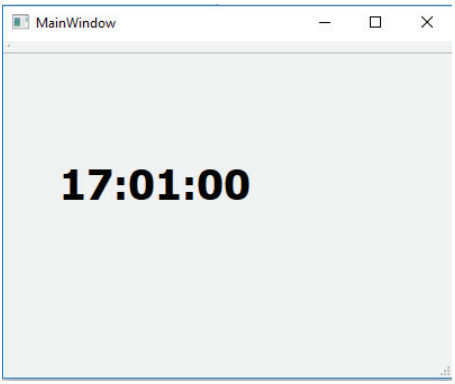

Рисунок 1.8. Вывод текущего времени в текстовую метку на форме

Используя объекты класса QFile, скопировать содержимое одного бинарного файла в другой блоками по 2 символа, при этом осуществить проверку на возможность открытия файлов для чтения и записи, возьмите за основу пример из Листинга П.1.10. Создайте в папке с проектом файл с именем file1.dat, внесите в него текстовую информацию. Аналогично создайте пустой файл с именем file2.dat. После выполнения программы проверьте содержимое файла file2.dat. Сделайте снимок экрана.

Повторить предыдущий пункт выполнения с использованием объектов класса QByteArray, взяв за основу пример в Листинге П.1.11. После выполнения программы проверьте содержимое файла file2.dat. Сделайте снимок экрана.

Используя объекты класса QBuffer, осуществить вывод текстовой строки в консоль, взяв за основу пример в Листинге П.1.12. После запуска программы сделайте снимок экрана (рисунок 1.9).

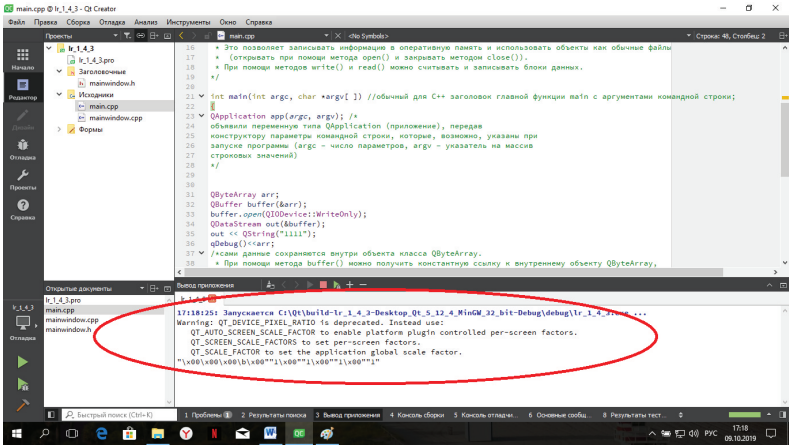

Рисунок 1.9. Вывод текстовой строки в консоль

Используя объекты класса QTextStream, построчно считать содержимое текстового файла и вывести его в консоль, взяв за основу пример в Листинге П.1.13. Создайте в папке с проектом файл с именем file.txt. Предварительно запишите в файл file.txt текстовую информацию. После запуска программы сделайте снимок экрана (рисунок 1.10).

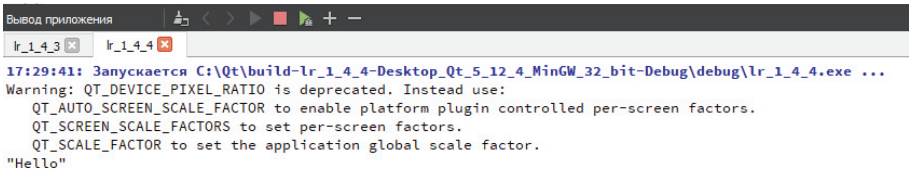

Рисунок 1.10. Вывод текстовой информации из файла file.txt в консоль

Повторить предыдущий пункт выполнения с использованием метода readAll(), взяв за основу пример в Листинге П.1.14. Записать текстовую информацию в файл, предварительно отформатировав текст. Создайте в папке с проектом файл с именем myfile.txt. Предварительно запишите в файл текстовую информацию. После выполнения программы проверьте содержимое файла file.txt и сделайте снимок экрана (рисунок 1.11).

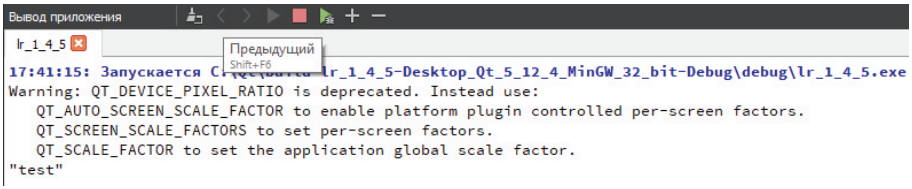

Рисунок 1.11. Вывод текстовой информации из файла "myfile.txt" в консоль

2. РАБОТА СО ЗВУКОМ

2.1. Краткая теория

Наиболее часто при создании аудиоплеера подразумевается разработка приложения, позволяющего воспроизводить аудиофайлы в формате *.mp3. Приложение должно добавлять аудиофайлы в плейлист, запускать их для воспроизведения, ставить на паузу, останавливать воспроизведение, а также осуществлять навигацию по плейлисту. Фреймворк Qt для реализации подобных задач предоставляет классы QMediaPlayer и QMediaPlaylist. Они включены в модуль multimedia, подключаемый как заголовочный файл. Классы QStandardItemModel и QTableView могут быть использованы для отображения плейлиста аудиоплеера.

Для реализации отображения плейлиста фреймворк Qt используется класс QStandardItemModel. В нем будут помещаться пути к аудиофайлам, а также имена аудиофайлов. В первой колонке будет имя аудиофайла, а во второй будет полный путь, но данная колонка будет скрыта в объекте QTableView, который будет отвечать за отображение плейлиста. Также пути к файлам в качестве медиаисточников необходимо будет поместить в объект QMediaPlaylist, который будет помещен в QMediaPlayer. Указатели на эти объекты помещаются в заголовочном файле окна приложения. Также здесь присутствует слот для обработки нажатия по кнопке добавления треков в плейлист.

Для реализации плеера необходимо инициализировать объекты классов QMediaPlayer, QMediaPlaylist и QStandardItemModel, объявленных в заголовочном файле окна приложения. В первой половине конструктора производится настройка внешнего вида таблицы для отображения плейлиста, тогда как во второй — настройка самого плеера. Управление плеером осуществляется через кнопки, которые подключены к управляющим слотам m_playlist (для навигации) и m_player (для запуска/паузы/остановки). При изменении текущего трека плеер автоматически завершает воспроизведение того трека, который был до изменения, и запускает на воспроизведение новый трек. В силу того, что класс QMediaPlaylist не имеет модели для отображения в таблице, рекомендуется использовать класс QStandardItemModel.

При разработке приложения для воспроизведения аудиофайлов следует придерживаться приведенных далее рекомендаций.

Структура проекта:

lr2.pro — профайл проекта;
main.cpp — файл с функцией «main»;
widget.cpp — файл исходных кодов окна приложения;
widget.h — заголовочный файл окна приложения;
buttons.qrc — ресурсный файл иконок кнопок приложения;
widget.ui — файл формы окна приложения.

Воспользуйтесь встроенным в Qt графическим дизайнером для создания интерфейса разрабатываемого приложения (рисунок 2.1). Элементы

графического интерфейса, используемые в разрабатываемом приложении аудиоплеера:
— текстовая метка, отображающая название воспроизводимого в настоящий момент аудиофайла, — currentTrack (QLabel*);
— кнопка для воспроизведения аудиофайла — btn_play (QToolButton*);
— кнопка для остановки аудиофайла — btn_stop (QToolButton*);
— кнопка для добавления аудиофайлов в плейлист — btn_add (QToolButton*);
— кнопка для постановки аудиофайла на паузу — btn_pause (QToolButton*);
— кнопка для пролистывания плейлиста вперед — btn_next (QToolButton*);
— кнопка для пролистывания плейлиста назад — btn_previous (QToolButton*);
— таблица, отвечающая за отображение плейлиста, — playlistView (QTableView*).

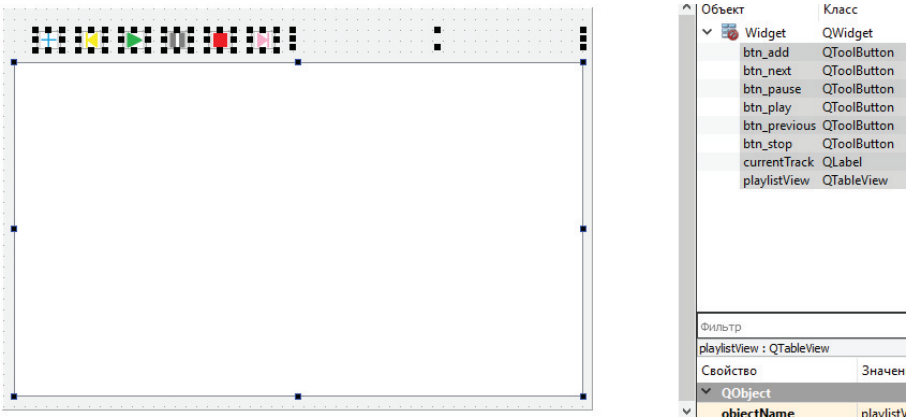

Рисунок 2.1. Форма mp3-плеера (файл widget.ui)

В профайле проекта lr2.pro не забудьте подключить модуль multimedia, иначе классы QMediaPlayer и QMediaPlaylist будут недоступны. Добавьте в проект файл ресурсов. Назовите его buttons.qrc. Добавление ресурсов изображений кнопок:

1. Дважды щелкните левой кнопкой мыши на файле buttons.qrc на боковой панели.
2. Затем нажмите левой кнопкой мыши на кнопку "Добавить" и выберите из меню "Добавить префикс".
3. Можете поменять название префикса для большего удобства.
4. Создайте при помощи графического редактора файлы, которые будут использоваться в качестве иконок кнопок. Размеры иконок должны быть 32 x 32 пикселей. Формат файлов — *.png.
5. Подключите графические изображения кнопок через ресурсный файл. Для этого переместите файлы изображений в папку с разрабатываемой программой и поместите их, например, в созданную ранее папку buttons.

2.2. Цель работы

Изучить и практически освоить основные приемы и методы работы со звуком в среде разработки Qt с использованием модуля multimedia.

2.3. Задание

- Ознакомиться по литературе [5, 9–11] с основными понятиями классов QMediaPlayer, QMediaPlaylist, QStandardItemModel и QTableView.
- Создать файлы с изображениями кнопок для работы с mp3-плеером, которые будут подключаться к проекту при помощи ресурсного файла.
- Реализовать mp3-плеер при помощи Qt Designer.
- Разработать код программы, позволяющий добавлять, воспроизводить, останавливать треки, а также ставить на паузу воспроизведение и перемещаться между треками.
- Добавить 5 треков и проверить правильность работы написанного программного обеспечения.
- Согласно пунктам выполнения лабораторной работы, сделать необходимые снимки экрана. Изучить полученную информацию и оформить ее в соответствии с требованиями раздела «Содержание отчета».

2.4. Выполнение

Примеры программной реализации выполнения пунктов раздела «Задание» приведены в листингах П.2.1–П.2.4, которые находятся в разделе «Приложение».

На рисунке 2.2 представлен запущенный на выполнение проект. Добавьте треки в список воспроизведения, запустите один из них, проверьте работоспособность всех кнопок, списков и текстовых меток. Сделайте снимок экрана во время воспроизведения одного из треков (рисунок 2.3).

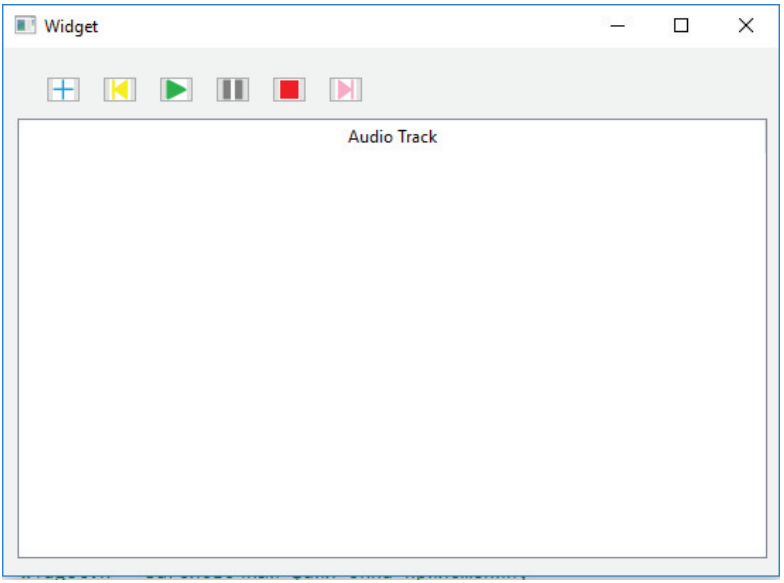

Рисунок 2.2. Разработанный mp3-плеер

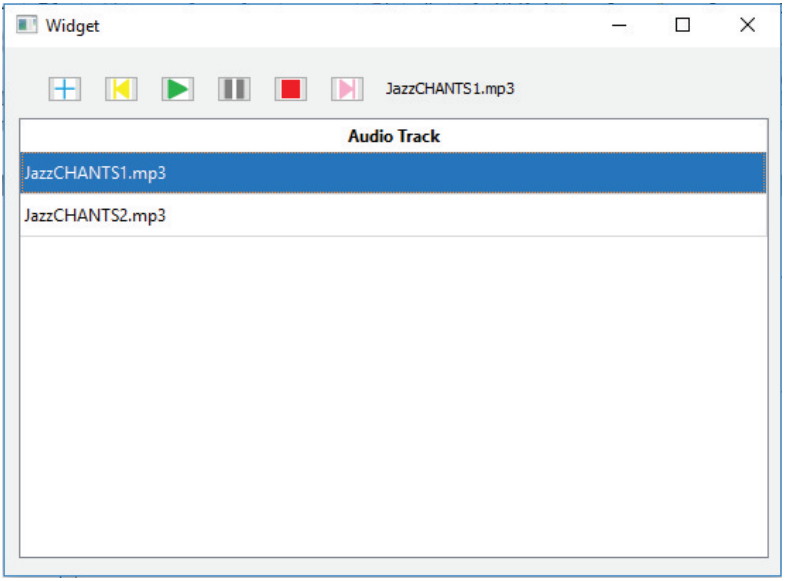

Рисунок 2.3. Воспроизведение трека

3. РАБОТА СО СТИЛЯМИ

3.1. Краткая теория

При создании кроссплатформенного программного обеспечения с использованием библиотек Qt одним из ключевых аспектов является управление видом и поведением приложения (так называемый аспект look & feel). Ввиду того, что Qt-приложения создаются для большого числа платформ, необходимо, чтобы их внешний вид не выбивался из общего стиля оформления при запуске в какой-либо операционной системе, и у пользователя не создавалось впечатление того, что программа не является родной (нативной) 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.

Хорошее приложение должно обладать возможностью изменения стилей. Во фреймворке Qt по умолчанию предопределено несколько стилей:
– Windows;
– Fusion;
– в зависимости от типа ОС: WindowsXP, WindowsVista, Gtk, Macintosh.

Работа со стилями и настройками приложения требует подключения ряда библиотек:

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

Например, далее приведен код для установки стиля Fusion:

```
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. В таблице 3.1 представлены некоторые из состояний виджетов.

Таблица 3.1. Состояния 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

Если кардинальных изменений в стиле не предполагается, то достаточно указать:

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

Аналогично устанавливаются стили для отдельных элементов:

label->setStyleSheet("background-color: yellow");

Данная лабораторная работа предполагает разработку стилей и применение их к приложению, имеющему несколько форм, полей ввода данных и текстовых меток. Корректность ввода данных в поля проверяется, и при неправильно введенном символе поле подсвечивается красным. Также для полей ввода данных используется фокусировка.

Предлагается выполнение этого задания на основе курсовой работы по настоящей дисциплине. Далее приведена методика расчета длины регенерационного участка волоконно-оптической линии связи (ВОЛС).

Перечисленные в таблице 3.2 и таблице 3.3 параметры фиксированы в расчете, и пользователь не должен иметь возможности изменять их.

Таблица 3.2. Параметр варианта, определяемый последней цифрой

N1	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Скорость передачи, Мб/с	34	565	140	2500	155	565	620	140	620	34

Таблица 3.3. Параметр варианта, определяемый предпоследней цифрой

N2	**0**	**2**	**4**	**5**	**8**	**1**	**3**	**5**	**7**	**9**
Тип оптического волокна	G.652 (SSF)					G.653 (DSF)				

Например, если две последние цифры билета (зачетной книжки) 12, то расчет длины регенерационного участка производится для скорости передачи 565 Мбит/с и волокна G.652 (SSF). Остальные необходимые данные имеются в таблицах 3.4 и 3.5.

Длина регенерационного участка волоконно-оптической линии связи ограничивается двумя явлениями: дисперсий и затуханием. Максимальная длина участка определяется наименьшим из рассчитанных параметров. Рассчитаем номинальную $L_{ном}$ длину участка регенерации по формуле:

$$L_{ном} = \frac{W - A_{рс} + A_{нсмакс} - A_{эза} - A_{эзк} - \Delta a}{\alpha_{макс} + \frac{A_{нсмакс}}{l_{стр}}}, \qquad (3.1)$$

где $A_{нсмакс}$ = 0,2 дБ — максимальная величина затухания одного неразъемного соединения; $A_{рс}$ = 1 дБ — суммарное затухание всех разъёмных соединений; $A_{эза}$ = 3 дБ — эксплуатационные запасы аппаратуры; $A_{эзк}$ = 3 дБ — эксплуатационные запасы кабеля; $l_{стр}$ = 5 км — строительная длина кабеля; Δa = 0,2 дБ — погрешность измерения затухания; $\alpha_{макс}$ — максимальное значение коэффициента затухания (дБ/км); W = 40 дБ — энергетический потенциал.

Максимальную длину участка регенерации по дисперсии можно определить из выражения:

$$L_{дисп} = \frac{2 \cdot \pi \cdot c \cdot \tau_0^2}{\lambda^2 \cdot D_x \cdot \sqrt{1 + 4 \cdot \pi^2 \cdot \Delta\nu^2 \cdot \tau_0^2}}, \qquad (3.2)$$

где c — скорость света в вакууме, $c = 3 \cdot 10^8$ м/с, τ_0 — ширина оптического импульса на выходе передатчика (зависит от скорости передачи), λ — длина волны излучения данного источника, D_x — величина хроматической дисперсии, зависящая от типа применяемого волокна, Δν — ширина спектра излучения; Δν зависит от ширины спектра излучения источника (примите значение 50 ГГц).

Таблица 3.4. Характеристики систем передачи

Системы передачи	**STM-1**	**STM-4**	**STM=16**	**STM-64**
C_T, Мбит/с	155	620	2500	10000
τ_0, нс	1.54	0.386	0.096	0.0024
τ_f, нс	1.96	0.49	0.122	0.031
ΔF, МГц	81.2	325	1300	5200

Таблица 3.5. Характеристики оптических волокон

Параметры оптических волокон, рекомендуемые МСЭ		**Рекомендации МСЭ**			
		G.652	*G.653*	*G.655*	*G.655*
Тип волокна		SSF	DSF	-NZDSF	+NZDSF
Окна прозрачности, нм		1300/1550	1500-1600	1530-1565	1530-1565
Коэффициент затухания α, дБ/км	1310 нм	0.34	<1.0	н/н	н/н
	1550 нм	0.22	0.22	0.19-0.22	0.19-0.22
Хроматическая дисперсия D_x, пс/км/нм	1310 нм	±3.5	17-18	н/н	н/н
	1550 нм	17-18	±3.5	-(5-8)	5-8

3.2. Цель работы

Изучить и практически освоить работу со стилями в среде разработки Qt с использованием класса QStyle.

3.3. Задание

- Ознакомиться по литературе [5, 9–11] с основными методами и понятиями класса QStyle.
- Разработать собственный файл стилей.
- Создать приложение, которое имеет несколько форм, полей для ввода данных, кнопок и текстовых меток, куда может выводиться информация.
- Разработать код программы, позволяющий изменять стиль оформления приложения в реальном времени.
- Согласно пунктам выполнения лабораторной работы, сделать необходимые снимки экрана. Изучить полученную информацию и оформить ее в соответствии с требованиями раздела «Содержание отчета».

3.4. Выполнение

Примеры программной реализации выполнения пунктов раздела «Задание» приведены в листингах П.3.1–П.3.12, которые находятся в разделе «Приложение».

Структура проекта:
lr3.pro — профайл проекта;
main.cpp — файл с главной функцией программы main;
rootwindow.ui — файл формы стартового (корневого) окна приложения;
firstpartwindow.ui — файл формы окна приложения первой части работы;
secondpartwindow.ui — файл формы окна приложения второй части работы;
aboutdialog.ui — файл формы справочного окна приложения;
rootwindow.h — заголовочный файл стартового окна приложения;
firstpartwindow.h — заголовочный файл окна приложения первой части работы;
secondpartwindow.h — заголовочный файл окна приложения второй части работы;
aboutdialog.h — заголовочный файл диалогового окна приложения;
calculate.h — заголовочный файл, в котором определены вычисления;
rootwindow.cpp — файл исходных кодов стартового (корневого) окна приложения;
firstpartwindow.cpp — файл исходных кодов окна приложения первой части работы;
secondpartwindow.cpp — файл исходных кодов окна приложения второй части работы;
aboutdialog.cpp — файл исходных кодов справочного окна приложения;
res.qrc — ресурсный файл логотипа для справочной формы (разместите в папке с проектом, например, создайте папку images, в нее поместите файл Unilogo.jpg.
MyStyle.qss и Ubuntu.qss — файлы стилей, создайте папку style в папке с проектом и поместите в нее файлы стилей.

Интерфейс приложения сделайте с использованием графического дизайнера (рисунки 3.1, 3.2, 3.3, 3.4). Добавьте на форму стартового окна приложения список, позволяющий выбирать стили приложения.

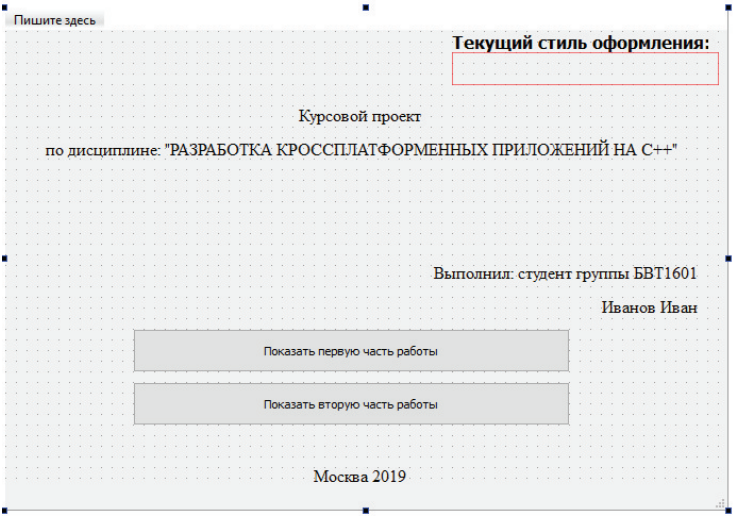

Рисунок 3.1. Форма стартового (корневого) окна приложения

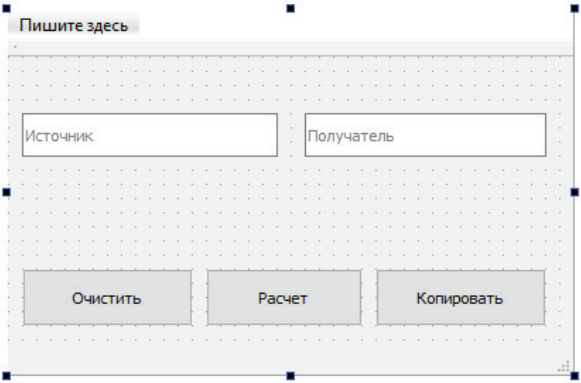

Рисунок 3.2. Форма окна приложения первой части работы

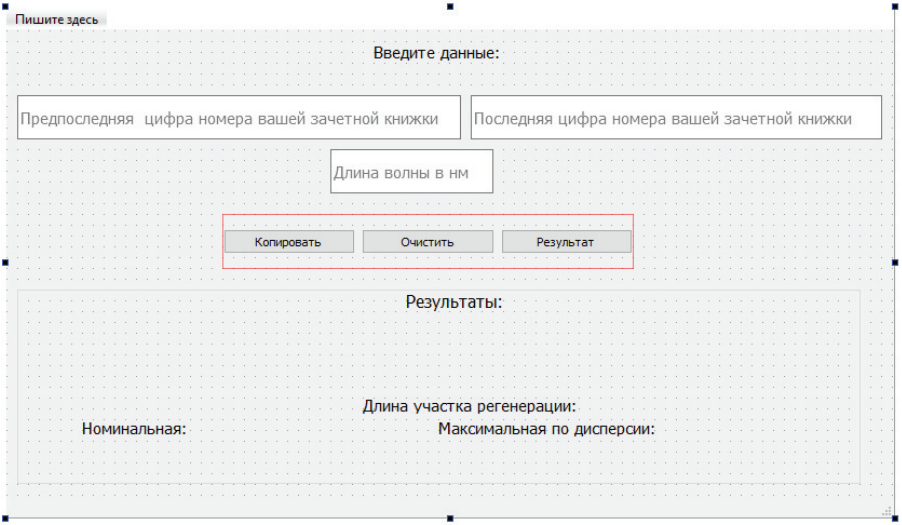

Рисунок 3.3. Форма окна приложения второй части работы

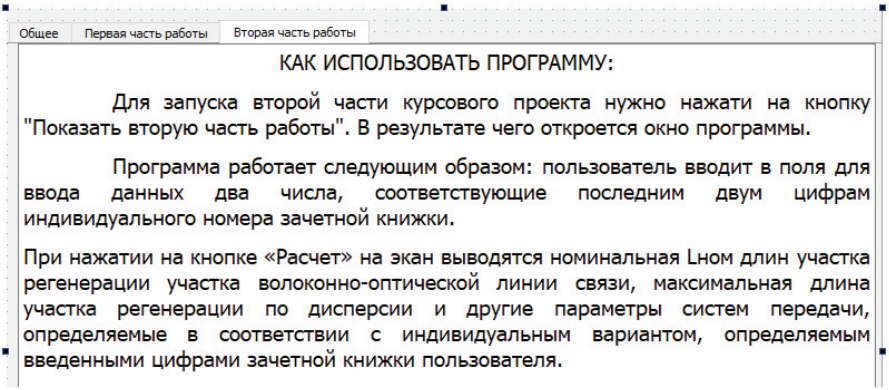

Рисунок 3.4. Форма справочного окна приложения

Для своего собственного стиля необходимо подключить файл с расширением *.qss. В Приложении приведен листинг файла для стиля Ubuntu (Листинг П.3.1). Аналогично создайте свой стиль. Назовите файл MyStyle.qss. Чтобы подключить этот стиль, достаточно написать небольшой код, взяв за основу Листинг П.3.2, где mystyle.qss – это название файла со стилями, его следует перенести в папку с исполняемым файлом. Содержание остальных файлов приведено в Листингах П.3.3–П.3.12.

Проверьте работоспособность всех кнопок, списков и текстовых меток для разных стилей. Сделайте снимки экрана для всех стилей на всех формах.

4. РАБОТА С ПОТОКАМИ

4.1. Краткая теория

Процессы и потоки в Qt

Приложение порождает один или несколько процессов. Каждый процесс может порождать один или несколько потоков. Если количество потоков в процессе стало больше одного, значит, программист использует в своей работе многопоточность. Многопоточность позволяет ускорить решения какой-либо задачи путем разбиения ее на подзадачи, каждая из которых выполняется в своем потоке. При этом важно отметить, если ПО запускается на многопроцессорной вычислительной машине, то имеется возможность производить выполнение потоков в параллельном режиме, одновременно, что значительно ускоряет процесс вычисления. Однако для достижения такого эффекта необходимо применять определенные правила синхронизации потоков, что будет описано далее.

Процессы представляют собой программы, независимые друг от друга и загруженные для исполнения. Каждый процесс должен создавать хотя бы один поток, называемый основным. Основной поток процесса создается в момент запуска программы. Однако сам процесс может создавать несколько потоков одновременно.

Многопоточность позволяет разделять задачи и независимо работать над каждой из них для того, чтобы максимально эффективно задействовать процессор. Написание многопоточных приложений требует больше времени и усложняет процесс отладки, поэтому многопоточность нужно применять тогда, когда это действительно необходимо. Многопоточность удобно использовать для того, чтобы блокировка или зависание одного из методов не стали причиной нарушения функционирования основной программы.

Для использования многопоточности нужно унаследовать класс от QThread и перезаписать метод run(), в который должен быть помещен код для исполнения в потоке. Чтобы запустить поток, нужно вызвать метод start().

Связь между объектами из разных потоков можно осуществлять при помощи сигналов и слотов или посредством обмена объектами событий.

При работе с потоками нередко требуется синхронизировать функционирование потоков. Причиной синхронизации является необходимость обеспечения доступа нескольких потоков к одним и тем же данным. Для этого библиотека Qt предоставляет классы QMutex, QWaitContion и QSemaphore.

QProcess — процессы в Qt

В том случае, когда пользователь или программа производят запуск другой программы, операционная система всегда создает новый процесс.

Процесс — это экземпляр программы, загруженной в память компьютера для выполнения.

По своей сути, процессы — это независимые друг от друга программы, обладающие своими собственными данными. Коротко процесс можно охарактеризовать как общность кода, данных и ресурсов, необходимых для его работы. Под ресурсами подразумеваются объекты, запрашиваемые и используемые процессами в период их работы. Любая прикладная программа, запущенная на компьютере, представляет собой не что иное, как процесс.

Создание процесса может оказаться полезным для использования функциональных возможностей программ, не имеющих графического интерфейса и работающих с командной строкой. Другое полезное свойство — довольно простой запуск других программ из текущей программы. Особенно он полезен для запуска команд или программ, действия которых непродолжительны по времени.

Процессы можно создавать с помощью класса QProcess, который определен в заголовочном файле QProcess. Благодаря тому, что этот класс унаследован от класса QIODevice, объекты этого класса в состоянии считывать информацию, выводимую запущенными процессами, и даже подтверждать их запросы на ввод информации. Этот класс содержит методы для манипулирования системными переменными процессами. Работа с объектами класса QProcess производится в асинхронном режиме, что позволяет сохранять работоспособность графического интерфейса программы в моменты, когда запущенные процессы находятся в работе. При появлении данных или других событий объекты класса QProcess высылают сигналы. Например, при возникновении ошибок объект процесса вышлет сигнал error() с кодом этой ошибки.

Для создания процесса его нужно запустить. Запуск процесса выполняется методом start(), в который необходимо передать имя команды и список ее аргументов, либо все вместе — команду и аргументы одной строкой. Как только процесс будет запущен, высылается сигнал started(), а после завершения его работы высылается сигнал finished(). Вместе с сигналом finished() высылается код и статус завершения работы процесса. Для получения статуса выхода можно вызвать метод exitStatis(), который возвращает только два значения: NormaiExit (нормальное завершение) и CrashExit (аварийное завершение).

Для чтения данных запущенного процесса класс QProcess предоставляет два разделенных канала: канал стандартного вывода (stdout) и канал ошибок (stderr). Эти каналы можно переключать с помощью метода setReadChannel(). Если процесс готов предоставить данные по текущему установленному каналу, то высылается сигнал readyRead(). Также высылаются сигналы для каждого канала в отдельности: readyReadStandardOutput() и readyReadStandardError().

Считывать и записывать данные в процесс можно с помощью методов класса QIODevice::write(), read(), readLine() и getChar() . Также для чтения можно воспользоваться методами, привязанными к конкретным каналам: readAllStandardOutput() и readAllStandardError(). Эти методы считывают данные в объекты класса QByteArray.

QThread — потоки в Qt

Для реализации потоков Qt предоставляет класс QThread. Поток — это независимая задача, которая выполняется внутри процесса и разделяет вместе с ним общее адресное пространство, код и глобальные данные.

Процесс, сам по себе, не является исполнительной частью программы, поэтому для исполнения программного кода он должен иметь хотя бы один поток (далее — основной поток). Можно создавать и более одного потока. Вновь созданные потоки начинают выполняться сразу же, параллельно с главным потоком, при этом их количество может изменяться — одни создаются, другие завершаются. Завершение основного потока приводит к завершению процесса, независимо от того, существуют другие потоки или нет. Создание нескольких потоков в процессе получило название «многопоточность».

Многопоточность требуется для выполнения действий в фоновом режиме, параллельно с действиями основной программы, и позволяет разбить выполнение задач на параллельные потоки, которые могут быть абсолютно независимы друг от друга. А если приложение выполняется на компьютере с несколькими процессорами, то разделение на потоки может значительно ускорить работу всей программы, так как каждый из процессоров получит отдельный поток для выполнения.

Приложения, имеющие один поток, могут выполнять только одну определенную операцию за один промежуток времени, а все остальные операции ждут ее окончания. Например, такие операции, как вывод на печать, считывание большого файла, ожидание ответа на посланный запрос или выполнение сложных математических вычислений, могут привести к блокировке или зависанию всей программы. При помощи многопоточности можно решить такую проблему, запустив подобные операции в отдельно созданных потоках. Тем самым при зависании одного из потоков функционирование основной программы не будет нарушено.

Многопоточное программирование начинается с наследования класса QThread и переопределения в нем чисто виртуального метода run(), в нем должен быть реализован код, который будет исполняться в потоке. Второй шаг заключается в создании объекта класса потока и вызове метода start(), который вызовет, в свою очередь, реализованный нами метод run().

GUI поток и рабочий поток

Каждая программа имеет один поток (thread) при запуске. Данный поток называется основным потоком или GUI потоком в Qt приложениях. Qt GUI должен запускаться в данном потоке. Все виджеты и несколько похожих классов, например QPixmap, не работают во вторичных потоках. Вторичным потоком обычно называют рабочий поток, который призван разгрузить основой поток программы.

Одновременный доступ к данным, синхронизация

Если два потока имеют указатель на некоторый объект, то возможен доступ обоих потоков к данному объекту в некоторый момент времени, и это может разрушить целостность объекта. Необходимо предотвращать одновременный доступ к объекту из разных потоков.

Основные сложности возникают тогда, когда потокам нужно совместно использовать одни и те же данные. Так как несколько потоков могут одновременно обращаться и записывать данные в одну область, то это может привести к нежелательным последствиям. Представьте себе такую ситуацию: один поток занимается вычислениями, используя значения какой-нибудь глобальной переменной, а в это время другой поток вдруг изменяет значение этой переменной, но поток, занимающийся вычислениями, продолжает свою работу, ничего не подозревая и используя уже измененное значение. Для предотвращения подобных ситуаций требуется механизм, позволяющий блокировать данные, когда один из потоков намеревается их изменить. Этот механизм получил название синхронизация.

Синхронизация позволяет задавать критические секции (critical sections), к которым в определенный момент имеет доступ только один из потоков. Это гарантирует то, что данные ресурса, контролируемые критической секцией, будут невидимы другими потоками, и они не изменят их. И только после того как поток выполнит всю необходимую работу, он освобождает ресурс, и затем доступ к этому ресурсу может получить любой другой поток. Например, если один поток записывает информацию в файл, то все другие не смогут использовать этот файл до тех пор, пока поток не освободит его.

Случаи использования многопоточности

Имеется два основных случая использования потоков:

- ускорение приложение за счет нескольких процессоров компьютера;
- сохранение отзывчивости GUI для пользователя в случае работы длительных процессов, которые могут вызвать блокировку графического интерфейса приложения.

Прежде чем создавать поток, подумайте, может быть, есть возможность решить проблему альтернативным способом (таблица 4.1).

Таблица 4.1. Аналогичные многопоточности способы решения задач

Альтернатива	Описание
QEventLoop::processEvents()	Вызов QEventLoop::processEvents() несколько раз при расчёте затрат времени позволяет предотвратить блокировку GUI. Однако, это решение не очень хорошо масштабируется, так как вызов processEvents() может происходить или слишком часто, или слишком редко в зависимости от аппаратной платформы компьютера.
QSocketNotifier QNetworkAccessManager QIODevice::readyRead()	Данное альтернативное решение имеет как один, так и несколько потоков, каждый с блокировкой на чтение из медленных соединений. До тех пор, пока выполнение операций по ответу может быть выполнено быстро, данная конструкция может быть лучше, чем использование дополнительных потоков. Данная конструкция имеет меньше ошибок, чем потоки. Во многих случаях имеется также улучшение производительности.
QTimer	Фоновые процессы могут иногда выполняться с использованием таймера для выполнения по расписанию. Например, будет выполняться с некоторой периодичностью некий код в специальном слоте объекта. Таймер со значением 0 будет срабатывать всегда, когда в процессе нет каких либо других событий потока.

Типы Qt потоков (QThread)

В таблице 4.2 представлены решения, выбираемые в зависимости от количества вызовов потоков и поставленных перед разработчиком задач.

Таблица 4.2. Типы потоков

Жизненный цикл потока	Задача разработки	Решение
Один вызов	Выполнение одного метода в другом потоке и выход из потока, когда метод завершится.	• Запись функции и запуск ее с QtConcurrent::run() • Наследование класса от QRunnable и запуск его в глобальном потоке cQThreadPool::globalInstance()->start() • Наследование класса от QThread , переопределение метода QThread::run()и использование его с помощью метода QThread::start() .
Один вызов	Длительная операция должна быть помещена в другой поток, а результат обработки должен быть отправлен в GUI поток.	Используется QThread , переопределяется метод run() и вызывается сигнал, если требуется. Подключение сигнала к слоту GUI потока с использованием очереди подключений сигналов и слотов.
Один вызов	Операции должны выполняться для всех объектов из списка. Обработка должна выполняться с использованием всех доступных ядер. Типичным примером является отрисовка эскизов изображений в списке.	QtConcurrent предоставляет метод map(), позволяющий выполнение операций для каждого элемента, filter() определяет элементы списка, над которыми будут применяться операции.
Постоянный	Имеется объект, который живёт в другом потоке и выполняет различные задачи по запросам. Это означает, что требуется некоторая связь с данным объектом и требуется рабочий поток.	Наследование класса от QObject и внедрение требуемых сигналов и слотов, передача объекта в поток с запуском event loop и связь с объектом через очередь сигналов/слотов.
Постоянный	Имеется объект, который живёт в другом потоке и выполняет повторяющиеся задачи.	Похоже на то, что может подойти таймер и также использование сигналов и слотов, но лучше избежать такого подхода. Например, может использоваться QSocketNotifier.

Одним из распространенных способов создания отдельных параллельных потоков в приложении на Qt и выполнения полезных действий в них является наследование от класса QThread и переопределение метода run(), в котором и

будет выполняться полезный код приложения. Если производится наследование от класса QThread, то логично будет предположить, что это делается с целью расширения функционала данного класса. Но это делается, чтобы вынести в метод run() некий полезный код, который должен выполняться в отдельном потоке. Может возникнуть проблема с масштабированием приложения и повторным использованием кода, особенно сильно это может проявиться в том случае, когда подобных наследованных классов становится достаточно много.

Данный метод является самым низкоуровневым и используется в первую очередь для кастомизации нативных потоков. Это несколько противоречит обычной необходимости выполнения задачи в отдельном потоке. Как было сказано выше, подобный подход в первую очередь необходим для того, чтобы расширить функционал класса. Тем не менее данный метод необходимо рассмотреть.

4.2. Цель работы

Изучить и практически освоить работу с потоками и многопоточное программирование в среде разработки Qt с использованием классов, унаследованных от класса QThread.

4.3. Задание

- Ознакомиться по литературе [5, 9–11] с основными понятиями многопоточного программирования, классом QThread, методом run(), используемым для выполнения в потоке определенного кода, а также методом start(), который запускает новый поток на исполнение.
- Разработайте консольное приложение, в файле main.cpp которого будет создано три потока с различными именами, а классы потоков станут потомками суперкласса QThread.
- Опишите класс ExampleObject, его объекты нужно переслать посредством метода moveToThread() для выполнения в разных потоках, также определите в классе ExampleObject слот-метод run() для выполнения полезной нагрузки.
- Согласно пунктам выполнения лабораторной работы, сделать необходимые снимки экрана. Изучить полученную информацию и оформить ее в соответствии с требованиями раздела «Содержание отчета».

4.4. Выполнение

Примеры программной реализации выполнения пунктов раздела «Задание» приведены в листингах П.4.1–П.4.8, которые находятся в разделе «Приложение».

Разработайте консольное приложение, в файле main.cpp которого будет создано три потока с различными именами, а классы потоков будут наследованы от QThread (примеры программного кода, решающего эту задачу, приведены в Листингах П.4.1–П.4.3). После запуска программы сделайте снимок экрана (рисунок 4.1).

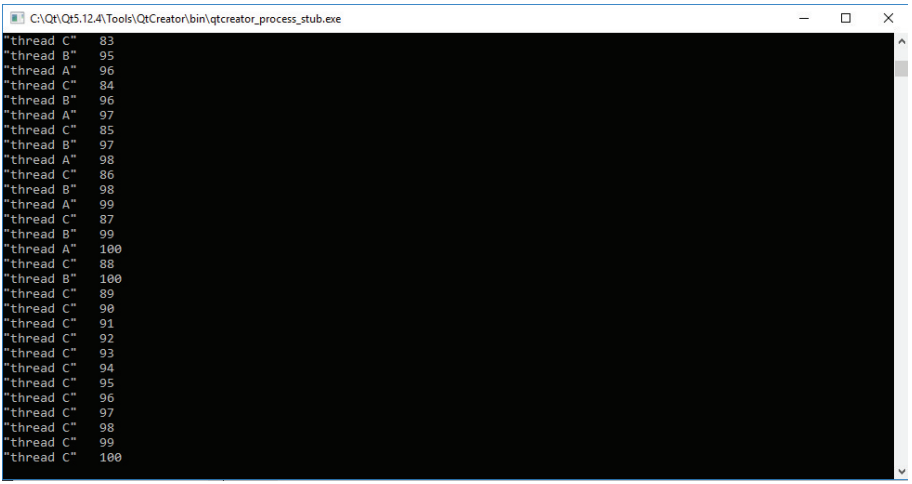

Рисунок 4.1. Результат работы программы с тремя потоками

Опишите класс ExampleObject, его объекты нужно переслать посредством метода moveToThread() для выполнения в разных потоках, также определите в классе ExampleObject слот-метод run() для выполнения полезной нагрузки. Важно отметить, что класс объектов будет наследован от QObject (примеры программного кода, решающего эту задачу, приведены в Листингах П.4.4–П.4.8).

Для того чтобы работа в методе run() могла выполняться циклично, используйте цикл типа while, управлять которым можно при помощи булевой переменной m_running. Для удобства работы с этой переменной определяем ее в качестве Q_PROPERTY. По завершении работы будет отправлен сигнал finished().

Рассмотрим непосредственно работу с потоком и объектом. Алгоритм разработки многопоточного приложения следующий:

1. Создать объект QThread и объект класса ExampleObject.
2. Подключить сигнал QThread::started() к методу ExampleObject::run().

3. Подключить сигнал ExampleObject::finished() к слоту QThread::terminate(), чтобы по завершении выполнения полезной работы завершить выполнение потока.
4. Установить переменную m_running в значение true, чтобы разрешить работу цикла, иначе метод сразу завершится.
5. Запустить поток с помощью метода start().
6. Когда нужно завершить выполнение полезной работы объекта, установить переменную m_running в значение false. Выполнение метода run() и потока, в котором живет объект, завершатся автоматически и корректно.

Структура проекта и внешний вид приложения:

lr_4_2.pro — профайл проекта;
main.cpp — файл исходных кодов с функцией main.
exampleobject.cpp — файл исходных кодов объекта, который будет передаваться в поток;
mainwindow.cpp — файл исходных кодов главного окна приложения;
exampleobject.h — заголовочный файл объекта, который будет передаваться в поток;
mainwindow.h — заголовочный файл главного окна приложения;
mainwindow.ui — файл формы главного окна приложения.

В приложении будет определено два объекта и два потока. С помощью интерфейса приложения будем задавать некоторую отображаемую в выводе qDebug() информацию, а также будем запускать и останавливать работу потоков. Интерфейс приложения сделайте с использованием графического дизайнера (рисунок 4.2).

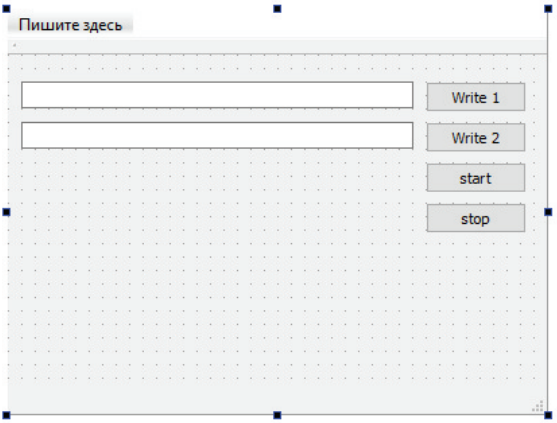

Рисунок 4.2. Форма приложения, позволяющая давать имена потокам, запускать и останавливать их

После запуска программы сделайте снимок экрана (рисунок 4.3).

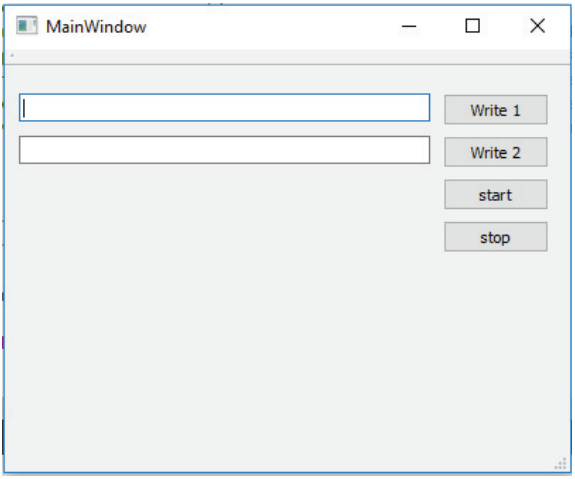

Рисунок 4.3. Главное окно программы

Запустите программу на выполнение, нажав на кнопку start. Какую информацию можно получить из выведенных данных в консоль? Сделайте снимок экрана. Дайте имена потокам (Thread1 и Thread2) при помощи полей для ввода текста и нажатия клавиш Write1 и Write2, соответственно, для первого и второго потоков.

В результате работы приложения получим следующий вывод qDebug(), который демонстрирует параллельную работу объектов (данные из первого объекта передаются во второй) в потоках, а также передачу информации из одного потока в другой (рисунок 4.4). Остановите работу программы, нажав на кнопку stop. Сделайте снимок экрана.

```
"Thread1"   ""   119606
"Thread2"   "Thread1"   119138
"Thread1"   ""   119607
"Thread2"   "Thread1"   119139
"Thread1"   ""   119608
"Thread2"   "Thread1"   119140
"Thread1"   ""   119609
"Thread2"   "Thread1"   119141
"Thread1"   ""   119610
```

Рисунок 4.4. Вывод информации в консоль, демонстрирующий параллельную работу объектов (данные из первого объекта передаются во второй) в потоках

5. РАБОТА С СОКЕТАМИ

5.1. Краткая теория

Программирование поддержки сети в Qt

Фреймворк Qt позволяет использовать модуль QtNetwork для работы с сетевыми соединениями, что облегчает разработку кроссплатформенных сетевых приложений. В этом модуле имеются классы QTcpServer, QUdpSocket и QAbstractSocket для низкоуровневой работы с сетью, а также классы для высокоуровневой работы с сетью, такие как QFtp или QHttp.

Сокетное соединение

Сокет — это устройство пересылки данных с одного конца связи на другой. Другой конец может принадлежать процессу, работающему на локальном компьютере, а может располагаться и на удаленном компьютере, подключенном к Интернету и расположенном в другом полушарии Земли. Сокетное соединение — это соединение типа точка-точка (point to point), которое производится между двумя процессами.

Сокеты разделяют на дейтаграммные (datagram) и поточные. Дейтаграммные сокеты осуществляют обмен пакетами данных. Поточные сокеты устанавливают связь и производят потоковый обмен данными через установленную ими связь. На практике поточные сокеты используются гораздо чаще, чем дейтаграммные, из-за того, что они предоставляют дополнительные механизмы, направленные против искажения и потери данных. Поточные сокеты работают в обоих направлениях, то есть то, что один из процессов записывает в поток, может быть считано процессом на другом конце связи, и наоборот.

Для дейтаграммных сокетов Qt предоставляет класс QUdpSocket, а для поточных — класс QTcpSocket.

Класс QTcpSocket содержит набор методов для работы с TCP (Transmition Control Protocol, протокол управления передачей данных) — это сетевой протокол низкого уровня, являющийся одним из основных протоколов в Интернете. С его помощью можно реализовать поддержку для стандартных сетевых протоколов, таких как: HTTP, FTP, POP3, SMTP, — и даже для своих собственных протоколов. Этот класс унаследован от класса QAbstractSocket, который, в свою очередь, наследует класс QIODevice. А это значит, что для доступа (чтения и записи) к его объектам необходимо применять все методы класса QIODevice и использовать классы потоков QDataStream или QTextStream. Работа класса QTcpSocket асинхронна, что дает возможность избежать блокирования приложения в процессе его работы. Но если вам это не нужно, то можете воспользоваться серией методов, начинающихся со слова waitFor. Вызов этих методов приведет к ожиданию выполнения операции и

заблокирует, на определенное время, исполнение вашей программы. Не рекомендуется вызывать эти методы в потоке графического интерфейса.

Модель «клиент-сервер»

Сценарий модели «клиент-сервер» выглядит очень просто: сервер предлагает услуги, а клиент ими пользуется. Программа, использующая сокеты, может выполнять либо роль сервера, либо роль клиента.

Для того чтобы клиент мог взаимодействовать с сервером, ему нужно знать его IP-адрес и номер порта, через который клиент, желающий воспользоваться этими услугами сервера, должен сообщить о себе. Когда клиент устанавливает соединение с сервером, система назначает данному соединению отдельный сокет. После этого устанавливается связь между двумя этими сокетами, по которой высылаются данные запроса к серверу. А сервер высылает клиенту, по этому соединению, готовые, обработанные результаты согласно его запросам. Сервер не ограничен связью только с одним клиентом, на самом деле он может обслуживать многих клиентов.

Каждому сокету соответствует уникальный номер порта. Некоторые номера зарезервированы для так называемых стандартных служб.

Реализация сервера с помощью класса QTcpServer

Для реализации сервера Qt предоставляет удобный класс QTcpServer, который предназначен для управления входящими TCP-соединениями. При этом сервер представляется как объект. Для запуска сервера требуется создать объект класса с передачей конструктору в качестве аргумента номера порта, по которому предоставляется указанный сервис.

Для установки сервера необходимо вызвать в конструкторе метод listen(). В этот метод необходимо передать номер порта, который получают в конструкторе. При возникновении ошибочных ситуаций, например невозможности захвата порта, этот метод возвратит значение false.

Реализация клиента с помощью класса QTcpSocket

Для реализации клиента нужно создать объект класса QTcpSocket, а затем вызвать метод connectToHost(), передав в него первым параметром имя компьютера (или его IP-адрес), а вторым — номер порта сервера. Объект класса QTcpSocket сам попытается произвести установку связи с сервером и, в случае успеха, вышлет сигнал connected(). В противном случае будет выслан сигнал error(int) с кодом ошибки, определенным в перечислении QAbstractSocket::SocketError. Это может произойти, например, в том случае, если на указанном компьютере не установлен сервер или не соответствует номер порта. После установления соединения объект класса QTcpSocket может высылать или считывать данные сервера.

Из объекта сокета вызывается метод connectToHost(), осуществляющий связь с сервером. Первым параметром в этот метод передается имя компьютера, а вторым — номер порта. Связь между сокетами асинхронна. Сокет высылает сигнал connected(), как только будет произведено соединение, а также высылает сигнал readyRead() о готовности предоставить данные для чтения.

5.2. Цель работы

Изучить и практически освоить основные приемы и методы работы с сокетами в среде разработки Qt с использованием модуля network и классов QTcpSocket и QTcpServer.

5.3. Задание

- Ознакомиться по литературе [5, 6, 9–11] с основными понятиями классов QTcpSocket и QTcpServer для разработки клиент-серверных приложений.
- Разработать код программы, реализующий серверную часть приложения.
- Разработать код программы, реализующий клиентскую часть приложения.
- Запустить сначала сервер, затем клиентское приложение.
- Переслать от клиента текстовые сообщения на сервер и получить в ответ подтверждающие сообщения.
- Завершить работу сначала сервера, а затем клиентского приложения.
- Согласно пунктам выполнения лабораторной работы, сделать необходимые снимки экрана. Изучить полученную информацию и оформить ее в соответствии с требованиями раздела «Содержание отчета».

5.4. Выполнение

Примеры программной реализации выполнения пунктов раздела «Задание» приведены в листингах П.5.1–П.5.8, которые находятся в разделе «Приложение».

В профайле проектов не забудьте подключить модуль network, иначе классы QTcpSocket и QTcpServer будут недоступны. Сначала создайте приложения сервера, затем приложение клиента. Структура проекта:

1. серверная часть приложения (Листинги П.5.1–П.5.4):

lr5.pro — профайл проекта серверной части приложения;
main.cpp — файл с функцией main серверной части приложения;
myserver.h — заголовочный файл окна приложения серверной части;

myserver.cpp — файл исходных кодов окна приложения серверной части.

2. клиентская часть приложения (Листинги П.5.5–П.5.8):

lr5_1.pro — профайл проекта клиентской части приложения;
main.cpp — файл с функцией main клиентской части приложения;
myclient.h — заголовочный файл окна приложения клиентской части;
myclient.cpp — файл исходных кодов окна приложения клиентской части.

После запуска серверной части приложения сделайте снимок экрана (рисунок 5.1).

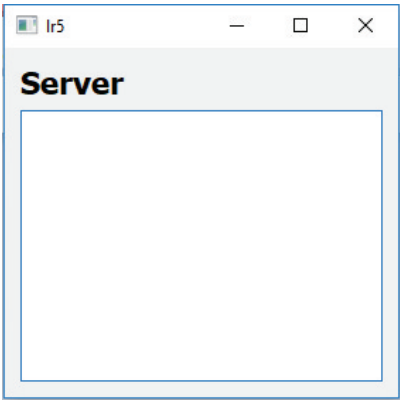

Рисунок 5.1. Окно серверной части приложения

После запуска клиентской части приложения сделайте снимок экрана (рисунок 5.2).

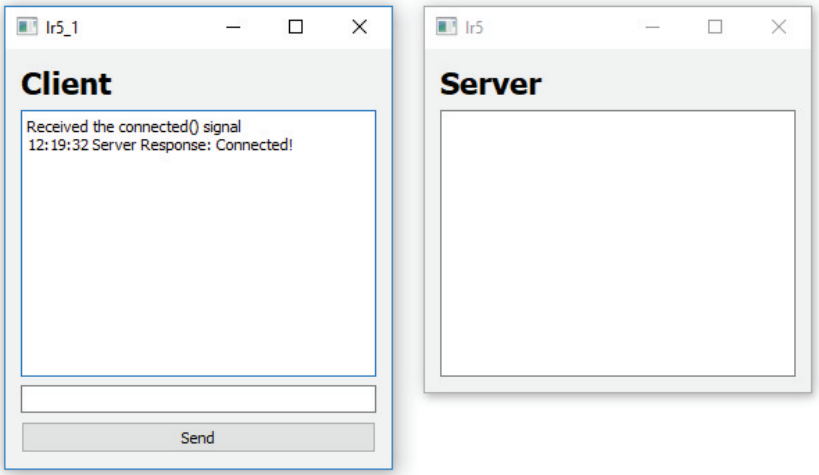

Рисунок 5.2. Окна серверной и клиентской частей приложения

Проверьте, был ли успешно подключен клиент к серверу? Перешлите несколько текстовых сообщений от клиента на сервер. Что получает клиент в ответ от сервера? Сделайте снимок экрана (рисунок 5.3).

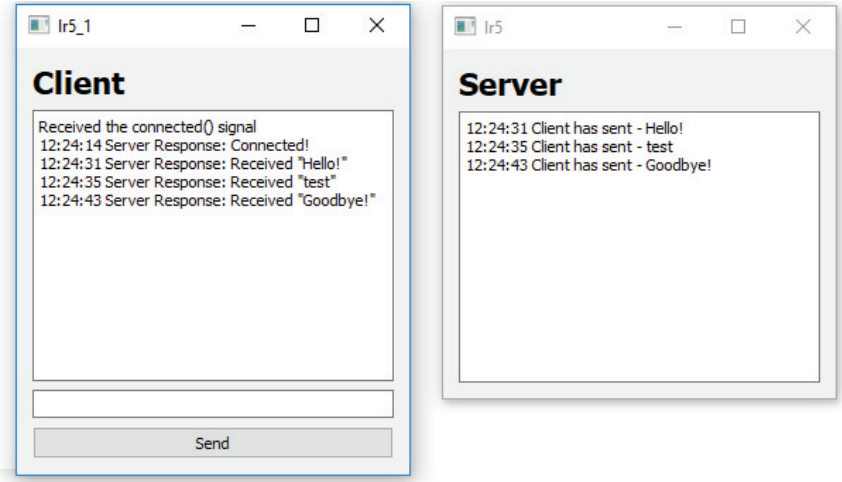

Рисунок 5.3. Успешная передача сообщений от клиента к серверу

Отключите сервер. Сделайте снимок экрана.

6. РЕАЛИЗАЦИЯ HTTP-ЗАПРОСОВ

6.1. Краткая теория

Класс QFtp

Фреймворк Qt для упрощения работы с сайтами и удаленными хранилищами данных предоставляет специализированные классы QFtp и QHttp, базирующиеся на классе QObject. Такие классы работают в асинхронном режиме, что позволяет не блокировать приложение при получении или отправке данных или сообщений.

Для получения информации о процессе выполнения команд указанные классы имеют сигналы, которые можно соединить, например, со слотом метода setProgress(), являющимся виджетом индикатора для визуального отображения прогресса. Так, в конце выполнения различных операций высылается сигнал done().

Передача файлов является часто выполняемой операцией практически во всех сетях. FTP (File Transfer Protocol, протокол передачи файлов) — наиболее известный из старых протоколов и являющийся одной из первых сетевых служб. Целью создания этого протокола было предоставление пользователям доступа к файлам на удаленном компьютере.

Кроме предоставляемого сервиса, протокол имеет целый ряд команд, с помощью которых можно управлять удаленным компьютером для передачи данных. Например:

— для создания каталога на удаленном сервере используйте команду mkdir;
— для копирования файла на удаленный сервер используйте команду put;
— для открытия каталога на удаленном сервере используйте команду cd;
— для копирования файла с удаленного сервера используйте команду get;
— для переименования файла или каталога на удаленном сервере используйте команду rename;
— для удаления каталога на удаленном сервере используйте команду rmdir;
— для закрытия соединения с удаленным сервером используйте команду close.

Класс QFtp реализует сторону клиента FTP-протокола и содержит в себе методы для наиболее часто используемых операций с FTP. Названия этих методов соответствует названиям FTP-команд, например: get() соответствует команде get, а put () — команде put, и т. д. Также класс QFtp предоставляет возможность исполнения любых FTP-команд. Для этого в метод rawCommand() нужно передать строку, содержащую нужную команду. Например:

ftp.rawCommand("MKDIR MyDir");

Каждый из методов возвращает идентификационный номер, который используется в сигналах класса QFtp. Этим можно воспользоваться для оповещения пользователя о проводимых операциях. Например, в начале исполнения одной из команд объект класса QFtp высылает сигнал commandStarted(int), а по завершении команды высылается сигнал commandFinished(int, bool). При этом идентификационный номер проводимой операции содержится в параметре типа int.

Класс QFtp содержит методы для осуществления соединения с FTP-сервером: connectToHost(), login().

Класс QHttp

HTTP (HyperText Transfer Protocol, протокол передачи гипертекста) является стандартным и самым известным протоколом для обмена данными в сетях. Его использование проще, чем использование рассмотренного FTP-протокола. В нем используется только одно соединение, в то время как FTP применяет два: одно для отсылки команд, другое — для перекачивания данных.

Qt предоставляет класс QHttp для реализации стороны клиента HTTP-протокола. Использование этого класса очень похоже на использование класса QFtp. Для работы с сетью, кроме использования классов QTcpSocket или QUdpSocket, можно использовать QNetworkAccessManager. Данный класс предоставляет функционал для отправки запросов по сети и получения ответов и удобен для работы с протоколом HTTP.

6.2. Цель работы

Изучить и практически освоить основные приемы и методы работы по реализации http-запросов в среде разработки Qt с использованием модуля network и классов QNetworkAccessManager, QNetworkRequest, QNetworkReply и QUrl.

6.3. Задание

- Ознакомиться по литературе [5, 6, 9–11] с основными понятиями классов QNetworkAccessManager, QNetworkRequest, QNetworkReply и QUrl для разработки приложения, осуществляющего http-запросы к сайту.
- Разработать приложение, которое скачивает с помощью http-запросов файл с сайта, затем сохраняет его на локальный диск и считывает содержимое файла в QTextEdit для отображения.
- Согласно пунктам выполнения лабораторной работы, сделать необходимые снимки экрана. Изучить полученную информацию и оформить ее в соответствии с требованиями раздела «Содержание отчета».

Необходимо написать приложение, которое позволит скачать xml-файл с сайта и записать его файл на локальном диске компьютера.

Последовательность выполняемых приложением операций:

1. Скачать файл;
2. Записать его на локальный диск по следующему пути "C:/Qt/lr6/file.xml";
3. Прочитать записанный файл и отобразить данные с помощью QTextEdit.

Структура проекта следующая:

lr6.pro — профайл проекта;
main.cpp — основной файл исходных кодов приложения;
widget.cpp — файл исходных кодов окна приложения;
downloader.cpp — файл исходных кодов класса для скачивания файла;
widget.h — заголовочный файл окна приложения;
downloader.h — заголовочный файл класса для скачивания файла;
widget.ui — файл формы окна приложения.

6.4. Выполнение

Примеры программной реализации выполнения пунктов раздела «Задание» приведены в листингах П.6.1–П.6.5, которые находятся в разделе «Приложение».

Файл main.cpp создается по умолчанию и не модифицируется, тогда как в файле lr6.pro необходимо подключить модуль network. В окне приложения располагается кнопка, по нажатии на которую будет производиться запуск скачивания данных с сайта и QTextEdit, в который будут помещены данные из сохраненного файла (рисунок 6.1).

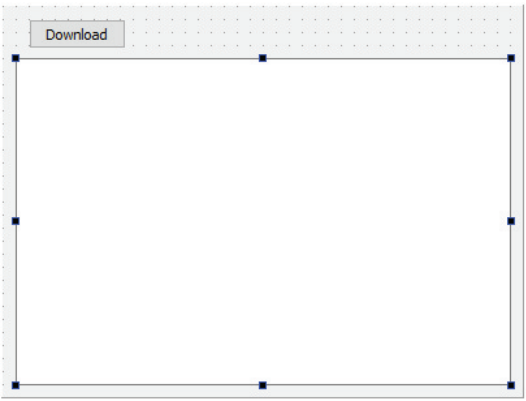

Рисунок 6.1. Форма окна приложения

После запуска программы нажмите на кнопку Download, сравните содержание файла file.xml и вывода программы и сделайте снимок экрана.

7. РАБОТА С БАЗАМИ ДАННЫХ

7.1. Краткая теория

Программирование баз данных с помощью Qt

База данных (БД) представляет собой систему хранения записей, организованных в виде таблиц. Типичная база данных содержит от одной до нескольких сотен таблиц, которые могут быть связаны между собой. Таблица состоит из набора строк и столбцов. Столбцы таблицы имеют имена и за каждым из них закреплен тип и/или область значения. Строки таблицы баз данных называются записями, а ячейки, на которые делится запись, полями. Первичный ключ — уникальный идентификатор, который может представлять собой не только один столбец, но и целую комбинацию столбцов. Пользователь может выполнять множество разных операций с таблицами, например: 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.

Основными действиями, производимыми с базой данных, являются создание новой таблицы, чтение (выборка и проекция), вставка, изменение и удаление данных. Важно отметить, что язык SQL нечувствителен к регистру, например, SELECT = select = Select и т. д. Далее для выделения ключевых слов, запросов и команд SQL будет использоваться верхний регистр. Язык запросов SQL — это стандарт, который используется в большом количестве систем управления базами данных (СУБД), что обеспечивает его кроссплатформенность. Для работы с базой данных нужно сначала активизировать драйвер, а затем установить связь с базой данных. После этого посредством запросов SQL можно получать, вставлять, изменять и удалять данные. Фреймворк Qt предоставляет модуль поддержки баз данных, классы которого разделены на три уровня: уровень драйверов, программный и пользовательский. Прежде чем начать работу с базой данных, необходимо соединиться с ней, активизировав драйвер. Запросы, реализуемые посредством языка SQL, оформляются в виде строки. Для высылки запросов во фреймворке Qt используется класс QSqlQuery. При помощи концепции «Интервью» можно легко отображать данные SQL-моделей в представлениях.

Создание таблицы

Для создания таблицы используется оператор CREATE TABLE, в котором указываются имена столбцов таблицы, их тип, а также задается первичный ключ:

```
CREATE TABLE addressbook (
number INTEGER PRIMARY KEY NOT NULL,
name VARCHAR(15),
phone VARCHAR(12),
email VARCHAR(15)
);
```

Операция вставки

После создания таблицы можно добавлять данные. Для этого SQL предоставляет оператор вставки INSERT INTO. Сразу после названия таблицы нужно указать в скобках имена столбцов, в которые будут заноситься данные. Сами данные указываются после ключевого слова VALUES.

Чтение данных

Составной оператор SELECT ... FROM ... WHERE осуществляет операции выборки и проекции. Операция выборки соответствует выбору строк, а операция проекции — выбору столбцов. Этот оператор возвращает таблицу, созданную согласно заданным критериям.

Ключевое слово SELECT является оператором для проведения проекции, то есть в нем указываются столбцы, которые должны стать ответом на запрос. Если указать после SELECT знак *, то результирующая таблица будет содержать все столбцы таблицы, к которой был адресован запрос. Указание конкретных имен столбцов устраняет в ответе все остальные. Ключевое слово FROM задает таблицу, к которой адресован запрос. Ключевое слово WHERE является оператором выборки. Выборка осуществляется согласно условиям, указанным сразу после оператора.

Изменение данных

Для изменения данных таблицы используется составной оператор UPDATE ... SET. После названия таблицы в операторе SET указывается название столбца (или столбцов, через запятую), в который будет заноситься нужное значение. Изменение данных производится в строках, удовлетворяющих условию, поставленному в ключевом слове WHERE.

Удаление

Удаление строк из таблицы производится при помощи оператора DELETE ... FROM. После ключевого слова WHERE следует критерий, согласно которому производится удаление строк.

Использование SQL в библиотеке Qt

Для использования баз данных Qt предоставляет отдельный модуль QtSql. Для его использования необходимо в проектном файле впечатать следующую строку:

QT += sql

Для использования в работе классов этого модуля необходимо включить заголовочный метафайл QtSql:

#include <QtSql>

Классы этого модуля разделяются на три уровня:

- уровень драйверов,
- программный уровень,
- уровень пользовательского интерфейса.

К первому уровню относятся классы для получения данных на физическом уровне. Это такие классы, как: QSqlDriver, QSqlDriverCreator<T*>, QSqlDriverCreatorBase, QSqlDriverPlugin и QSqlResult.

Классы второго уровня предоставляют программный интерфейс для обращения к базе данных. К классам этого уровня относятся следующие классы: QSqlDatabase, QSqlQuery, QSqlError, QSqlField, QSqlIndex и QSqlRecor.

Третий уровень предоставляет модели для отображения результатов запросов в представлениях интервью. К этим классам относятся: QSqlQueryModel, QSqlTableModel и QSqlRelationalTableModel.

Классы первого уровня не используются, если нет необходимости писать свой собственный драйвер для менеджера базы данных. В большинстве случаев все ограничивается использованием конкретной СУБД (система управления базами данных), поддерживаемой Qt.

Соединение с базой данных с помощью Qt

Для соединения с базой данных прежде всего нужно активизировать драйвер. Для этого вызывается статический метод QSqlDatabase::addDatabase(). В него нужно передать строку, обозначающую идентификатор драйвера СУБД. Для того чтобы подключиться к базе данных, потребуется четыре следующих параметра:

- имя базы данных — передается в метод QSqlDatabase:: setDatabaseName();
- имя пользователя, желающего к ней подключиться, — передается в метод QSqlDatabase::setUserName();
- имя компьютера, на котором размещена база данных, — передается в метод QSqlDatabase::setHostName();
- пароль — передается в метод QSqlDatabase::setPassword().

Методы должны вызываться из объекта, созданного с помощью статического метода QSqlDatabase::addDatabase(). Само соединение осуществляется методом QSqlDatabase::open(). Значение, возвращаемое им, рекомендуется проверять. В случае возникновения ошибки информацию о ней можно получить с помощью метода QSqlDatabase::lastError(), который возвращает объект класса QSqlError. Его содержимое можно вывести на экран с помощью qDebug(). Если имеется необходимость получить строку с ошибкой, то нужно вызвать из объекта класса QSqlError метод text().

Исполнение команд SQL в Qt

Для исполнения команд SQL, после установления соединения, можно использовать класс QSqlQuery. Запросы (команды) оформляются в виде обычной строки, которая передается в конструктор или в метод QSqlQuery::exec(). В случае конструктора запуск команды будет производиться автоматически, при создании объекта.

Класс QSqlQuery предоставляет возможность навигации. Например, после выполнения запроса SELECT можно перемещаться по собранным данным при помощи методов next(), previous(), first(), last() и seek(). С помощью метода next() происходит перемещение на следующую строку данных, а вызов метода previous() перемещает на предыдущую строку данных. При помощи методов first() и last() можно установить первую и последнюю строку данных соответственно. Метод seek() устанавливает строку данных по указанному целочисленному индексу в его параметре. Количество строк данных можно получить вызовом метода size().

Дополнительные сложности возникают с запросом-вставкой INSERT. Дело в том, что в запрос нужно внедрять данные. Для достижения этого можно воспользоваться двумя методами: prepare() и bindValue(). В методе prepare() задается шаблон, данные в который подставляются методами bindValue(). Также можно воспользоваться известным из ODBC вариантом использования безымянных параметров. В качестве третьего варианта можно воспользоваться классом QString, в частности методом QString::arg(), с помощью которого можно произвести подстановку значений данных.

В случае удачного соединения с базой данных с помощью метода createConnection() создается строка, содержащая команду SQL для создания таблицы. Эта строка передается в метод exec() объекта класса QSqlQuery. Если создать таблицу не удается, то на консоль будет выведено предупреждающее сообщение. Для получения результата запроса следует вызвать метод QSqlQuery::value(), в котором необходимо передать номер столбца. Для этого используют метод record(). Этот метод возвращает объект класса QSqlRecord, который содержит информацию, относящуюся к запросу SELECT. С его помощью, вызовом метода QSqlRecord::indexOf(), получаем индекс столбца.

Метод value() возвращает значения типа QVariant. QVariant — специальный класс, объекты которого могут содержать в себе значения разных типов. Поэтому, если необходимо, полученное значение нужно преобразовать

к требуемому типу, воспользовавшись методами QVariant::toInt() и QVariant::toString().

Классы SQL-моделей для «Интервью»

Модуль QtSql поддерживает концепцию «Интервью», предоставляя целый ряд моделей для использования их в представлениях. Класс QSqlTableModel позволяет, например, отображать данные в табличной и иерархической форме.

Использование «Интервью» — самый простой способ отобразить данные таблицы. Здесь не потребуется цикла для прохождения по строкам таблицы. После соединения с базой данных, проводимого с помощью функции createConnection(), создается объект табличного представления класса QTableView и объект табличной модели класса QSqlTableModel. Вызовом метода setTable() устанавливается актуальная база в модели. Вызов метода select() производит заполнение данными.

Рассмотрим возможности редактирования и записи данных. Класс QSqlTableModel предоставляет для этого три следующие стратегии редактирования, которые устанавливаются с помощью метода setEditStrategy():

- onRowChange — производит запись данных, как только пользователь перейдет к другой строке таблицы;
- onFieldChange — производит запись данных после того, как пользователь перейдет к другой ячейке таблицы;
- OnManualSubmit — записывает данные по вызову слота submitAl(), если вызывается слот revertAll(), то данные возвращаются в исходное состояние.

Вызовом метода setEditStrategy(), применяется стратегия QSqlTableModel::OnFieldChange. Теперь данные модели можно изменять после двойного щелчка на ячейке. Вызовом метода setModel() устанавливают модель в представлении.

Если необходимо произвести отображение данных какого-либо конкретного запроса SELECT, то для этого целесообразно будет воспользоваться другим классом SQL-моделей — классом QSqlQueryModel.

Таким образом, чтобы представить информацию, содержащуюся в таблице базы данных, во фреймворке Qt используется несколько классов:

- QSqlQueryModel — формирует таблицу путем задания «сырого» SQL-запроса, такая модель полезна при формировании особо сложных фильтров и компиляции информации из различных таблиц базы данных;
- QSqlTableModel — формирует таблицу по имени той таблицы, которая существует в базе данных, к недостаткам такой модели можно отнести отсутствие методов подключения связей с другими таблицами, чтобы подставлять значения в поля из других таблиц по ID;
- QSqlRelationalTableModel — формирует таблицу со связями из других таблиц, подменяя значения таблицы, которую представляет данная модель, по ID записей, содержащихся в других таблицах.

Для отображения данных в виджет с произвольной формой используется класс QDataMapperWidget . Для работы с этим виджетом требуется модель, для представления данных, например, с помощью классов QSqlTableModel или QSqlRelationalTableModel , но данные подставляются уже не в QTableView , а в различные произвольные объекты, например, QLineEdit, QComboBox или в диалоговое окно для добавления записей.

7.2. Цель работы

Изучить и практически освоить работу с базами данных в среде разработки Qt с использованием классов QSql, QSqlTableModel, QSqlDatabase и QDataWidgetMapper.

7.3. Задание

- Ознакомиться по литературе [5, 9–11] с основными понятиями классов QSql, QSqlTableModel, QSqlDatabase и QDataWidgetMapper.
- Разработать приложение, которое при помощи графического интерфейса пользователя позволяет создавать базу данных (БД), добавлять в нее информацию, редактировать в ней данные. База данных должна состоять из одной таблицы, содержащей информацию о компьютерах. К этой информации относятся данные об имени компьютера (поле данных «Имя компьютера»), о сетевом адресе компьютера (поле данных «IP-адрес»), а также о физическом адресе компьютера (поле данных «MAC-адрес»).
- Необходимо также реализовать возможность добавления в таблицу данных о новом компьютере посредством диалогового окна, вызываемого через отдельную кнопку на основной форме приложения.
- Добавить в таблицу данные о нескольких компьютерах (например, о семи компьютерах), изменить информацию о 2–3 компьютерах.
- Согласно пунктам выполнения лабораторной работы, сделать необходимые снимки экрана. Изучить полученную информацию и оформить ее в соответствии с требованиями раздела «Содержание отчета».

7.4. Выполнение

Примеры программной реализации выполнения пунктов раздела «Задание» приведены в листингах П.7.1–П.7.8, которые находятся в разделе «Приложение».

При создании проекта выберите тип «Приложение Qt Widgets». В проекте используются следующие файлы:
lr7.pro – файл проекта;
main.cpp — основной исходный файл, с которого стартует приложение;
mainwindow.cpp — исходный код основного окна приложения;
database.cpp — исходный код вспомогательного класса, применяющегося для работы с информацией, которая помещена в базу данных;
dialogadddevice.cpp — исходный код диалогового окна для добавления и редактирования записей;
mainwindow.h — заголовочный файл основного окна приложения;
database.h — заголовочный файл вспомогательного класса, применяющегося для работы с информацией, которая помещена в базу данных;
dialogadddevice.h — заголовочный файл диалогового окна для добавления и редактирования записей;
mainwindow.ui — форма основного окна приложения;
dialogadddevice.ui — форма диалогового окна для добавления и редактирования записей.

Интерфейс приложения сделайте с использованием графического дизайнера (рисунок 7.1 и рисунок 7.2). В состав интерфейса приложения входят следующие элементы:

1) форма главного окна будет простой, использоваться будут два объекта:
- addDeviceButton
- deviceTableView

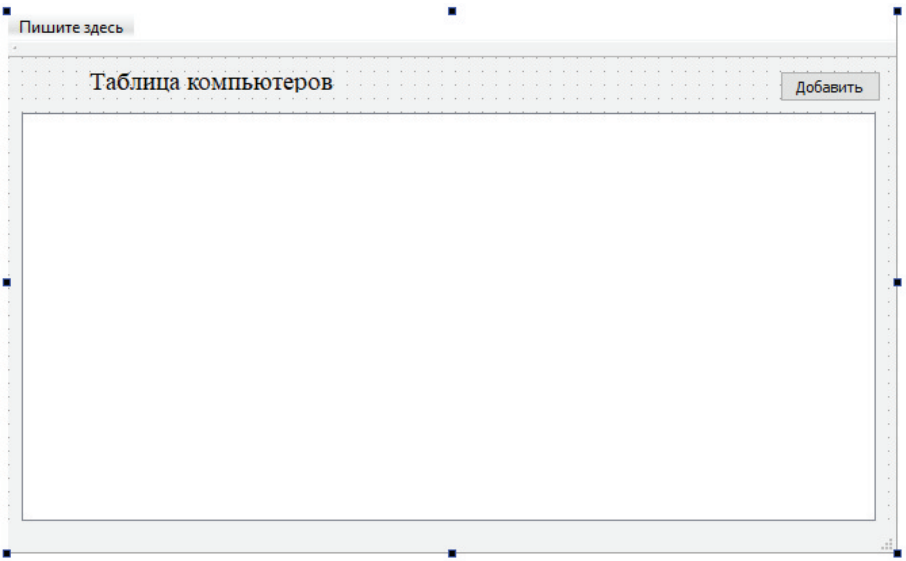

Рисунок 7.1. Форма главного окна приложения

2) в форме диалогового окна используются три поля класса QLineEdit, две кнопки и один элемент ButtonBox, являющийся элементом по умолчанию для класса, который наследуется от класса QDialog, на форме используются следующие объекты:

- buttonBox
- HostnameLineEdit
- IPAddressLineEdit
- MACLineEdit
- nextButton
- previousButton

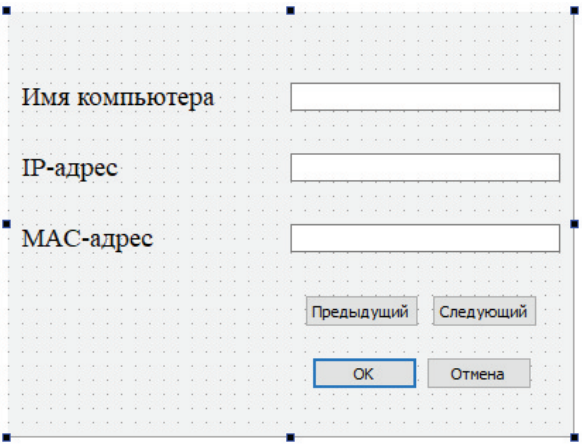

Рисунок 7.2. Форма диалогового окна приложения

В файл проекта необходимо добавить директиву, которая объявляет об использовании библиотек SQL.

Запустите программу. Сделайте снимок экрана (рисунок 7.3). Нажмите кнопку «Добавить». Внесите информацию об одном компьютере. Сделайте снимок экрана (рисунок 7.4 и рисунок 7.5). Добавьте информацию еще о нескольких компьютерах. Сделайте снимок экрана (рисунок 7.6). Проверьте работоспособность всех кнопок. Измените информацию об одном компьютере через диалоговое окно напрямую, два раза нажав на строку таблицы. Сделайте снимок экрана. Проверьте, сохранились ли изменения в таблице и появился ли в каталоге с программой файл базы данных. Сделайте снимок экрана.

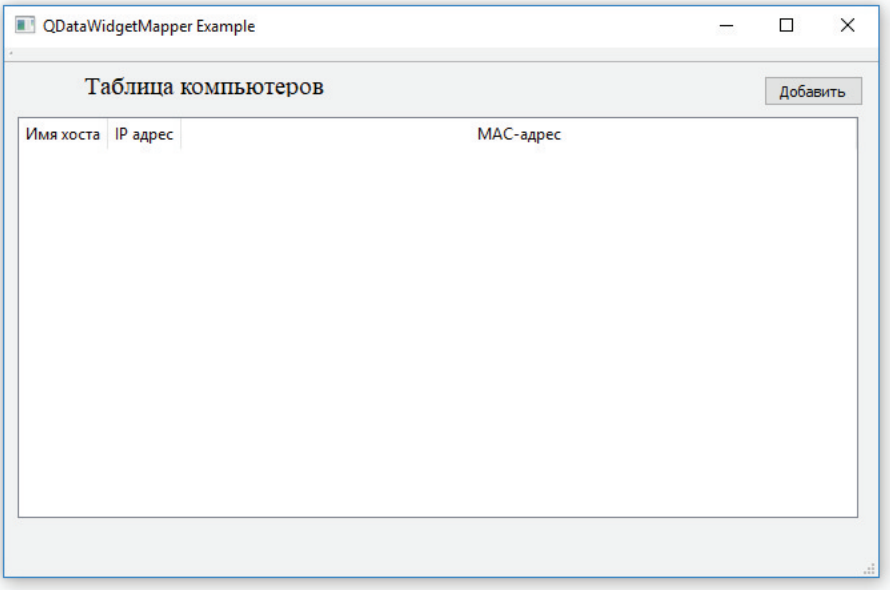

Рисунок 7.3. Главное окно приложения

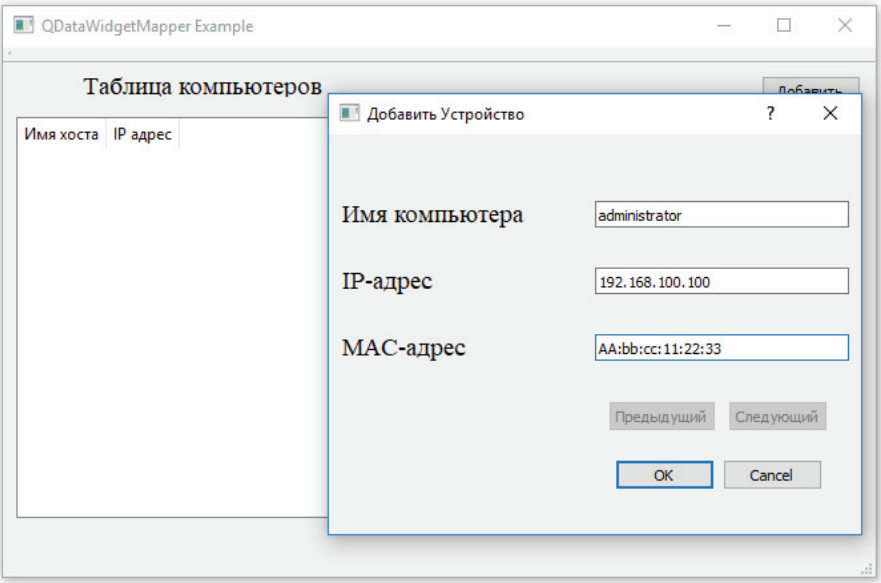

Рисунок 7.4. Добавление информации в пустую базу данных через диалоговое окно приложения

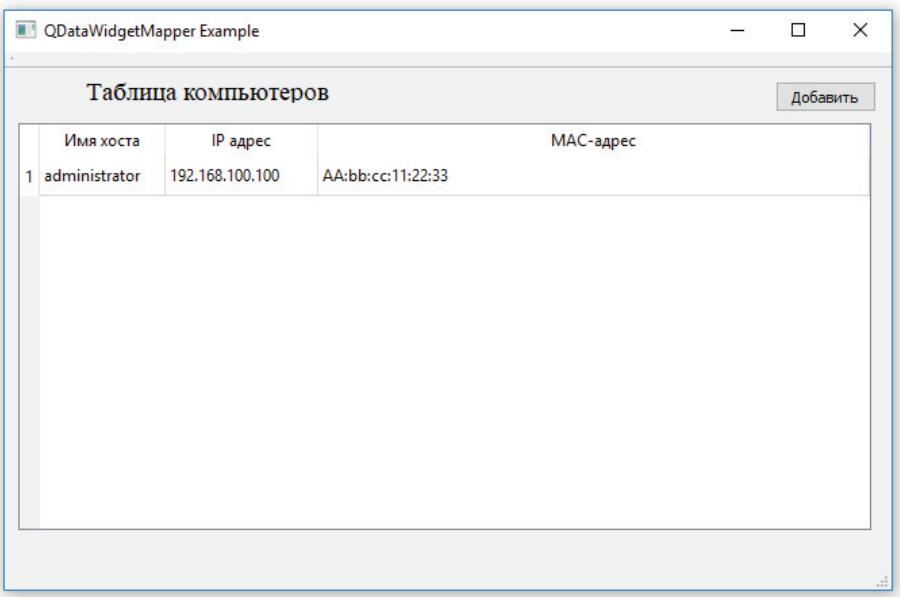

Рисунок 7.5. База данных после добавления информации об одном компьютере

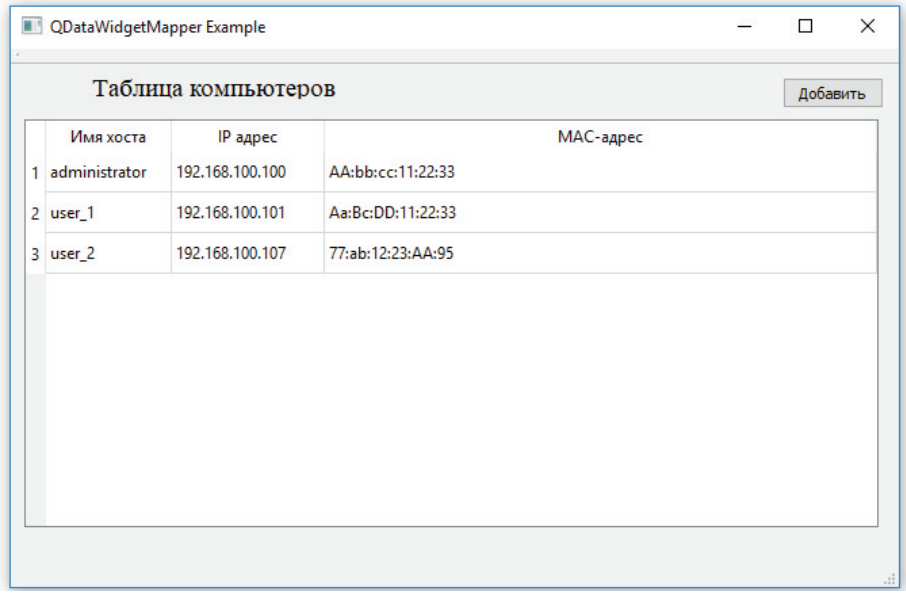

Рисунок 7.6. База данных после добавления информации о нескольких компьютерах

8. ИТОГОВОЕ КОМПИЛИРОВАНИЕ ПРИЛОЖЕНИЯ ДЛЯ ЗАПУСКА НА ДРУГИХ КОМПЬЮТЕРАХ

8.1. Краткая теория

В настоящей работе представлены основные способы по созданию приложения, независимо переносимого на другие компьютеры. За основу рекомендуется взять ПО, созданное в работе № 5. Полагаем, что написанное приложение успешно компилируется и запускается в QtCreator. Чтобы разработать приложение, которое можно будет запускать на любом ПК с нативной для него ОС, как любую другую программу, необходимо скомпилировать проект в режиме «Выпуск». Приложение, созданное на данном этапе, будет запускаться только в среде самого Qt. Это объясняется тем, что исполняемый файл при запуске пытается использовать динамические библиотеки. Во время работы фреймворка они подключаются и используются по умолчанию. Без него исполняемый файл ищет динамические библиотеки в системных файлах ОС. Если определенным образом не указать путь к ним, то разработанное ПО запускаться не будет. Далее рассмотрим основные способы создания готового к запуску ПО.

Способ 1. Ручной сбор dll-библиотек, используемых в Qt-проекте

В ОС Windows после осуществления компиляции в режиме «Выпуск» в папку с названием release, находящуюся в папке build…, нужно скопировать следующие файлы для рассматриваемого примера (из каталога, в котором установлен Qt, например, "C:\Qt\5.12.4\mingw73_32\bin"):

— Qt5Core.dll;
— libwinpthread-1.dll;
— libstdc++-6.dll;
— Qt5Gui.dll;
— Qt5Network.dll (для работы с сетевыми функциями);
— Qt5Widgets.dll;
— libgcc_s_dw2-1.dll.

Остальное зависит от того, какие еще опции указаны в файле с расширением *.pro. Если приложение собрано в режиме «Выпуска», то копировать надо библиотеки без символа d на конце имени файла: Qt5Core.dll и т. д., если собрано в режиме «Отладки», то Qt5Cored.dll, и т. д. В итоге приложение может запускаться путем двойного нажатия на файл с расширением *.exe.

Способ 2. Использование утилиты windeployqt для сбора dll-библиотек

Рассмотрим утилиту **windeployqt**, которая поставляется вместе с *Qt* . Данная утилита является консольным приложением, которое, как правило, идет в поставке с компиляторами **MinGW** или **MSVC.**

Для использования **windeployqt** необходимо запустить консоль, передав в качестве аргумента путь к скомпилированному исполняемому файлу приложения. После этого **windeployqt** соберет все необходимые библиотеки в папке, где располагается исполняемый файл. Утилита **windeployqt** располагается в папке bin, которая в свою очередь располагается в папке компилятора для Qt. Таким образом, путь к папке может быть следующим (обратите внимание на версию Qt):

- для MinGW — C:\QT\5.5\mingw492_32\bin\windeployqt.exe;
- для MSVC — C:\QT\5.5\msvc2013\bin\windeployqt.exe.

Опишем работу с утилитой **windeployqt**. Допустим, имеется скомпилированный исполняемый файл и требуется собрать все необходимые для его работы dll-библиотеки. Для этого сделаем небольшую подготовку и перенесем исполняемый файл в подготовленную для сбора папку. Используем для демонстрации проект (в примере он называется *EColor)*, который написан с использованием компилятора **MinGW.** Исполняемый файл проекта будет размещен по следующему пути — D:\EColor\EColor.exe.

Далее необходимо будет открыть консоль (*cmd*), и перейти в консоли в папку с **windeployqt**. Теперь запускаем утилиту, передав ей путь расположения исполняемого файла, как показано на рисунке 8.1.

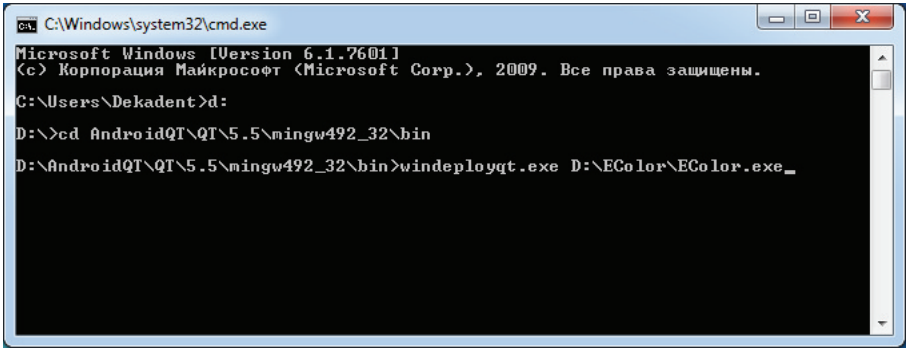

Рисунок 8.1. Запуск утилиты windeployqt с передачей пути расположения исполняемого файла

После этого утилита соберет все необходимые для работы исполняемого файла dll-библиотеки. Выглядеть это будет следующим образом (рисунки 8.2 и 8.3).

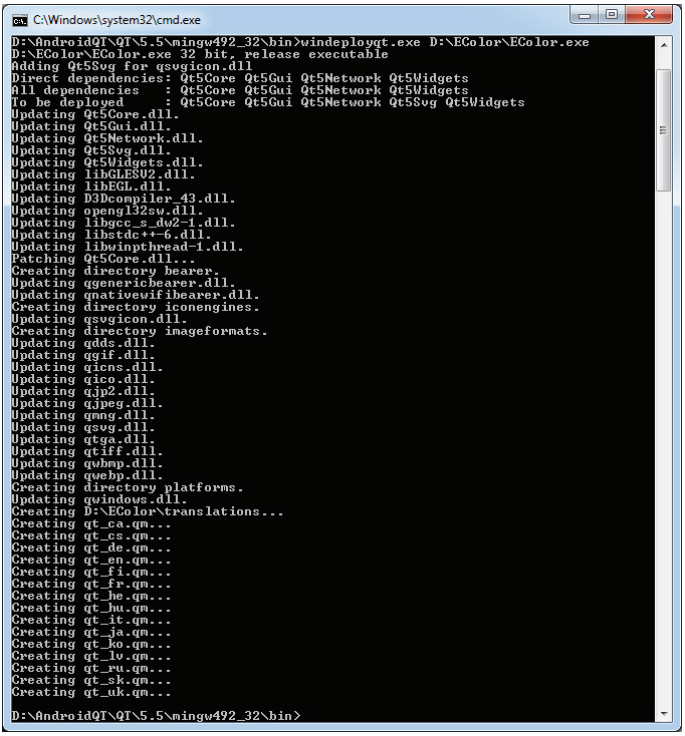

Рисунок 8.2. Процесс сбора утилитой windeployqt необходимых dll-библиотек

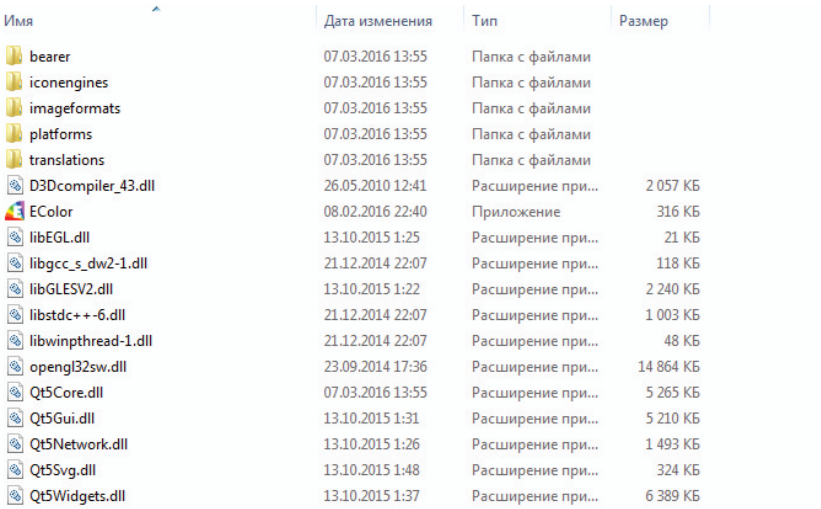

Рисунок 8.3. Собранные утилитой windeployqt необходимые библиотеки в папке с исполняемым файлом

Важно отметить, что могут возникать ошибки при работе утилиты. Например, утилита **windeployqt** может не найти библиотеки компилятора *gcc* и выдать следующую ошибку:

Cannot find GCC installation directory. g++.exe must be in the system path.

Эта ошибка может быть исправлена прописыванием в настройках ОС «Переменные среды» пути к папке bin, которая содержит компилятор *gcc* и соответствующие библиотеки. Для этого перейдите по следующему направлению для *ОС Windows 7*:
Панель Управления / Система / Дополнительные параметры системы / Переменные среды;
для *ОС Windows 10:*
Панель управления / Система и безопасность / Система / Изменение переменных среды текущего пользователя.

Затем добавьте переменную «PATH» в системные переменные, где будут прописаны пути к компилятору *gcc* и к его библиотекам. Это может выглядеть следующим образом:

- Имя переменной: PATH
- Значение переменной: **"D:\AndroidQT\QT\5.5\mingw492_32\bin; D:\AndroidQT\QT\Tools\mingw492_32\bin;%PATH%".**

После этого следует перезагрузить ПК. Теперь ошибка исчезнет, а соответствующие библиотеки должны будут перемещены в папку с исполняемым файлом.

Также некоторые файлы могут быть не собраны с помощью утилиты **windeployqt**, что стоит также учесть и протестировать приложение на другом ПК перед выпуском конечному пользователю. К примеру, при сборке dll-библиотек для проекта *EColor,* утилита **windeployqt** все же пропустила следующие файлы:

- LIBEAY32.dll
- MSVCP90.dll
- MSVCR90.dll
- MSVCR120.dll
- ssleay32.dll

Если в проекте используется QML, то необходимо добавить специальную директиву **–qmldir**, при работе с утилитой **windeployqt**. Данная директива должна будет указать, откуда брать исходники QML, например:

windeployqt --qmldir f:\myApp\sources f:\build-myApp\myApp.exe

Способ 3. Использование «Монитора ресурсов» для сборки dll-библиотек

Рассмотрим наиболее надежный способ выявления используемых приложением библиотек. В очередной раз запустите приложения из папок «Выпуска» (проект, созданный на основе лабораторной работы № 5). Далее откройте встроенный в ОС Windows 10 «Монитор ресурсов» (рисунок 8.4).

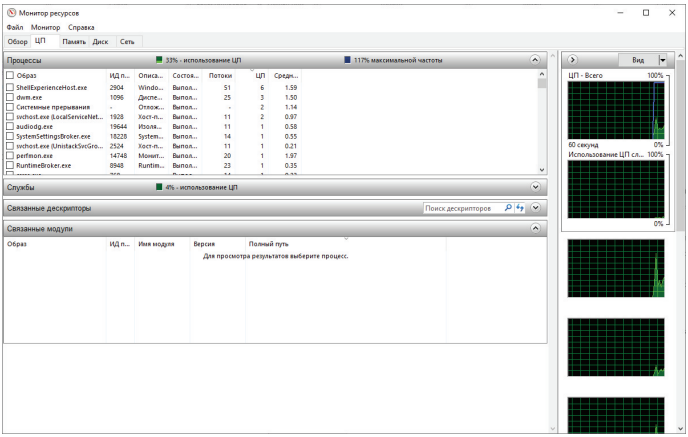

Рисунок 8.4. Окно «Монитора ресурсов»

Теперь необходимо найти и отметить в мониторе ресурсов процессы приложения сервера и клиента (lr5_1.exe и lr5.exe), как показано на рисунке 8.5.

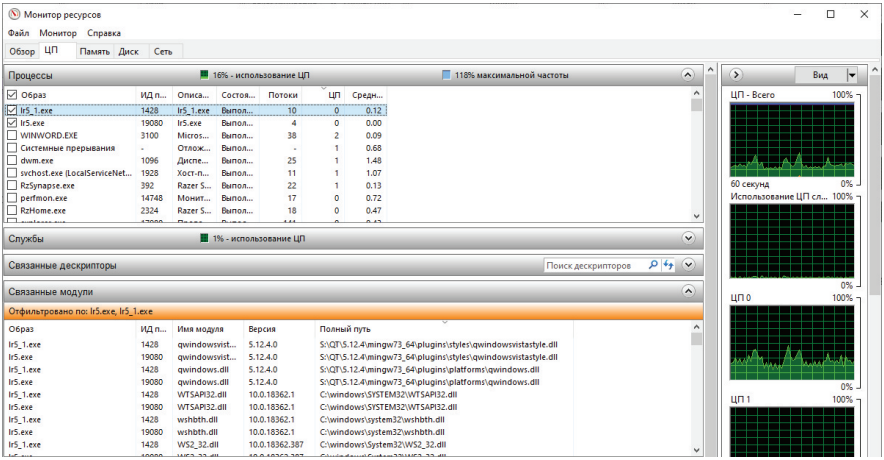

Рисунок 8.5. Отмеченные в «Мониторе ресурсов» процессы приложения сервера и клиента

Как видно из появившегося меню «Отфильтровано», теперь видны все имена библиотек, которые используют эти процессы, а также версии их модулей и полные пути до них (рисунок 8.6).

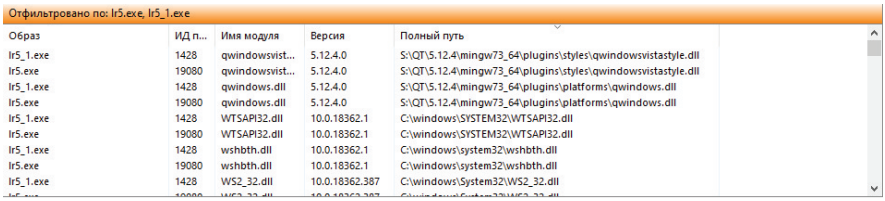

Рисунок 8.6. Меню «Отфильтровано» в мониторе ресурсов

Далее необходимо скопировать все указанные библиотеки (из папок, указанных в меню «Полный путь» по аналогии со *Способом 1*) в директории «Выпуска» соответствующих приложений и убедиться в их работе. Данный способ хоть и не является самым быстрым и может занять некоторое время на сбор всех библиотек, тем не менее позволяет убедиться в том, что во время сбора dll-библиотек не будут упущены какие-либо библиотеки, необходимые для запуска и корректного использования приложения.

Способ 4. Использование дополнительных инструментов для сбора dll-библиотек

Следует отметить, что существуют готовые коммерческие и бесплатно распространяемые инструменты для отслеживания зависимостей приложений. К ним относятся такие программные продукты, как Microsoft Visual Studio Debugger (рисунок 8.7) [4], Dependency walker (рисунок 8.8) [7], Dependencies (рисунок 8.9) [8] и другие [1–3].

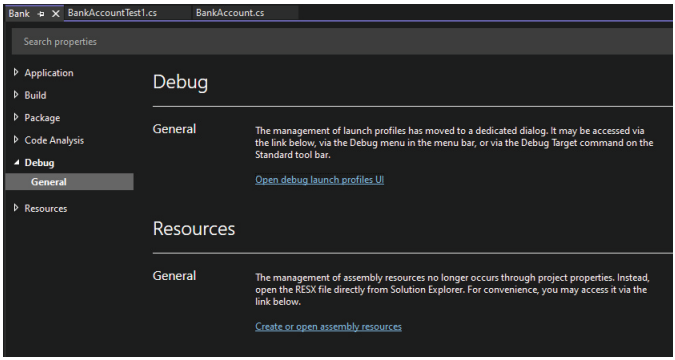

Рисунок 8.7. Настройки отладки в Microsoft Visual Studio Debugger

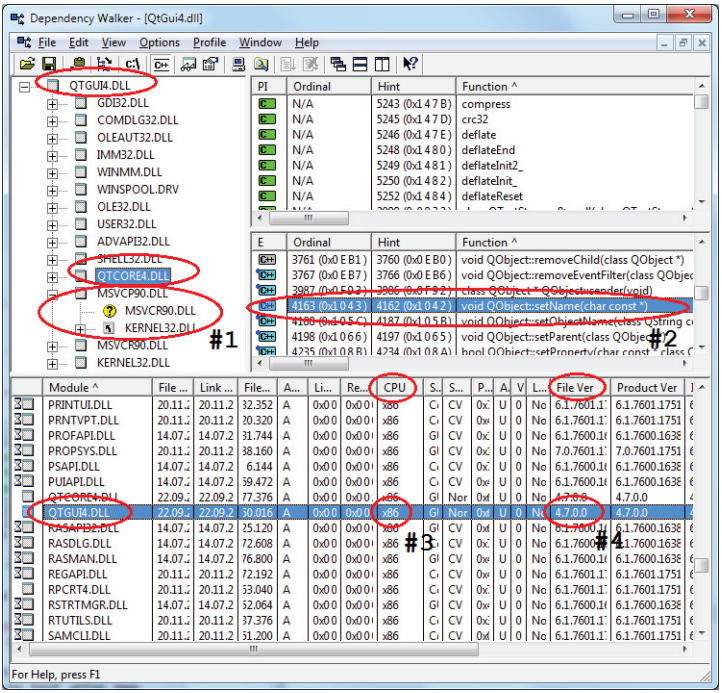

Рисунок 8.8. Анализ зависимостей для запущенного приложения посредством инструмента Dependency walker

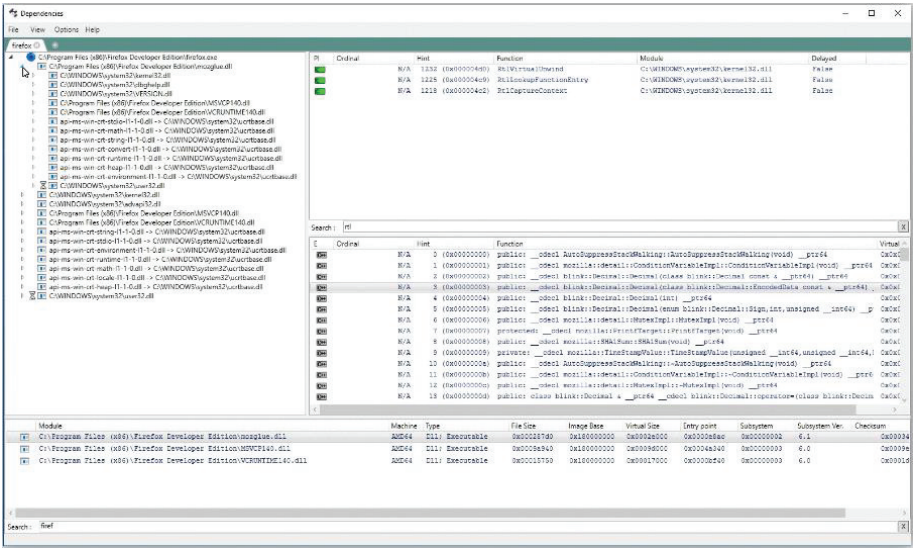

Рисунок 8.9. Использование инструмента Dependencies

8.2. Цель работы

Изучить и практически освоить основные способы создания полноценного переносимого приложения с помощью фреймворка Qt и дополнительных утилит.

8.3. Задание

- Ознакомиться по литературе [1–3, 5, 9–11] с основными понятиями создания переносимых кроссплатформенных программных продуктов.
- В качестве примера программного продукта взять за основу клиент-серверное приложение из лабораторной работы № 5.
- Разработать код программы, реализующий серверную часть приложения.
- Разработать код программы, реализующий клиентскую часть приложения.
- Скомпилировать приложение в режиме «Выпуск».
- Добавить ручным способом (далее *Способ 1*) все необходимые библиотеки для самостоятельного запуска приложения.
- Запустить сначала сервер, затем клиентское приложение.
- Переслать от клиента текстовые сообщения на сервер и получить в ответ подтверждающие сообщения.
- Завершить работу сначала сервера, а затем клиентского приложения.
- Использовать другие способы сбора dll-библиотек, продемонстрировать их работу с рассматриваемым приложением, повторив указанные в 2 предыдущих пунктах задания (запуск ПО и пересылка сообщений) для каждого способа.
- Согласно пунктам выполнения лабораторной работы, сделать необходимые снимки экрана. Изучить полученную информацию и оформить ее в соответствии с требованиями раздела «Содержание отчета».

8.4. Выполнение

В профайле проектов не забудьте подключить модуль network, иначе классы QTcpSocket и QTcpServer будут недоступны. Сначала создайте приложения сервера, затем приложение клиента. Структура проекта:

1. Серверная часть приложения:

lr8.pro — профайл проекта серверной части приложения;
main.cpp — файл с функцией main серверной части приложения;
myserver.h — заголовочный файл окна приложения серверной части;
myserver.cpp — файл исходных кодов окна приложения серверной части.

2. Клиентская часть приложения:
lr8_1.pro — профайл проекта клиентской части приложения;
main.cpp — файл с функцией main клиентской части приложения;
myclient.h — заголовочный файл окна приложения клиентской части;
myclient.cpp — файл исходных кодов окна приложения клиентской части;
Программный код файлов (кроме профайлов проекта) совпадает с содержимым файлов лабораторной работы № 5.

Реализация Способа 1

Соберите проекты в режиме «Выпуск». Закройте фреймворк Qt. Скопируйте из папки, в которой установлен Qt, файлы библиотек в папки release проектов lr8 и lr8_1. Проверьте содержимое папок. Сделайте снимки экрана (рисунок 8.10 и рисунок 8.11).

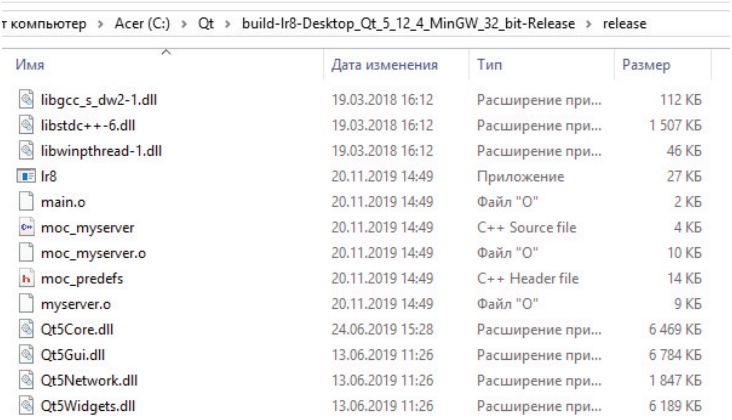

Рисунок 8.10. Содержимое папки release проекта lr8

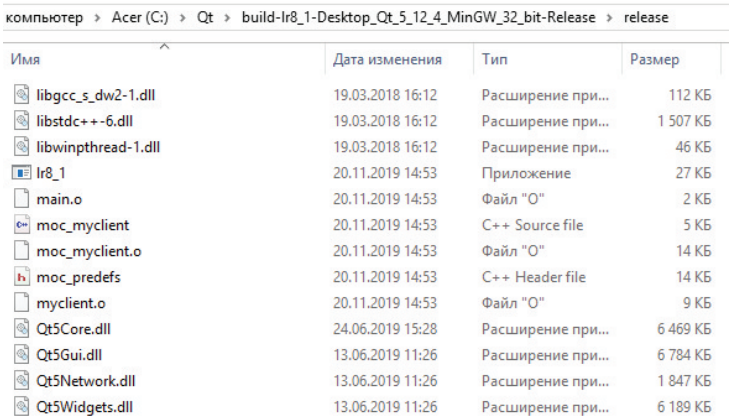

Рисунок 8.11. Содержимое папки release проекта lr8_1

Запустите исполняемый файл lr8.exe серверной части приложения, сделайте снимок экрана (рисунок 8.12).

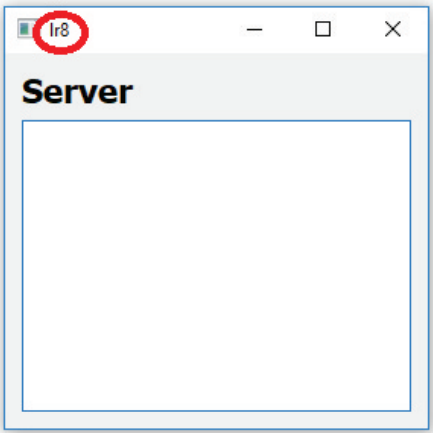

Рисунок 8.12. Окно серверной части приложения, запущенное через файл lr8.exe

После запуска клиентской части приложения сделайте снимок экрана (рисунок 8.13).

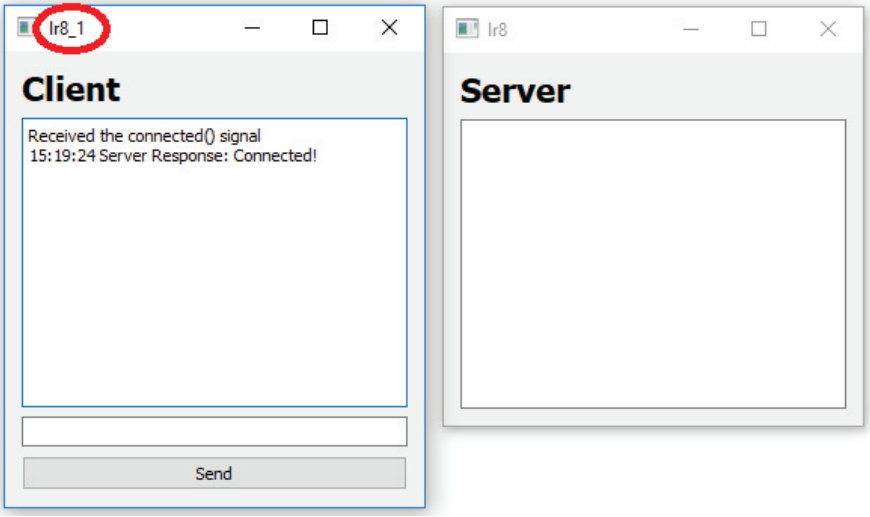

Рисунок 8.13. Окна серверной и клиентской частей приложения, запущенные через файлы lr8.exe и lr8_1.exe соответственно

Проверьте, был ли успешно подключен клиент к серверу? Перешлите несколько текстовых сообщений от клиента на сервер. Что получает клиент в ответ от сервера? Сделайте снимок экрана (рисунок 8.14).

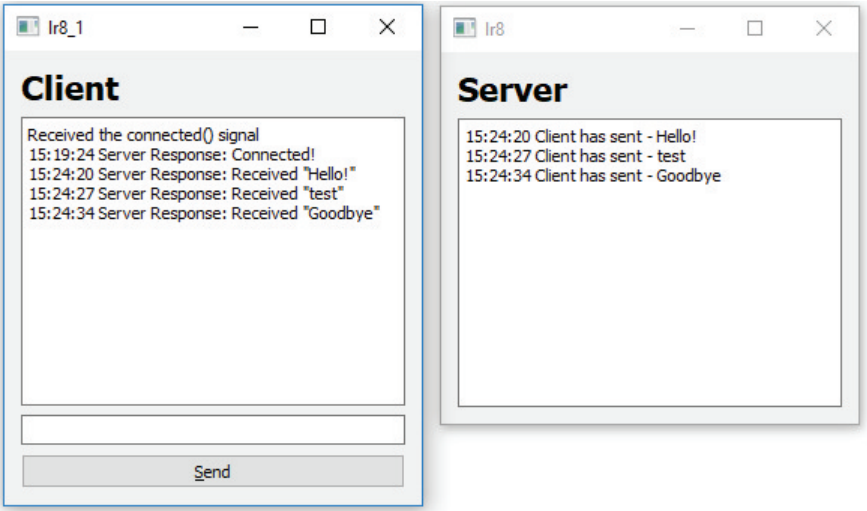

Рисунок 8.14. Успешная передача сообщений от клиента к серверу, которые запущены через файлы lr8.exe и lr8_1.exe соответственно

Отключите сервер. Сделайте снимок экрана.

Используйте другие способы сбора dll-библиотек, продемонстрируйте их работу с рассматриваемым приложением, повторив указанные ранее задания (запуск ПО и пересылка сообщений) для каждого способа. Сделайте необходимы снимки экранов, подтверждающие выполнение *Способов 2–4*, приведенных в разделе «Краткая теория».

СОДЕРЖАНИЕ ОТЧЕТА

В индивидуальном отчете должны быть указаны цель, задание, краткие теоретические сведения, представлены необходимые листинги программного кода, снимки экрана и пояснения к ним. Следует проанализировать полученные данные, ответить на соответствующие контрольные вопросы и сделать выводы.

КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что собой представляет Qt?
2. Что такое «виджет»?
3. В чем состоит смысл понятия «кроссплатформенность»?
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?
8. Каким методом можно получить текущую дату?
9. Каким методом можно получить текущее время?
10. Как работает таймер в Qt?
11. Объясните порядок работы с текстовыми и бинарными файлами в соответствии с выполненными заданиями.
12. Для чего необходимо подключать модуль multimedia?
13. Какие классы используются для воспроизведения аудиофайлов?
14. Как организовать работу с плейлистом в Qt?
15. Какие средства по управлению стилями есть в Qt?
16. Как можно применять стили в Qt?
17. Что такое процессы?
18. Что представляют собой потоки?
19. Как работает метод run() при многопоточном программировании в Qt?
20. Какие способы передачи объектов между потоками существуют в Qt?
21. Что такое сокет? Зачем используются классы QTcpSocket и QTcpServer?
22. Какие методы применяются для организации серверной части приложения?
23. Какие методы применяются для организации клиентской части приложения?
24. Опишите реализацию FTP-серверов в Qt.
25. Как осуществляются http запросы в Qt?
26. С какими СУБД может работать Qt?
27. Какие методы применяются для работы с базами данных?
28. Каково назначение класса QSqlTableModel?
29. Какие файлы необходимы для автономной работы приложения?
30. Как определить, какие файлы библиотек следует использовать, для данного режима компилирования программы?

СПИСОК ЛИТЕРАТУРЫ

1. Беспалов, Д. А. Операционные системы реального времени и технологии разработки кроссплатформенного программного обеспечения. Часть 1: учебное пособие / Д. А. Беспалов, С. М. Гушанский, Н. М. Коробейникова; Южный федеральный университет. – Ростов-на-Дону; Таганрог: Издательство Южного федерального университета, 2019. – 139 с.
2. Беспалов, Д. А. Операционные системы реального времени и технологии разработки кроссплатформенного программного обеспечения. Часть 2: учебное пособие / Д. А. Беспалов, С. М. Гушанский, Н. М. Коробейникова; Южный федеральный университет. – Ростов-на-Дону; Таганрог: Издательство Южного федерального университета, 2019. – 168 с.
3. Беспалов, Д. А. Операционные системы реального времени и технологии разработки кроссплатформенного программного обеспечения. Часть 3: учебное пособие / Д. А. Беспалов, С. М. Гушанский, Н.М. Коробейникова; Южный федеральный университет. – Ростов-на-Дону; Таганрог: Издательство Южного федерального университета, 2019. – 216 с.
4. Microsoft Visual Studio Debugger [Электронный ресурс]: // Документация по работе с инструментом Microsoft Visual Studio Debugger. – Режим доступа: https://learn.microsoft.com/ru-ru/visualstudio/debugger/debugger-feature-tour?view=vs-2019, свободный. – Загл. с экрана. (дата обращения 29.09.2022).
5. Qt [Электронный ресурс]: // Документация по работе с Qt. – Режим доступа: https://doc.qt.io/, свободный. – Загл. с экрана. (дата обращения 29.09.2022).
6. QTcpSocket class [Электронный ресурс]: // Документация по работе с QTcpSocket class. – Режим доступа: https://doc.qt.io/qt-6/qtcpsocket.html, свободный. – Загл. с экрана. (дата обращения 29.09.2022).
7. Dependency walker [Электронный ресурс]: // Официальный сайт по работе с инструментом Dependency walker. – Режим доступа: https://www.dependencywalker.com/, свободный. – Загл. с экрана. (дата обращения 29.09.2022).
8. Dependencies [Электронный ресурс]: // Официальный сайт по работе с инструментом Dependencies. – Режим доступа: https://lucasg.github.io/Dependencies/, свободный. – Загл. с экрана. (дата обращения 29.09.2022).
9. Qt уроки [Электронный ресурс]: // Видеокурс EVILEG. – Режим доступа: https://youtube.com/playlist?list=PL0MK5M74MpyIf1rX-VcX7hL6OjaFjkhbj, свободный. – Загл. с экрана. (дата обращения 20.02.2023).
10. Документация по работе с Qt [Электронный ресурс]: // Русскоязычная документация по Qt. – Режим доступа: https://iot-embedded.ru/?s=Qt, свободный. – Загл. с экрана. (дата обращения 20.02.2023).
11. Qt/C++ [Электронный ресурс]: // Статьи по работе с фреймворком Qt. – Режим доступа: https://evileg.com/ru/knowledge/qt/, свободный. – Загл. с экрана. (дата обращения 20.02.2023).

СПИСОК СОКРАЩЕНИЙ

БД – база данных

ОС – операционная система

ПО – программное обеспечение

СУБД – система управления базами данных

ФС – файловая система

FTP (File Transfer Protocol) – протокол передачи файлов

GUI (Graphical User Interface) – графический пользовательский интерфейс

HTTP (HyperText Transfer Protocol) – протокол передачи гипертекста

SCSI (Small Computer System Interface) – набор стандартов для физического подключения и передачи данных между компьютерами и периферийными устройствами

SQL (Structured Query Language) – язык структурированных запросов

TCP (Transmition Control Protocol) – протокол управления передачей данных, обеспечивает гарантированную доставку

UDP (User Datagram Protocol) – протокол пользовательских датаграмм, обеспечивает негарантированную доставку

ПРИЛОЖЕНИЕ

Листинг П.1.1. Содержание файла main.cpp

```
#include <QApplication> //подключили заголовочные файлы с определениями
классов QApplication (приложение)
#include <QPushButton>  //и QPushButton (кнопка)

int main(int argc, char *argv[ ]) //обычный для C++ заголовок главной функции
main с аргументами командной строки;
{
QApplication app(argc, argv); /*
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) */

QPushButton button("Hello, World!"); /*создали главное окно приложения,
которое представляет собой обычную кнопку с текстом «Hello, World!» */

button.resize(200, 60); //определили размеры окна (ширину и высоту) в пикселях
button.show( ); //вывели окно на экран
return app.exec( ); /* 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) */
}
```

Листинг П.1.2. Содержание файла main.cpp

```
#include <QApplication> //подключили заголовочные файлы с определениями
классов QApplication (приложение)
#include <QPushButton>  // QPushButton (кнопка)
#include <QLabel>       // QLabel (надпись)

int main(int argc, char *argv[ ]) //обычный для C++ заголовок главной функции
main с аргументами командной строки;
{
QApplication app(argc, argv); /*
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) */

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

```
label->show( ); // вывод сообщения с использованием указателей и класса
QLabel

return app.exec( ); /* 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) */
}
```

Листинг П.1.3. Содержание файла main.cpp

```
#include <QApplication> //подключили заголовочные файлы с определениями
классов QApplication (приложение)
#include <QPushButton>  // QPushButton (кнопка)
#include <QLabel>       // QLabel (надпись)

int main(int argc, char *argv[ ]) //обычный для C++ заголовок главной функции
main с аргументами командной строки;
{
QApplication app(argc, argv); /*
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) */

QLabel label("<h2><i>Hello,</i> " "<font color = red > Qt! </font></h2>");
label.show( ); // вывод сообщения без использования указателей

return app.exec( ); /* 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) */
}
```

Листинг П.1.4. Содержание файла main.cpp

```
#include <QApplication> //подключили заголовочные файлы с определениями
классов QApplication (приложение)
#include <QPushButton>  // QPushButton (кнопка)
#include <QLabel>       // QLabel (надпись)

int main(int argc, char *argv[ ]) //обычный для C++ заголовок главной функции
main с аргументами командной строки;
{
QApplication app(argc, argv); /*
```

```
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) */

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

return app.exec( ); /* 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) */
}
```

Листинг П.1.5. Содержание файла main.cpp

```
#include <QApplication> //подключили заголовочные файлы с определениями
классов QApplication (приложение)
#include <QPushButton>  // QPushButton (кнопка)
#include <QLabel>       // QLabel (надпись)
#include <QDate>        // QDate (Класс QDate представляет собой структуру
данных для хранения дат и проведения с ними разного рода операций)
#include <QTime>        // QTime (аналогичен классу QDate, используется для
работы со временем)
#include <QDateTime>    // QDateTime
#include <QTimer>       // QTimer (таймер)
#include <QTimerEvent>  // QTimerEvent (Событие таймера)

int main(int argc, char *argv[ ]) //обычный для C++ заголовок главной функции
main с аргументами командной строки;
{
QApplication app(argc, argv); /*
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) */

QDate dateToday = QDate::currentDate(); // создание объекта dateToday класса
QDate. Для получения текущей даты нужно вызвать метод currentDate()
QTime Now = QTime::currentTime();// создание объекта Now класса QTime. Для
получения текущего времени нужно вызвать метод currentTime()
QString str1; // создание объекта str1 класса QString для записи в него текущей
даты
```

```
QString str2; // создание объекта str2 класса QString для записи в него текущего времени
QString str3; // создание объекта str3 класса QString для записи в него информации о текущей дате и времени
str1 = dateToday.toString("dd/MM/yy"); // Метод toString() позволяет получить текстовое представление даты в необходимом формате.
str2 = Now.toString("hh:mm:ss"); // Метод toString() позволяет получить текстовое представление времени в необходимом формате.
str3 = str1 + "   " + str2; // Создание строки с информацией и о дате, и о времени, разделеных символами пробелов
QLabel label("<H2><CENTER>" + str3 + "<H2><CENTER>"); //создание объекта label класса QLabel
label.resize(200,60);
label.show( );

return app.exec( ); /* 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) */
}
```

Листинг П.1.6. Содержание файла main.cpp

```
#include <QApplication> //подключили заголовочные файлы с определениями классов QApplication (приложение)
#include <QPushButton>  // QPushButton (кнопка)
#include <QLabel>       // QLabel (надпись)
#include <QDate>        // QDate (Класс QDate представляет собой структуру данных для хранения дат и проведения с ними разного рода операций)
#include <QTime>        // QTime (аналогичен классу QDate, используется для работы со временем)
#include <QDateTime>    // QDateTime
#include <QTimer>       // QTimer (таймер)
#include <QTimerEvent>  // QTimerEvent (Событие таймера)

/* Класс BlinkLabel содержит атрибут булевого типа m_bBlink,
 управляющий отображением надписи, и атрибут m_strText, содержащий текст надписи.
 * В конструктор класса BlinkLabel передается интервал мигания nInterval.
 * По умолчанию он равен 200 миллисекундам.
 * Вызов метода startTimer() запускает таймер со значением переданного интервала запуска.
 * По истечении этого интервала происходит создание события QTimerEvent,
 которое передается в метод timerEvent(), в котором происходит смена значения атрибута m_bBlink на противоположное.
```

```
 * И в соответствии с установленным значением, методом setText() выполняется
одно из действий:
false — вся область надписи очищается установкой пустой строки;
true — текст надписи устанавливается заново.*/

class BlinkLabel : public QLabel {
private:
   bool    m_bBlink;
   QString m_strText;

protected:
   virtual void timerEvent(QTimerEvent*)
   {
      m_bBlink = !m_bBlink;
      setText(m_bBlink ? m_strText : "");
   }

public:
   BlinkLabel(const QString& strText,
              int      nInterval = 200,
              QWidget* pwgt      = 0
          )
      : QLabel(strText, pwgt)
      , m_bBlink(true)
      , m_strText(strText)
   {
      startTimer(nInterval);
   }
};

int main(int argc, char *argv[ ]) //обычный для C++ заголовок главной функции
main с аргументами командной строки;
{
QApplication app(argc, argv); /*
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)

В функции main() создается виджет класса BlinkLabel, в конструктор которого
первым параметром передается отображаемый текст в формате RichText. */

BlinkLabel   lbl("<FONT COLOR =
RED><CENTER>Blink</CENTER></FONT>");
lbl.resize(200, 60);
```

```
    lbl.show();

return app.exec( ); /* 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) */
}
```

Листинг П.1.7. Содержание файла main.cpp

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

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
/* Всё, что надо в этом проекте, — это слот, который будет реагировать на
срабатывание QTimer, да сам объект этого класса. */
```

Листинг П.1.8. Содержание файла mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QLabel>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    /* Изменим QLabel, чтобы он был больше и заметнее в окне */
    QFont font("Times", 28, QFont::Bold);
    ui->label->setFont(font);

    /* При первом запуске приложения поместим текущее время в QLabel */
    ui->label->setText(QTime::currentTime().toString("hh:mm:ss"));

    /* Инициализируем Таймер и подключим его к слоту, который будет
обрабатывать timeout() таймера */
    timer = new QTimer();
    connect(timer, SIGNAL(timeout()), this, SLOT(slotTimerAlarm()));
```

```
    timer->start(1000); // И запустим таймер
}

MainWindow::~MainWindow()
{
    delete ui;
}

/* Слот для обработки timeout() таймера */
void MainWindow::slotTimerAlarm()
{
    /* Ежесекундно обновляем данные по текущему времени
     * Перезапускать таймер не требуется */
    ui->label->setText(QTime::currentTime().toString("hh:mm:ss"));
}
```

Листинг П.1.9. Содержание файла mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QFont>
#include <QTimer>
#include <QTime>
#include <QLabel>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT


public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

    /* Будем использовать только один слот */
private slots:
    void slotTimerAlarm();

private:
    Ui::MainWindow *ui;
```

```
    /* Да сам објект QTimer */
    QTimer *timer;
};

#endif // MAINWINDOW_H
```

Листинг П.1.10. Содержание файла main.cpp

```
#include <QApplication> //подключили заголовочные файлы с определениями
классов QApplication (приложение)
#include <QPushButton>  // QPushButton (кнопка)
#include <QLabel>       // QLabel (надпись)
#include <QFile>        // QFile (класс для проведения операций с файлами)
#include <QBuffer>      // QBuffer (позволяет записывать и считывать данные в
массив QByteArray, как будто бы это устройство или файл)
#include <QDebug>

int main(int argc, char *argv[ ]) //обычный для C++ заголовок главной функции
main с аргументами командной строки;
{
QApplication app(argc, argv); /*
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) */

QFile file1("file1.dat");
QFile file2("file2.dat");

if(file2.exists())
{
    //Файл уже существует. Перезаписать?
}

if (!file1.open(QIODevice::ReadOnly))
{
    qDebug() << "Ошибка открытия для чтения";
}

if(!file2.open(QIODevice::WriteOnly))
{
    qDebug() << "Ошибка открытия для записи";
}

char a [1];
while(!file1.atEnd())
```

```
{
   int nBlocksize = file1.read(a, sizeof(a));
   file2.write(a, nBlocksize);
}
file1.close(); // обязательно необходимо закрыть файлы (потоки ввода/вывода)
file2.close();
return app.exec( ); /* 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) */
}
```

Листинг П.1.11. Содержание файла main.cpp

```
#include <QApplication> //подключили заголовочные файлы с определениями
классов QApplication (приложение)
#include <QPushButton>  // QPushButton (кнопка)
#include <QLabel>       // QLabel (надпись)
#include <QFile>        // QFile (класс для проведения операций с файлами)
#include <QBuffer>      // QBuffer (позволяет записывать и считывать данные в
массив QByteArray, как будто бы это устройство или файл)
#include <QDebug>

int main(int argc, char *argv[ ]) //обычный для C++ заголовок главной функции
main с аргументами командной строки;
{
QApplication app(argc, argv); /*
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) */
/*Если требуется считать или записать данные за один раз, то используют
методы QIODevice::write() и QIODevice::readAll(). Все данные можно считать в
объект класса QByteArray, а потом записать из него в другой файл */
QFile file1("file1.dat");
QFile file2("file2.dat");

if(file2.exists())
{
   //Файл уже существует. Перезаписать?
}

if(!file1.open(QIODevice::ReadOnly))
{
   qDebug() << "Ошибка открытия для чтения";
}
```

```
if(!file2.open(QIODevice::WriteOnly))
{
   qDebug() << "Ошибка открытия для записи";
}

QByteArray a = file1.readAll();
file2.write(a);
file1.close();
file2.close();

return app.exec( ); /* 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) */
}
```

Листинг П.1.12. Содержание файла main.cpp

```
#include <QApplication> //подключили заголовочные файлы с определениями
классов QApplication (приложение)
#include <QPushButton>  // QPushButton (кнопка)
#include <QLabel>       // QLabel (надпись)
#include <QFile>        // QFile (класс для проведения операций с файлами)
#include <QBuffer>      // QBuffer (позволяет записывать и считывать данные в
массив QByteArray, как будто бы это устройство или файл)
#include <QDebug>

int main(int argc, char *argv[ ]) //обычный для C++ заголовок главной функции
main с аргументами командной строки;
{
QApplication app(argc, argv); /*
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) */

QByteArray arr;
QBuffer buffer(&arr);
buffer.open(QIODevice::WriteOnly);
QDataStream out(&buffer);
out << QString("1111");
qDebug()<<arr;
/*сами данные сохраняются внутри объекта класса QByteArray.
 * При помощи метода buffer() можно получить константную ссылку к
внутреннему объекту QByteArray, а при помощи метода setBuffer() можно
```

```
устанавливать другой объект QByteArray для его использования в качестве внутреннего.*/

return app.exec( ); /* 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) */
}
```

Листинг П.1.13. Содержание файла main.cpp

```
#include <QApplication> //подключили заголовочные файлы с определениями классов QApplication (приложение)
#include <QPushButton>  // QPushButton (кнопка)
#include <QLabel>       // QLabel (надпись)
#include <QFile>        // QFile (класс для проведения операций с файлами)
#include <QBuffer>      // QBuffer (позволяет записывать и считывать данные в массив QByteArray, как будто бы это устройство или файл)
#include <QDebug>

int main(int argc, char *argv[ ]) //обычный для C++ заголовок главной функции main с аргументами командной строки;
{
QApplication app(argc, argv); /*
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) */

QFile file ("file.txt");
if(file.open(QIODevice::ReadOnly))
{
   QTextStream stream(&file);
   QString str;
   while (!stream.atEnd())
   {
      str = stream.readLine();
      qDebug() << str;
   }
   if(stream.status()!= QTextStream::Ok)
   {
      qDebug() << "Ошибка чтения файла";
   }
   file.close();
}
```

```
return app.exec( ); /* 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) */
}
```

Листинг П.1.14. Содержание файла main.cpp

```
#include <QApplication> //подключили заголовочные файлы с определениями классов QApplication (приложение)
#include <QPushButton>  // QPushButton (кнопка)
#include <QLabel>       // QLabel (надпись)
#include <QFile>        // QFile (класс для проведения операций с файлами)
#include <QBuffer>      // QBuffer (позволяет записывать и считывать данные в массив QByteArray, как будто бы это устройство или файл)
#include <QDebug>

int main(int argc, char *argv[ ]) //обычный для C++ заголовок главной функции main с аргументами командной строки;
{
QApplication app(argc, argv); /*
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) */

QFile file("myfile.txt");
if(file.open(QIODevice::ReadOnly))
{
QTextStream stream(&file);
QString str = stream.readAll(); //Методом QTextStream::readAll() можно считать сразу весь текстовый файл в строку.
qDebug()<<str;

if(stream.status()!= QTextStream::Ok)
   {
      qDebug() << "Ошибка чтения файла";
   }
   file.close();
}
/*Чтобы записать текстовую информацию в файл, необходимо создать объект класса QFile и воспользоваться оператором <<.
 * Перед записью можно провести необходимые преобразования строки.*/

QFile file1("file.txt");
QString str = "This is a test";
```

```
if (file1.open(QIODevice::WriteOnly))
{
   QTextStream stream(&file1);
   stream << str.toUpper(); //Запишет-THIS IS A TEST
   file1.close();
   if (stream.status() != QTextStream::Ok)
   {
      qDebug() << "Ошибка записи файла";
   }
}

return app.exec( ); /* 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) */
}
```

Листинг П.2.1. Содержание файла lr2.pro

```
#-------------------------------------------------
# Project created by QtCreator 2019-09-21T14:54:42
#-------------------------------------------------

QT       += core gui
QT       += core gui multimedia

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = lr2
TEMPLATE = app

# The following define makes your compiler emit warnings if you use
# any feature of Qt which has been marked as deprecated (the exact warnings
# depend on your compiler). Please consult the documentation of the
# deprecated API in order to know how to port your code away from it.
DEFINES += QT_DEPRECATED_WARNINGS

# You can also make your code fail to compile if you use deprecated APIs.
# In order to do so, uncomment the following line.
# You can also select to disable deprecated APIs only up to a certain version of Qt.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all
the APIs deprecated before Qt 6.0.0

CONFIG += c++11
SOURCES += \
        main.cpp \
```

```
    widget.cpp

HEADERS += \
    widget.h

FORMS += \
    widget.ui

#Важно подключить ресурсный файл, который создается при нажатии правой
кнопкой мыши на проетке
# Добавить новый - Qt - Файл ресурсов Qt - дайте имя "buttons"

RESOURCES += \
   buttons.qrc \
   buttons.qrc

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

Листинг П.2.2. Содержание файла main.cpp

```
#include "widget.h"
#include <QApplication>
int main(int argc, char *argv[])
{
   QApplication a(argc, argv);
   Widget w;
   w.show();

   return a.exec();
}
```

Листинг П.2.3. Содержание файла widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QStandardItemModel>
#include <QMediaPlayer>
#include <QMediaPlaylist>

namespace Ui {
class Widget;
}
```

```
class Widget : public QWidget
{
    Q_OBJECT

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

private slots:
    void on_btn_add_clicked();              // Слот для обработки добавления треков
через диалоговое окно

private:
    Ui::Widget *ui;
    QStandardItemModel  *m_playListModel;   // Модель данных плейлиста для
отображения
    QMediaPlayer        *m_player;          // Проигрыватель треков
    QMediaPlaylist      *m_playlist;        // Плейлист проигрывателя
};

#endif // WIDGET_H
```

Листинг П.2.4. Содержание файла widget.cpp

```
#include "widget.h"
#include "ui_widget.h"
#include <QFileDialog>
#include <QDir>

Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    ui->setupUi(this);
    // Настройка таблицы плейлиста
    m_playListModel = new QStandardItemModel(this);
    ui->playlistView->setModel(m_playListModel);    // Устанавливаем модель
данных в TableView
    // Устанавливаем заголовки таблицы
    m_playListModel->setHorizontalHeaderLabels(QStringList()  << tr("Audio
Track")
                                               << tr("File Path"));
    ui->playlistView->hideColumn(1);    // Скрываем колонку, в которой хранится
путь к файлу
    ui->playlistView->verticalHeader()->setVisible(false);                // Скрываем
нумерацию строк
```

```
    ui->playlistView->setSelectionBehavior(QAbstractItemView::SelectRows);  //
Включаем выделение строк
    ui->playlistView->setSelectionMode(QAbstractItemView::SingleSelection); //
Разрешаем выделять только одну строку
    ui->playlistView->setEditTriggers(QAbstractItemView::NoEditTriggers);   //
Отключаем редактирование
    // Включаем подгонку размера последней видимой колонки к ширине
TableView
    ui->playlistView->horizontalHeader()->setStretchLastSection(true);

    m_player = new QMediaPlayer(this);          // Инициализируем плеер
    m_playlist = new QMediaPlaylist(m_player);  // Инициализируем плейлист
    m_player->setPlaylist(m_playlist);          // Устанавливаем плейлист в плеер
    m_player->setVolume(100);                   // Устанавливаем громкость
воспроизведения треков
    m_playlist->setPlaybackMode(QMediaPlaylist::Loop);  // Устанавливаем
циклический режим проигрывания плейлиста

    // подключаем кнопки управления к слотам управления
    // Здесь отметим, что навигация по плейлисту осуществляется именно через
плейлист
    // а запуск/пауза/остановка — через сам плеер
    connect(ui->btn_previous, &QToolButton::clicked, m_playlist,
&QMediaPlaylist::previous);
    connect(ui->btn_next, &QToolButton::clicked, m_playlist,
&QMediaPlaylist::next);
    connect(ui->btn_play, &QToolButton::clicked, m_player, &QMediaPlayer::play);
    connect(ui->btn_pause, &QToolButton::clicked, m_player,
&QMediaPlayer::pause);
    connect(ui->btn_stop, &QToolButton::clicked, m_player, &QMediaPlayer::stop);

    // При двойном нажатии по треку в таблице устанавливаем трек в плейлисте
    connect(ui->playlistView, &QTableView::doubleClicked, [this](const
QModelIndex &index){
        m_playlist->setCurrentIndex(index.row());
    });

    // при изменении индекса текущего трека в плейлисте устанавливаем
название файла в специальной текстовой метке
    connect(m_playlist, &QMediaPlaylist::currentIndexChanged, [this](int index){
        ui->currentTrack->setText(m_playListModel->data(m_playListModel-
>index(index, 0)).toString());
    });
}
```

```
Widget::~Widget()
{
    delete ui;
    delete m_playListModel;
    delete m_playlist;
    delete m_player;
}

void Widget::on_btn_add_clicked()
{
    // С помощью диалога выбора файлов делаем множественный выбор mp3-
файлов
    QStringList files = QFileDialog::getOpenFileNames(this,
                                        tr("Open files"),
                                        QString(),
                                        tr("Audio Files (*.mp3)"));

    // Далее устанавливаем данные по именам и пути к файлам
    // в плейлист и таблицу, отображающую плейлист
    foreach (QString filePath, files) {
        QList<QStandardItem *> items;
        items.append(new QStandardItem(QDir(filePath).dirName()));
        items.append(new QStandardItem(filePath));
        m_playListModel->appendRow(items);
        m_playlist->addMedia(QUrl(filePath));
    }
}
```

Листинг П.3.1. Содержание файла Ubuntu.qss

```
/*
Ubuntu Style Sheet for QT Applications
Author: Jaime A. Quiroga P.
Company: GTRONICK
Last updated: 09/10/2019 (dd/mm/yyyy), 12:31.
Available at: https://github.com/GTRONICK/QSS/blob/master/Ubuntu.qss
*/
QMainWindow {
        background-color:#f0f0f0;
}
QCheckBox {
        padding:2px;
}
QCheckBox:hover {
        border-radius:4px;
        border-style:solid;
```

```
        border-width:1px;
        padding-left: 1px;
        padding-right: 1px;
        padding-bottom: 1px;
        padding-top: 1px;
        border-color: rgb(255,150,60);
        background-color:qlineargradient(spread:pad, x1:0.5, y1:1, x2:0.5, y2:0, stop:0
rgba(190, 90, 50, 50), stop:1 rgba(250, 130, 40, 50));
}
QCheckBox::indicator:checked {
        border-radius:4px;
        border-style:solid;
        border-width:1px;
        border-color: rgb(246, 134, 86);
        background-color:rgb(246, 134, 86)
}
QCheckBox::indicator:unchecked {
        border-radius:4px;
        border-style:solid;
        border-width:1px;
        border-color:rgb(246, 134, 86);
        background-color:rgb(255,255,255);
}
QColorDialog {
        background-color:#f0f0f0;
}
QComboBox {
        color:rgb(81,72,65);
        background: #ffffff;
}
QComboBox:editable {
        background: #ffffff;
        color: rgb(81,72,65);
        selection-color:rgb(81,72,65);
        selection-background-color: #ffffff;
}
QComboBox QAbstractItemView {
        color:rgb(81,72,65);
        background: #ffffff;
        selection-color: #ffffff;
        selection-background-color: rgb(246, 134, 86);
}
QComboBox:!editable:on, QComboBox::drop-down:editable:on {
        color:  #1e1d23;
        background: #ffffff;
```

```
}
QDateTimeEdit {
	color:rgb(81,72,65);
	background-color: #ffffff;
}
QDateEdit {
	color:rgb(81,72,65);
	background-color: #ffffff;
}
QDialog {
	background-color:#f0f0f0;
}
QDoubleSpinBox {
	color:rgb(81,72,65);
	background-color: #ffffff;
}
QFontComboBox {
	color:rgb(81,72,65);
	background-color: #ffffff;
}
QLabel {
	color:rgb(17,17,17);
}
QLineEdit {
	background-color:rgb(255,255,255);
	selection-background-color:rgb(236,116,64);
	color:rgb(17,17,17);
}
QMenuBar {
	color:rgb(223,219,210);
	background-color:rgb(65,64,59);
}
QMenuBar::item {
	padding-top:4px;
	padding-left:4px;
	padding-right:4px;
	color:rgb(223,219,210);
	background-color:rgb(65,64,59);
}
QMenuBar::item:selected {
	color:rgb(255,255,255);
	padding-top:2px;
	padding-left:2px;
	padding-right:2px;
	border-top-width:2px;
```

```
        border-left-width:2px;
        border-right-width:2px;
        border-top-right-radius:4px;
        border-top-left-radius:4px;
        border-style:solid;
        background-color:rgb(65,64,59);
        border-top-color: rgb(47,47,44);
        border-right-color: qlineargradient(spread:pad, x1:0, y1:1, x2:1, y2:0, stop:0
rgba(90, 87, 78, 255), stop:1 rgba(47,47,44, 255));
        border-left-color:  qlineargradient(spread:pad, x1:1, y1:0, x2:0, y2:0, stop:0
rgba(90, 87, 78, 255), stop:1 rgba(47,47,44, 255));
}
QMenu {
        color:rgb(223,219,210);
        background-color:rgb(65,64,59);
}
QMenu::item {
        color:rgb(223,219,210);
        padding-left:20px;
        padding-top:4px;
        padding-bottom:4px;
        padding-right:10px;
}
QMenu::item:selected {
        color:rgb(255,255,255);
        background-color: qlineargradient(spread:pad, x1:0.5, y1:1, x2:0.5, y2:0,
stop:0 rgba(225, 108, 54, 255), stop:1 rgba(246, 134, 86, 255));
        border-style:solid;
        border-width:3px;
        padding-left:17px;
        padding-top:4px;
        padding-bottom:4px;
        padding-right:7px;
        border-bottom-color:qlineargradient(spread:pad, x1:0.5, y1:1, x2:0.5, y2:0,
stop:0 rgba(175,85,48,255), stop:1 rgba(236,114,67, 255));
        border-top-color:qlineargradient(spread:pad, x1:0.5, y1:1, x2:0.5, y2:0, stop:0
rgba(253,156,113,255), stop:1 rgba(205,90,46, 255));
        border-right-color:qlineargradient(spread:pad, x1:0, y1:0.5, x2:1, y2:0.5, stop:0
rgba(253,156,113,255), stop:1 rgba(205,90,46, 255));
        border-left-color:qlineargradient(spread:pad, x1:1, y1:0.5, x2:0, y2:0.5, stop:0
rgba(253,156,113,255), stop:1 rgba(205,90,46, 255));
}
QPlainTextEdit {
        border-width: 1px;
        border-style: solid;
```

```
        border-color:transparent;
        color:rgb(17,17,17);
        selection-background-color:rgb(236,116,64);
}
QProgressBar {
        text-align: center;
        color: rgb(0, 0, 0);
        border-width: 1px;
        border-radius: 10px;
        border-style: inset;
        border-color: rgb(150,150,150);
        background-color:rgb(221,221,219);
}
QProgressBar::chunk:horizontal {
        background-color: qlineargradient(spread:pad, x1:0.5, y1:1, x2:0.5, y2:0,
stop:0 rgba(225, 108, 54, 255), stop:1 rgba(246, 134, 86, 255));
        border-style: solid;
        border-radius:8px;
        border-width:1px;
        border-bottom-color:qlineargradient(spread:pad, x1:0.5, y1:1, x2:0.5, y2:0,
stop:0 rgba(175,85,48,255), stop:1 rgba(236,114,67, 255));
        border-top-color:qlineargradient(spread:pad, x1:0.5, y1:1, x2:0.5, y2:0, stop:0
rgba(253,156,113,255), stop:1 rgba(205,90,46, 255));
        border-right-color:qlineargradient(spread:pad, x1:0, y1:0.5, x2:1, y2:0.5, stop:0
rgba(253,156,113,255), stop:1 rgba(205,90,46, 255));
        border-left-color:qlineargradient(spread:pad, x1:1, y1:0.5, x2:0, y2:0.5, stop:0
rgba(253,156,113,255), stop:1 rgba(205,90,46, 255));
}
QPushButton{
        color:rgb(17,17,17);
        border-width: 1px;
        border-radius: 6px;
        border-bottom-color: rgb(150,150,150);
        border-right-color: rgb(165,165,165);
        border-left-color: rgb(165,165,165);
        border-top-color: rgb(180,180,180);
        border-style: solid;
        padding: 4px;
        background-color: qlineargradient(spread:pad, x1:0.5, y1:1, x2:0.5, y2:0,
stop:0 rgba(220, 220, 220, 255), stop:1 rgba(255, 255, 255, 255));
}
QPushButton:hover{
        color:rgb(17,17,17);
        border-width: 1px;
        border-radius:6px;
```

```
        border-top-color: rgb(255,150,60);
        border-right-color: qlineargradient(spread:pad, x1:0, y1:1, x2:1, y2:0, stop:0
rgba(200, 70, 20, 255), stop:1 rgba(255,150,60, 255));
        border-left-color:  qlineargradient(spread:pad, x1:1, y1:0, x2:0, y2:0, stop:0
rgba(200, 70, 20, 255), stop:1 rgba(255,150,60, 255));
        border-bottom-color: rgb(200,70,20);
        border-style: solid;
        padding: 2px;
        background-color: qlineargradient(spread:pad, x1:0.5, y1:1, x2:0.5, y2:0,
stop:0 rgba(220, 220, 220, 255), stop:1 rgba(255, 255, 255, 255));
}
QPushButton:default{
        color:rgb(17,17,17);
        border-width: 1px;
        border-radius:6px;
        border-top-color: rgb(255,150,60);
        border-right-color: qlineargradient(spread:pad, x1:0, y1:1, x2:1, y2:0, stop:0
rgba(200, 70, 20, 255), stop:1 rgba(255,150,60, 255));
        border-left-color:  qlineargradient(spread:pad, x1:1, y1:0, x2:0, y2:0, stop:0
rgba(200, 70, 20, 255), stop:1 rgba(255,150,60, 255));
        border-bottom-color: rgb(200,70,20);
        border-style: solid;
        padding: 2px;
        background-color: qlineargradient(spread:pad, x1:0.5, y1:1, x2:0.5, y2:0,
stop:0 rgba(220, 220, 220, 255), stop:1 rgba(255, 255, 255, 255));
}
QPushButton:pressed{
        color:rgb(17,17,17);
        border-width: 1px;
        border-radius: 6px;
        border-width: 1px;
        border-top-color: rgba(255,150,60,200);
        border-right-color: qlineargradient(spread:pad, x1:0, y1:1, x2:1, y2:0, stop:0
rgba(200, 70, 20, 255), stop:1 rgba(255,150,60, 200));
        border-left-color:  qlineargradient(spread:pad, x1:1, y1:0, x2:0, y2:0, stop:0
rgba(200, 70, 20, 255), stop:1 rgba(255,150,60, 200));
        border-bottom-color: rgba(200,70,20,200);
        border-style: solid;
        padding: 2px;
        background-color: qlineargradient(spread:pad, x1:0.5, y1:0, x2:0.5, y2:1,
stop:0 rgba(220, 220, 220, 255), stop:1 rgba(255, 255, 255, 255));
}
QPushButton:disabled{
        color:rgb(174,167,159);
        border-width: 1px;
```

```
        border-radius: 6px;
        background-color: qlineargradient(spread:pad, x1:0.5, y1:1, x2:0.5, y2:0,
stop:0 rgba(200, 200, 200, 255), stop:1 rgba(230, 230, 230, 255));
}
QRadioButton {
        padding: 1px;
}
QRadioButton::indicator:checked {
        height: 10px;
        width: 10px;
        border-style:solid;
        border-radius:5px;
        border-width: 1px;
        border-color: rgba(246, 134, 86, 255);
        color: #a9b7c6;
        background-color:rgba(246, 134, 86, 255);
}
QRadioButton::indicator:!checked {
        height: 10px;
        width: 10px;
        border-style:solid;
        border-radius:5px;
        border-width: 1px;
        border-color: rgb(246, 134, 86);
        color: #a9b7c6;
        background-color: transparent;
}
QScrollArea {
        color: #FFFFFF;
        background-color:#f0f0f0;
}
QSlider::groove {
        border-style: solid;
        border-width: 1px;
        border-color: rgb(207,207,207);
}
QSlider::groove:horizontal {
        height: 5px;
        background: rgb(246, 134, 86);
}
QSlider::groove:vertical {
        width: 5px;
        background: rgb(246, 134, 86);
}
QSlider::handle:horizontal {
```

```
        background: rgb(253,253,253);
        border-style: solid;
        border-width: 1px;
        border-color: rgb(207,207,207);
        width: 12px;
        margin: -5px 0;
        border-radius: 7px;
}
QSlider::handle:vertical {
        background: rgb(253,253,253);
        border-style: solid;
        border-width: 1px;
        border-color: rgb(207,207,207);
        height: 12px;
        margin: 0 -5px;
        border-radius: 7px;
}
QSlider::add-page:horizontal {
        background: white;
}

QSlider::add-page:vertical {
        background: white;
}
QSlider::sub-page:horizontal {
        background: rgb(246, 134, 86);
}
QSlider::sub-page:vertical {
        background: rgb(246, 134, 86);
}
QStatusBar {
        color:rgb(81,72,65);
}
QSpinBox {
        color:rgb(81,72,65);
        background-color: #ffffff;
}
QScrollBar:horizontal {
        max-height: 20px;
        border: 1px transparent grey;
        margin: 0px 20px 0px 20px;
}
QScrollBar::handle:horizontal {
        background: rgb(253,253,253);
        border-style: solid;
```

```
        border-width: 1px;
        border-color: rgb(207,207,207);
        border-radius: 7px;
        min-width: 25px;
}
QScrollBar::handle:horizontal:hover {
        background: rgb(253,253,253);
        border-style: solid;
        border-width: 1px;
        border-color: rgb(255,150,60);
        border-radius: 7px;
        min-width: 25px;
}
QScrollBar::add-line:horizontal {
        border: 1px solid;
        border-color: rgb(207,207,207);
        border-top-right-radius: 7px;
        border-top-left-radius: 7px;
        border-bottom-right-radius: 7px;
        background: rgb(255, 255, 255);
        width: 20px;
        subcontrol-position: right;
        subcontrol-origin: margin;
}
QScrollBar::add-line:horizontal:hover {
        border: 1px solid;
        border-top-right-radius: 7px;
        border-top-left-radius: 7px;
        border-bottom-right-radius: 7px;
        border-color: rgb(255,150,60);
        background: rgb(255, 255, 255);
        width: 20px;
        subcontrol-position: right;
        subcontrol-origin: margin;
}
QScrollBar::add-line:horizontal:pressed {
        border: 1px solid grey;
        border-top-left-radius: 7px;
        border-top-right-radius: 7px;
        border-bottom-right-radius: 7px;
        background: rgb(231,231,231);
        width: 20px;
        subcontrol-position: right;
        subcontrol-origin: margin;
}
```

```
QScrollBar::sub-line:horizontal {
        border: 1px solid;
        border-color: rgb(207,207,207);
        border-top-right-radius: 7px;
        border-top-left-radius: 7px;
        border-bottom-left-radius: 7px;
        background: rgb(255, 255, 255);
        width: 20px;
        subcontrol-position: left;
        subcontrol-origin: margin;
}
QScrollBar::sub-line:horizontal:hover {
        border: 1px solid;
        border-color: rgb(255,150,60);
        border-top-right-radius: 7px;
        border-top-left-radius: 7px;
        border-bottom-left-radius: 7px;
        background: rgb(255, 255, 255);
        width: 20px;
        subcontrol-position: left;
        subcontrol-origin: margin;
}
QScrollBar::sub-line:horizontal:pressed {
        border: 1px solid grey;
        border-top-right-radius: 7px;
        border-top-left-radius: 7px;
        border-bottom-left-radius: 7px;
        background: rgb(231,231,231);
        width: 20px;
        subcontrol-position: left;
        subcontrol-origin: margin;
}
QScrollBar::left-arrow:horizontal {
        border: 1px transparent grey;
        border-top-left-radius: 3px;
        border-bottom-left-radius: 3px;
        width: 6px;
        height: 6px;
        background: rgb(230,230,230);
}
QScrollBar::right-arrow:horizontal {
        border: 1px transparent grey;
        border-top-right-radius: 3px;
        border-bottom-right-radius: 3px;
        width: 6px;
```

```
        height: 6px;
        background: rgb(230,230,230);
}
QScrollBar::add-page:horizontal, QScrollBar::sub-page:horizontal {
        background: none;
}
QScrollBar:vertical {
        max-width: 20px;
        border: 1px transparent grey;
        margin: 20px 0px 20px 0px;
}
QScrollBar::add-line:vertical {
        border: 1px solid;
        border-color: rgb(207,207,207);
        border-bottom-right-radius: 7px;
        border-bottom-left-radius: 7px;
        border-top-left-radius: 7px;
        background: rgb(255, 255, 255);
        height: 20px;
        subcontrol-position: bottom;
        subcontrol-origin: margin;
}
QScrollBar::add-line:vertical:hover {
        border: 1px solid;
        border-color: rgb(255,150,60);
        border-bottom-right-radius: 7px;
        border-bottom-left-radius: 7px;
        border-top-left-radius: 7px;
        background: rgb(255, 255, 255);
        height: 20px;
        subcontrol-position: bottom;
        subcontrol-origin: margin;
}
QScrollBar::add-line:vertical:pressed {
        border: 1px solid grey;
        border-bottom-left-radius: 7px;
        border-bottom-right-radius: 7px;
        border-top-left-radius: 7px;
        background: rgb(231,231,231);
        height: 20px;
        subcontrol-position: bottom;
        subcontrol-origin: margin;
}
QScrollBar::sub-line:vertical {
        border: 1px solid;
```

```
        border-color: rgb(207,207,207);
        border-top-right-radius: 7px;
        border-top-left-radius: 7px;
        border-bottom-left-radius: 7px;
        background: rgb(255, 255, 255);
        height: 20px;
        subcontrol-position: top;
        subcontrol-origin: margin;
}
QScrollBar::sub-line:vertical:hover {
        border: 1px solid;
        border-color: rgb(255,150,60);
        border-top-right-radius: 7px;
        border-top-left-radius: 7px;
        border-bottom-left-radius: 7px;
        background: rgb(255, 255, 255);
        height: 20px;
        subcontrol-position: top;
        subcontrol-origin: margin;
}
QScrollBar::sub-line:vertical:pressed {
        border: 1px solid grey;
        border-top-left-radius: 7px;
        border-top-right-radius: 7px;
        background: rgb(231,231,231);
        height: 20px;
        subcontrol-position: top;
        subcontrol-origin: margin;
}
QScrollBar::handle:vertical {
        background: rgb(253,253,253);
        border-style: solid;
        border-width: 1px;
        border-color: rgb(207,207,207);
        border-radius: 7px;
        min-height: 25px;
}
QScrollBar::handle:vertical:hover {
        background: rgb(253,253,253);
        border-style: solid;
        border-width: 1px;
        border-color: rgb(255,150,60);
        border-radius: 7px;
        min-height: 25px;
}
```

```
QScrollBar::up-arrow:vertical {
	border: 1px transparent grey;
	border-top-left-radius: 3px;
	border-top-right-radius: 3px;
	width: 6px;
	height: 6px;
	background: rgb(230,230,230);
}
QScrollBar::down-arrow:vertical {
	border: 1px transparent grey;
	border-bottom-left-radius: 3px;
	border-bottom-right-radius: 3px;
	width: 6px;
	height: 6px;
	background: rgb(230,230,230);
}
QScrollBar::add-page:vertical, QScrollBar::sub-page:vertical {
	background: none;
}
QTabWidget {
	color:rgb(0,0,0);
	background-color:rgb(247,246,246);
}
QTabWidget::pane {
	border-color: rgb(180,180,180);
	background-color:rgb(247,246,246);
	border-style: solid;
	border-width: 1px;
	border-radius: 6px;
}
QTabBar::tab {
	padding-left:4px;
	padding-right:4px;
	padding-bottom:2px;
	padding-top:2px;
	color:rgb(81,72,65);
	background-color: qlineargradient(spread:pad, x1:0.5, y1:1, x2:0.5, y2:0,
stop:0 rgba(221,218,217,255), stop:1 rgba(240,239,238,255));
	border-style: solid;
	border-width: 1px;
	border-top-right-radius:4px;
	border-top-left-radius:4px;
	border-top-color: rgb(180,180,180);
	border-left-color: rgb(180,180,180);
	border-right-color: rgb(180,180,180);
```

```
        border-bottom-color: transparent;
}
QTabBar::tab:selected, QTabBar::tab:last:selected, QTabBar::tab:hover {
        background-color:rgb(247,246,246);
        margin-left: 0px;
        margin-right: 1px;
}
QTabBar::tab:!selected {
        margin-top: 1px;
        margin-right: 1px;
}
QTextEdit {
        border-width: 1px;
        border-style: solid;
        border-color:transparent;
        color:rgb(17,17,17);
        selection-background-color:rgb(236,116,64);
}
QTimeEdit {
        color:rgb(81,72,65);
        background-color: #ffffff;
}

QToolBox {
        color:rgb(81,72,65);
        background-color: #ffffff;
}
QToolBox::tab {
        color:rgb(81,72,65);
        background-color: #ffffff;
}
QToolBox::tab:selected {
        color:rgb(81,72,65);
        background-color: #ffffff;
}
```

Листинг П.3.2. Код, который необходимо использовать для подключения стиля

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

Листинг П.3.3. Содержание файла main.cpp

```
#include "rootwindow.h"
#include <QApplication>//Класс QApplication руководит управляющей логикой GUI и основными настройками.
#include <QFile>
#include <QTextStream>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    /*При снятии этих комментариев стиль Ubuntu будет утсановлен по умолчанию для прораммы
     *
     * QFile styleFile(qApp->applicationDirPath()+"/style/Ubuntu.qss");
       if(styleFile.open(QIODevice::ReadOnly))
       {
          QTextStream *textStream = new QTextStream(&styleFile);
          QString styleSheet = textStream->readAll();
          styleFile.close();
          a.setStyleSheet(styleSheet);
       }*/
    RootWindow w;
    w.show();

    return a.exec();
}
```

Листинг П.3.4. Содержание файла rootwindow.h

```
/*Директива #ifndef проверяет, определено ли имя ROOTWINDOW_H, если нет, то управление передаётся директиве #define и определяется интерфейс класса.
 * Если же имя ROOTWINDOW_H уже определено, управление передается директиве #endif.
 *  Таким образом, исключается возможность многократного определения класса ROOTWINDOW*/
#ifndef ROOTWINDOW_H
#define ROOTWINDOW_H

/*подключаем необходимые заголовочные файлы*/
#include <QApplication>
#include <QFile>
#include <QMainWindow> //заголовок базового класса главного окна QMainWindow
#include <QStyleFactory>
#include "firstpartwindow.h" //заголовочный файл окна для первой части работы
```

```
#include "secondpartwindow.h" //заголовочный файл окна для второй части
работы
#include"aboutdialog.h"

/*Определяем пространство имен, в котором будет существовать наш класс
RootWindow*/
namespace Ui {
class RootWindow;
}

/*объявление класса, описывающего методы, используемые в первой части
работы*/
class RootWindow : public QMainWindow //объявленный класс является
наследником класса QMainWindow
{
   //Класс QObject — это базовый класс для всех объектов Qt.
   Q_OBJECT //макрос Q_OBJECT является обязательным для любого объекта,
реализующим сигналы, слоты и свойства.

public:
   explicit RootWindow(QWidget *parent = nullptr); //конструктор
   ~RootWindow(); //деструктор

private:
   Ui::RootWindow *ui;
   //описание слотов
   //Слот вызывается тогда, когда подключенный к нему сигнал был получен.
В данном случае обрабатывается сигнал clicked()
public slots:
   void FirstPartButtonClickHandler();
   void SecondPartButtonClickHandler();
   void AboutDialogButtonClickHandler()
   {
      AboutDialog *aboutDialog = new AboutDialog(this);
    aboutDialog->show();
   }

   //слот обработки события м=изменения стиля приложения
   void slotChangeStyle(const QString& str)//слот выбора стиля приложения
   {

      if(str=="Ubuntu"){
         QString strCSS;
         QFile stylefile(qApp->applicationDirPath()+"/style/Ubuntu.qss");
         if (stylefile.open(QFile::ReadOnly)){
```

```
            QString strCSS=QLatin1String(stylefile.readAll());
            stylefile.close();
            qApp->setStyleSheet(strCSS);
            }
        }else if (str=="MyStyle") {
            QString strCSS;
            QFile stylefile(qApp->applicationDirPath()+"/style/MyStyle.qss");
            if (stylefile.open(QFile::ReadOnly)){
            QString strCSS=QLatin1String(stylefile.readAll());
            stylefile.close();
            qApp->setStyleSheet(strCSS);
            }
} else {
            qApp->setStyleSheet(styleSheet());
        QStyle* pstyle = QStyleFactory::create(str);
        QApplication::setStyle(pstyle);
            }
    }
};

#endif // ROOTWINDOW_H
```

Листинг П.3.5. Содержание файла rootwindow.cpp

```
#include "rootwindow.h"
#include "ui_rootwindow.h"

#include <QComboBox>
#include <QFile>
#include <QStyleFactory>
#include <QTextStream>
#include <QMenuBar>
#include <QAction>
//определяем конструктор класса
RootWindow::RootWindow(QWidget *parent) :
    QMainWindow(parent),//вызов конструктора базового класса с аргументом
в качестве родителя
    ui(new Ui::RootWindow)//инициализация объекта класса интерфейса
{
    ui->setupUi(this);//создание интерфейса

    QMenu *file = menuBar()->addMenu("Файл"); //Создаем меню бар и
помещаяем в него первый пункт меню file
   QAction *getInfo = new QAction(tr("О программе"), this); // Создаем экшн для
нашего меню файл с пунктом «О программе», также обязательно в качестве
родителя оператором this указываем наше главное окно
```

```
    getInfo->setShortcut(tr("Ctrl+I")); // Создаем горячую клавишу для нашего
экшена
    connect(getInfo, SIGNAL(triggered()), this,
SLOT(AboutDialogButtonClickHandler()));
    file->addAction(getInfo); // Привязка экшена к меню File

   /*ДОБАВЛЕНИЕ СПИСКА ВЫБОРА СТИЛЯ ПРИЛОЖЕНИЯ*/
   QComboBox* cbo=new QComboBox;
   cbo->addItems(QStringList() << QStyleFactory::keys()<<"Ubuntu"<<"MyStyle");
   connect(cbo,
   SIGNAL(activated(const QString&)),
   SLOT(slotChangeStyle(const QString&))
   ) ;
    ui->verticalLayout->addWidget(cbo);
    /*_END_ДОБАВЛЕНИЕ СПИСКА ВЫБОРА СТИЛЯ ПРИЛОЖЕНИЯ*/
   //свяжем сигналы со слотами
   QObject::connect(ui->pushButton,
        SIGNAL(clicked()),this, SLOT(FirstPartButtonClickHandler()));
   QObject::connect(ui->pushButton_2,
        SIGNAL(clicked()),this, SLOT(SecondPartButtonClickHandler()));
}

RootWindow::~RootWindow() //определяем деструктор
{
   delete ui;
}
/* определяем обработчики событий нажатий на кнопки (слоты)*/
void RootWindow::FirstPartButtonClickHandler()
{   //создаем указатель на окно FirstPartWindow
   FirstPartWindow *frm1= new FirstPartWindow();
   frm1->show();//вызываем окно на экран
}
void RootWindow::SecondPartButtonClickHandler()
{   //создаем указатель на окно SecondPartWindow
   SecondPartWindow *frm2= new SecondPartWindow();
   frm2->show();//вызываем окно на экран
}
```

Листинг П.3.6. Содержание файла firstpartwindow.h

```
/*Директива #ifndef проверяет, определено ли имя FIRSTPARTWINDOW_H,
если нет, то управление передаётся директиве #define и определяется
интерфейс класса.
 * Если же имя FIRSTPARTWINDOW_H уже определено, управление
передается директиве #endif.
```

```
 *  Таким образом, исключается возможность многократного определения
класса FIRSTPARTWINDOW*/
#ifndef FIRSTPARTWINDOW_H
#define FIRSTPARTWINDOW_H

#include <QMainWindow>//заголовок базового класса главного окна
QMainWindow
#include <QMessageBox>
/*подключение библиотеки математических функций*/
#include "math.h"

/*Определяем пространство имен, в котором будет существовать наш класс
FirstPartWindow*/
namespace Ui {
class FirstPartWindow;
}
/*объявление класса, описывающего методы, используемые в первой части
работы*/
class FirstPartWindow : public QMainWindow //объявленный нами класс
является наследником класса QMainWindow
{
    //Класс QObject — это базовый класс для всех объектов Qt.

    Q_OBJECT //макрос Q_OBJECT является обязательным для любого объекта,
реализующим сигналы, слоты и свойства.

public:
    explicit FirstPartWindow(QWidget *parent = nullptr); //описание конструктора
класса
    ~FirstPartWindow(); //описание деструктора класса

//описание приватных членов класса
private:
    Ui::FirstPartWindow *ui;

    //описание слотов
    //Слот вызывается тогда, когда подключенный к нему сигнал был получен
public slots:
    void MyEventHandler();
    void CalcHandler();

    //описание сигналов
    //Сигналы выпускаются объектом, когда его внутреннее состояние
изменилось.
```

```
signals:
    void MySignal(QString);
    void CalcSignal();
};

#endif // FIRSTPARTWINDOW_H
```

Листинг П.3.7. Содержание файла firstpartwindow.cpp

```
#include "firstpartwindow.h"
#include "ui_firstpartwindow.h"

//определяем конструктор класса FirstPartWindow
FirstPartWindow::FirstPartWindow(QWidget *parent) :
    QMainWindow(parent), //вызов конструктора базового класса с аргументом в
качестве родителя
    ui(new Ui::FirstPartWindow) //инициализация объекта класса интерфейса
{
    ui->setupUi(this); //создание интерфейса

    // изменение свойства text объекта labelResult
    ui->labelResult->setText("Результат:");

    // свяжем сигнал со слотом
    QObject::connect(ui->pushButton_Copy,
          SIGNAL(clicked()),this, SLOT(MyEventHandler()));
        QObject::connect(this, SIGNAL(MySignal(QString)),
          ui->lineEdit_2, SLOT(setText(QString)));
        QObject::connect(this, SIGNAL(MySignal(QString)),
          ui->Result, SLOT(setText(QString)));
        QObject::connect(ui->pushButtonCalc,
                    SIGNAL(clicked()),this,SLOT(CalcHandler()));
}

FirstPartWindow::~FirstPartWindow() //определяем деструктор
{
    delete ui;
}
// тело слота (т. е. функции обработки сигнала)
void FirstPartWindow::MyEventHandler()
{
emit MySignal(ui->lineEdit_1->text());
}
// Расчет корня из суммы чисел
void FirstPartWindow::CalcHandler()
{
```

```
// Объявление трех переменных типа float
float A;
float B;
float S;

// Объявление двух переменных логического типа
bool ok1, ok2;

// Объявление переменной типа QString
QString Res;

// Считывание данных введенных в поля редактирования.
// Производим преобразование строки в число
// логическая переменная ok1 и ok2 сигнализирует о
// успешности выполнении преобразования
A=ui->lineEdit_1->text().toFloat(&ok1);
B=ui->lineEdit_2->text().toFloat(&ok2);

// Индикация корректности ввода данных в поле 1
if (ok1)
{
   // устанавливаем стиль для элемента
   ui->lineEdit_1->setStyleSheet("background-color: white");
}
else
{
   ui->lineEdit_1->setStyleSheet("background-color: red");
}

// Индикация корректности ввода данных в поле 2
if (ok2)
{
   ui->lineEdit_2->setStyleSheet("background-color: white");
}
else
{
   ui->lineEdit_2->setStyleSheet("background-color: red");
      }
      // Корень из суммы
      S=sqrt(A+B);
      // Преобразование числа в строку
      Res.setNum(S);
      // Вывод на экран
      ui->Result->setText(Res);
 }
```

Листинг П.3.8. Содержание файла secondpartwindow.h

```
/*Директива #ifndef проверяет, определено ли имя SECONDPARTWINDOW_H,
если нет, то управление передаётся директиве #define и определяется
интерфейс класса.
 * Если же имя SECONDPARTWINDOW_H уже определено, управление
передается директиве #endif.
 *  Таким образом, исключается возможность многократного определения
класса SECONDPARTWINDOW*/
#ifndef SECONDPARTWINDOW_H
#define SECONDPARTWINDOW_H

/*подключаем необходимые заголовочные файлы*/
#include <QMainWindow> //заголовочный файл базового класса главного окна
QMainWindow
#include <QLineEdit> //заголовочный файл класса QLineEdit, реализующего
виджет LineEdit
#include <QMessageBox>//заголовочный файл класса QMessageBox,
реализующего виджет MessageBox
#include "calculate.h"

/*Определяем пространство имен, в котором будет существовать наш класс
SecondPartWindow*/
namespace Ui {
class SecondPartWindow;
}
/*объявление класса*/
class SecondPartWindow : public QMainWindow //объявленный класс является
наследником класса QMainWindow
{
   Q_OBJECT //макрос Q_OBJECT является обязательным для любого объекта,
реализующим сигналы, слоты и свойства.

public:
   explicit SecondPartWindow(QWidget *parent = nullptr); //конструктор
   ~SecondPartWindow();//деструктор

private:
   Ui::SecondPartWindow *ui;

   bool valid,valid2,valid3; //переменные для проверки валидность заполнения
lineEdit
   //описание слотов
private slots:
   void clearEditLines(); //метод, выполняемый при получении сигнала от
кнопки «Очистить»
```

```
    void copyItem(); //метод, выполняемый при получении сигнала от кнопки
«Копировать»
    void getCalcResult(); //метод, выполняемый при получении сигнала от
кнопки «Результат»

protected slots:
    bool eventFilter(QObject *obj, QEvent *ev); //метод для перехвата
определенных событий

 //описание сигналов
signals:
    void clearButtonSignal();
    void copyButtonSignal(QString);
    void getCalcButtonSignal();
};

#endif // SECONDPARTWINDOW_H
```

Листинг П.3.9. Содержание файла secondpartwindow.cpp

```
#include "secondpartwindow.h"
#include "ui_secondpartwindow.h"

SecondPartWindow::SecondPartWindow(QWidget *parent) :
    QMainWindow(parent),//вызов конструктора базового класса с аргументом в
качестве родителя
    ui(new Ui::SecondPartWindow)//инициализация объекта класса интерфейса
{
    ui->setupUi(this);//создание интерфейса

    /*устанавливаем фильтры на события от текстовых полей*/
    ui->lineEdit2->installEventFilter(this);
    ui->lineEdit_1->installEventFilter(this);
    ui->lambdaStr->installEventFilter(this);
    ui->groupBox1->hide();
    /*свяжем сигналы со слотами*/
    QObject::connect(ui->resultButton, SIGNAL(clicked()), this,
SLOT(getCalcResult()));  //соединение сигнала кнопки resultButton со слотом
calc()
    QObject::connect(ui->copyButton, SIGNAL(clicked()), this, SLOT(copyItem()));
//соединение сигнала кнопки copyButton со слотом copy()
    QObject::connect(this, SIGNAL(copyButtonSignal(QString)),
      ui->lineEdit2, SLOT(setText(QString)));
}
/*определение метода eventFilter*/
bool SecondPartWindow::eventFilter(QObject *obj, QEvent *ev){
```

```
    //отлавливаем события потери фокуса на lineEdit
   if ((obj == ui->lineEdit2) && ev->type() == QEvent::FocusOut)
   {
      //если введенные данные не число
      ui->lineEdit2->text().toInt(&valid2);
      if(!valid2){ui->lineEdit2->setStyleSheet("background-color: red");
//устанавливаем стиль для элемента
              ui->error_label_2->setText("Введите целое число в диапазоне от 0 до 9
"); //выводим подсказку
             }
      //иначе
      else {ui->lineEdit2->setStyleSheet(styleSheet()); //возвращаем исходный стиль
           ui->error_label_2->clear(); //убираем подсказку
          }
   }
   if (obj == ui->lineEdit_1 && ev->type() == QEvent::FocusOut)
       {
      ui->lineEdit_1->text().toInt(&valid);
      if(!valid){ui->lineEdit_1->setStyleSheet("background-color: red");
             ui->error_label->setText("Введите целое число в диапазоне от 0 до 9 ");
             }
      else {ui->lineEdit_1->setStyleSheet(styleSheet());
           ui->error_label->clear();
          }
       }
   if (obj == ui->lambdaStr && ev->type() == QEvent::FocusOut)
       {
      ui->lambdaStr->text().toDouble(&valid3);
      if(!valid3){ui->lambdaStr->setStyleSheet("background-color: red");
              ui->error_label_3->setText("Введите рациональное число ");
             }
      else {ui->lambdaStr->setStyleSheet(styleSheet());
          ui->error_label_3->clear();
           }
       }

   return false;
}
/*определяем деструктор*/
SecondPartWindow::~SecondPartWindow()
{
    delete ui;
}
```

```
void SecondPartWindow::copyItem()  //определяем метод (слот)
{
   emit copyButtonSignal(ui->lineEdit_1->text());
}

void SecondPartWindow::getCalcResult() //определяем метод расчета
задания(слот)
{
   double lambda,             //длина волны излучения источника
          nominal_length,
          maxLenghtPD;

  bool ok1, ok2, ok3;              //флаги проверки корректности ввода данных
      int  gradebook_num,          //переменная для хранения введенной
предпоследней цифры зачетной книжки
       gradebook_lastNum;          //переменная для хранения введенной последней
цифры зачетной книжки
       if ((!valid)||(!valid2)||(!valid3))             //проверка на корректность введенных
данных
          {//Выводим сообщение на экран
            QMessageBox::information(this,"Information", «Одно или несколько
полей заполнены не верно! Повторите ввод»);
          return;
          }
       //если ввод был корректным, записываем введенные данные в переменные
       gradebook_num=ui->lineEdit_1->text().toInt(&ok1);
          gradebook_lastNum=ui->lineEdit2->text().toInt(&ok2);
          lambda=ui->lambdaStr->text().toDouble(&ok3);
         Calculate solver; //создаем объект класса, производящего расчет
                  solver.spotFiberType(gradebook_num); //вызываем метод
определения типа волокна
                  ui->groupBox1->setVisible(1);
         ui->fiberLabel->setText(solver.get_fiberType());//устанавливаем
полученный тип волокна в качестве текста метки на форме
         /*вызываем метод, устанавливающий максимальное значение
коэффициента затухания и величину хроматической дисперсии,
           *в зависимости от параметра*/
         solver.InstallParam(lambda);
         solver.spotTransSpeed(gradebook_lastNum);//вызываем метод определения
скорости передачи
         nominal_length= solver.Raschet_nominal_length(); //рассчитываем
номинальную длину
          maxLenghtPD=solver.Raschet_maxLenghtPD(lambda);//рассчитываем
максимальную длину участка регенерации по дисперсии
```

```
        ui->transSpeedLabel->setText("Скорость передачи - " +
QString::number(solver.get_transSpeed()) + " Мб/с");   //вывод значения скорости
передачи данных
        ui->nomRes->setText(QString::number(nominal_length));       //перевод
результатов расчетов из численного в строковый вид
            ui->maxRes->setText(QString::number(maxLenghtPD));
}
```

Листинг П.3.10. Содержание файла aboutdialog.h

```
#ifndef ABOUTDIALOG_H
#define ABOUTDIALOG_H
#include <QDialog>

namespace Ui {
class AboutDialog;
}
class AboutDialog : public QDialog
{
   Q_OBJECT

public:
   explicit AboutDialog(QWidget *parent = nullptr);
   ~AboutDialog();

private:
   Ui::AboutDialog *ui;
};

#endif // ABOUTDIALOG_H
```

Листинг П.3.11. Содержание файла aboutdialog.cpp

```
#include "aboutdialog.h"
#include "ui_aboutdialog.h"

AboutDialog::AboutDialog(QWidget *parent) :
   QDialog(parent),
   ui(new Ui::AboutDialog)
{
   ui->setupUi(this);
}

AboutDialog::~AboutDialog()
{
   delete ui;
}
```

Листинг П.3.12. Содержание файла calculate.h

```
/*Директива #ifndef проверяет, определено ли имя CALCULATE_H, если нет,
 то управление передается директиве #define и определяется интерфейс класса.
 * Если же имя CALCULATE_H уже определено, управление передается
директиве #endif.
 *  Таким образом, исключается возможность многократного определения
класса CALCULATE*/
#ifndef CALCULATE_H
#define CALCULATE_H
/*подключаем необходимые заголовки*/
#include <QString>
#include <math.h>          // заголовочный файл  для выполнения простых
математических операций

/*объявление нашего класса, описывающего методы, используемые для расчета
поставленной задачи*/
class Calculate
{
private:
   //определяем константы
   static const long long Apc = 1, //суммарное затухание всех разъемных
соединений
            Aeza = 3,      //эксплуатационные запасы аппаратуры
            Aezk = 3,      //эксплуатационные запасы кабеля
            lstr = 5000,   //строительная длина кабеля
            W = 40,        //энергетический потенциал
            c = 300000000,      //скорость света в вакууме
            delV = 50000000000;  //ширина спектра излучения

//объявляем переменные
    int transSpeed;         //скорость передачи, Мб/с
    double  alphMax,        //максимальное значение коэффициента затухания
          tau0,           //ширина оптического импульса на выходе передатчика
          Dx,             //величина хроматической дисперсии
          Ansmax,         //максимальная величина затухания одного неразъемного
соединения
          delA;           //погрешность измерения затухания
    QString fiberType;     //тип волокна

public:
   explicit Calculate(){   //конструктор
       //инициализируем не целочисленные константы
       Ansmax = (0.2);
       delA=(0.2);
    }
```

```
    ~Calculate(){}        //деструктор

    //определяем функцию расчета максимальной длины
    double Raschet_maxLenghtPD(double lambda)
    {
        return(2*3.14*c*pow(this->tau0, 2))/(pow(lambda, 2)*this-
>Dx*sqrt(1+4*pow(3.14, 2)*pow(delV, 2)*pow(this->tau0, 2)));
    }
    //определяем функцию расчета номинальной длины
    double Raschet_nominal_length()
    {
        return (W - Apc + this->Ansmax - Aeza - Aezk - this->delA)/(this->alphMax +
(this->Ansmax/lstr));
    }
public:
    //setters
    void set_transSpeed(int transSpeed)
    {this->transSpeed=transSpeed;}

    void set_alphMax(double alphMax)
    {this->alphMax=alphMax;}

    void set_tau0(double tau0)
    {this->tau0=tau0;}

    void set_Dx(double Dx)
    {this->Dx=Dx;}

    void set_fiberType(QString type)
    {
        this->fiberType=type;
    }

    //определяем метод, устанавливающий тип волокна, в зависимости от
параметра
void spotFiberType(int gradebook_num)
{
    if(gradebook_num & 1) {set_fiberType("Тип волокна - G.653 (DSF)");}//1357

     set_fiberType("Тип волокна - G.652 (SSF)");//0246
}
 //определяем метод, устанавливающий скорость передачи и ширину
оптического импульса, в зависимости от параметра
void spotTransSpeed(int gradebook_lastNum)
{
```

```
    switch(gradebook_lastNum)
    {
       case 1:
    case 5:{ set_tau0(0.386); set_transSpeed(565);break;}
       case 2:
    case 7:{ set_tau0(1.54); set_transSpeed(140);break;}
    case 3:{ set_tau0(0.096); set_transSpeed(2500);break;}
    case 4:{ set_tau0(1.54); set_transSpeed(155);break;}
       case 6:
    case 8:{set_tau0(0.386); set_transSpeed(620);break;}
    default: {set_tau0(1.54); set_transSpeed(34);break;}
    }
}
/*определяем метод, устанавливающий максимальное значение коэффициента
затухания и величину хроматической дисперсии, в зависимости от параметра*/
void InstallParam(double lambda)
{
    if(this->fiberType=="Тип волокна - G.653 (DSF)")
    {
       if (lambda == 1310.0)
       {
          //вызываем сеттеры
           set_alphMax(1.0);
           set_Dx(17.5);
       }
       else { set_alphMax(0.22);
              set_Dx(3.5);
            }
    }
    else
    {
       if (lambda == 1310.0)              //данные из таблицы характеристик
оптических волокон
                {
                   set_alphMax(0.34);
                   set_Dx(3.5);
                }
                else {set_alphMax(0.22);
                      set_Dx(17.5);
                     }
    }
}
//getters
int get_transSpeed()
{return this->transSpeed;}
```

```
double get_alphMax()
{return this->alphMax;}

double get_tau0()
{return this->tau0;}

double get_Dx()
{return this->Dx;}

QString get_fiberType()
{
   return this->fiberType;
}
//QString getAppConst(){
//    QString Const[9]=
//    {"Суммарное затухание всех разъёмных соединений: Apc = 1;",
//     "Эксплуатационные запасы аппаратуры: Aeza = 3;",
//     "Эксплуатационные запасы кабеля: Aezk = 3;",
//     "Строительная длина кабеля: lstr = 5000;",
//     "Энергетический потенциал: 40;",
//     "Скорость света в вакууме: c = 300000000;",
//     "Ширина спектра излучения: delV = 50000000000;",
//     "Максимальная величина затухания одного неразъёмного соединения:
Ansmax = 0.2;",
//     "Погрешность измерения затухания: delA = 0.2;"
//    };
//    return Const;
//}

};
#endif // CALCULATE_H
```

Листинг П.4.1. Содержание файла "main.cpp"

```
#include <QCoreApplication>
#include "examplethreads.h"

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

   ExampleThreads threadA("thread A");
   ExampleThreads threadB("thread B");
   ExampleThreads threadC("thread C");

   threadA.start();    // Запускаем потоки
```

```
    threadB.start();    // и наблюдаем их параллельную работу
    threadC.start();    // в выводе qDebug

    return a.exec();
}
```

Листинг П.4.2. Содержание файла examplethreads.h

```
#ifndef EXAMPLETHREADS_H
#define EXAMPLETHREADS_H
#include <QThread>

class ExampleThreads : public QThread
{
public:
    explicit ExampleThreads(QString threadName);

    // Переопределяем метод run(), в котором будет
    // располагаться выполняемый код
    void run();
private:
    QString name;   // Имя потока
};

#endif // EXAMPLETHREADS_H
```

Листинг П.4.3. Содержание файла examplethreads.cpp

```
#include "examplethreads.h"
#include <QDebug>

ExampleThreads::ExampleThreads(QString threadName) :
    name(threadName)
{
}

void ExampleThreads::run()
{
    for (int i = 0; i <= 100; i++ ) {
        qDebug() << name << " " << i;
    }
}
```

Листинг П.4.4. Содержание файла main.cpp

```
#include "mainwindow.h"
#include <QApplication>
int main(int argc, char *argv[])
```

```
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

Листинг П.4.5. Содержание файла exampleobject.h

```
/*****************************************************************
Для начала рассмотрим содержимое заголовочного файла объекта. В данном
файле объявлено три свойства Q_PROPERTY :
running — это переменная, с помощью которой будет осуществляться
управление циклом в методе run() и, соответственно, будет влиять на
завершение выполнения полезной работы объекта.
message — это строка, которая будет передаваться для вывода в qDebug() из
главного окна приложения.
message_2 — это вторая строка, которая также будет отображаться в qDebug(),
но при этом будет использоваться и для передачи во второй поток.
Также в заголовочном файле объявлены слот-метод run(), сигнал finished() и
переменная int count */

#ifndef EXAMPLEOBJECT_H
#define EXAMPLEOBJECT_H
#include <QObject>

class ExampleObject : public QObject
{
    Q_OBJECT
    // Свойство, управляющее работой потока
    Q_PROPERTY(bool running READ running WRITE setRunning NOTIFY
runningChanged)
    // Первое сообщение в объекте
    Q_PROPERTY(QString message READ message WRITE setMessage NOTIFY
messageChanged)
    // Второе сообщение, которое будем передавать через сигнал/слот во второй
объект
    Q_PROPERTY(QString message_2 READ message_2 WRITE setMessage_2
NOTIFY message_2Changed)

    bool m_running;
    QString m_message;
    QString m_message_2;
    int count;  // Счетчик, по которому будем ориентироваться на то,
            // что потоки выполняются и работают
```

```
public:
    explicit ExampleObject(QObject *parent = 0);
    bool running() const;
    QString message() const;
    QString message_2() const;

signals:
    void finished();    // Сигнал, по которому будем завершать поток, после
завершения метода run
    void runningChanged(bool running);
    void messageChanged(QString message);
    void message_2Changed(QString message_2);
    void sendMessage(QString message);

public slots:
    void run(); // Метод с полезной нагрузкой, который может выполняться в
цикле
    void setRunning(bool running);
    void setMessage(QString message);
    void setMessage_2(QString message_2);
};

#endif // EXAMPLEOBJECT_H
```

Листинг П.4.6. Содержание файла exampleobject.cpp

```
/*****************************************************************
Весь наш интерес сводится к методу run(), в котором в цикле while будет
инкрементироваться счетчик count и будет выводиться информация с
сообщением m_message, m_message_2 и данным счетчиком до тех пор, пока
переменная m_running не будет выставлена в значение false.
 По выходу из цикла при завершении выполнения метода run() будет выпущен
сигнал finished(), по которому завершится поток, в котором будет находиться
данный объект. */

#include "exampleobject.h"
#include <QDebug>

ExampleObject::ExampleObject(QObject *parent) :
    QObject(parent),
    m_message(""),
    m_message_2("")
{
}
```

```
bool ExampleObject::running() const
{
    return m_running;
}
QString ExampleObject::message() const
{
    return m_message;
}

QString ExampleObject::message_2() const
{
    return m_message_2;
}

// Самый важный метод, в котором будет выполняться «полезная» работа
объекта
void ExampleObject::run()
{
    count = 0;
    // Переменная m_running отвечает за работу объекта в потоке.
    // При значении false работа завершается
    while (m_running)
    {
        count++;
        emit sendMessage(m_message); // Высылаем данные, которые будут
передаваться в другой поток
        qDebug() << m_message << " " << m_message_2 << " " << count;
    }
    emit finished();
}

void ExampleObject::setRunning(bool running)
{
    if (m_running == running)
        return;

    m_running = running;
    emit runningChanged(running);
}

void ExampleObject::setMessage(QString message)
{
    if (m_message == message)
        return;
```

```
    m_message = message;
    emit messageChanged(message);
}

void ExampleObject::setMessage_2(QString message_2)
{
    if (m_message_2 == message_2)
        return;

    m_message_2 = message_2;
    emit message_2Changed(message_2);
}
```

Листинг П.4.7. Содержание файла mainwindow.h

```
/*****************************************************************
В заголовочном файле главного окна приложения объявлены слоты для
обработки нажатий кнопок для записи сообщений в объекты, которые будут
выполняться в потоках, а также слоты для запуска и остановки потоков.
Помимо этого, объявим два объекта класса ExampleObject и два объекта класса
QThread. */

#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
#include <QThread>
#include "exampleobject.h"

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
    void on_write_1_clicked();  // Слот для записи данных из lineEdit_1 в первый
объект первого потока
    void on_write_2_clicked();  // Слот для записи данных из lineEdit_2 во второй
объект второго потока
    void on_start_clicked();    // Слот для запуска потоков
```

```
    void on_stop_clicked();     // Слот для остановки потоков

private:
    Ui::MainWindow *ui;
    QThread thread_1;       // Первый поток
    QThread thread_2;       // Второй поток
    ExampleObject exampleObject_1;  // первый объект, который будет работать в
первом потоке
    ExampleObject exampleObject_2;  // второй объект, который будет работать во
втором потоке
};

#endif // MAINWINDOW_H
```

Листинг П.4.8. Содержание файла mainwindow.cpp

```
/*****************************************************************
Теперь объединим весь предыдущий код в рабочее приложение в исходном
коде главного окна приложения.
 В конструкторе данного класса, как уже было сказано в самом начале,
необходимо подключить сигналы started() потоков к слотам run() наших
тестовых объектов, а сигналы finished() — к слотам потоков terminate(), чтобы
завершать работу потоков при завершении выполнения метода run().
Также присоединим сигнал посылки сообщения объекта exampleObject_1 к
слоту установки сообщения exampleObject_2.  Чтобы информация могла
передаваться, нужно пятым аргументом в метод connect передать флаг
Qt::DirectConnection , который установит непосредственное соединение
объектов и позволит выполнять передачу информации через систему сигналов
и слотов.
Ну и для передачи объектов в потоки необходимо воспользоваться методом
moveToTrhead(). */
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    // Запуск выполнения метода run будет осуществляться по сигналу запуска от
соответствующего потока
    connect(&thread_1, &QThread::started, &exampleObject_1,
&ExampleObject::run);
    connect(&thread_2, &QThread::started, &exampleObject_2,
&ExampleObject::run);
```

```
    // Остановка потока же будет выполняться по сигналу finished от
соответствующего объекта в потоке
    connect(&exampleObject_1, &ExampleObject::finished, &thread_1,
&QThread::terminate);
    connect(&exampleObject_2, &ExampleObject::finished, &thread_2,
&QThread::terminate);
    // соединение для передачи данных из первого объекта в первом потоке, ко
второму объекту во втором потоке
    connect(&exampleObject_1, &ExampleObject::sendMessage, &exampleObject_2,
&ExampleObject::setMessage_2, Qt::DirectConnection);
    exampleObject_1.moveToThread(&thread_1);    // Передаем объекты в
соответствующие потоки
    exampleObject_2.moveToThread(&thread_2);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::on_write_1_clicked()
{
    // Устанавливаем текст в первый объект в первом потоке
    exampleObject_1.setMessage(ui->lineEdit_1->text());
}

void MainWindow::on_write_2_clicked()
{
    // Устанавливаем текст во второй объект во втором потоке
    exampleObject_2.setMessage(ui->lineEdit_2->text());
}
void MainWindow::on_start_clicked()
{
    // Запуск потоков
    exampleObject_1.setRunning(true);
    exampleObject_2.setRunning(true);
    thread_1.start();
    thread_2.start();
}
void MainWindow::on_stop_clicked()
{
    // Остановка потоков через завершение выполнения методов run в объектах
    exampleObject_1.setRunning(false);
    exampleObject_2.setRunning(false);
}
```

Листинг П.5.1. Содержание файла lr5.pro серверной части приложения

```
#-------------------------------------------------
# Project created by QtCreator 2019-11-02T11:07:58
#-------------------------------------------------

QT       += core gui
QT += network

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = lr5
TEMPLATE = app

# The following define makes your compiler emit warnings if you use
# any feature of Qt which has been marked as deprecated (the exact warnings
# depend on your compiler). Please consult the documentation of the
# deprecated API in order to know how to port your code away from it.
DEFINES += QT_DEPRECATED_WARNINGS

# You can also make your code fail to compile if you use deprecated APIs.
# In order to do so, uncomment the following line.
# You can also select to disable deprecated APIs only up to a certain version of Qt.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all
the APIs deprecated before Qt 6.0.0

CONFIG += c++11

SOURCES += \
        main.cpp \
        myserver.cpp

HEADERS += \
        myserver.h

FORMS += \
        myserver.ui

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

Листинг П.5.2. Содержание файла main.cpp серверной части приложения

```
/*****************************************************************
Для реализации сервера Qt предоставляет удобный класс QTcpServer, который
предназначен для управления входящими TCP-соединениями. Программа
является реализацией простого сервера, который принимает и подтверждает
получение запросов клиентов.
Создается объект сервера. Чтобы запустить сервер, нужно создать объект
класса MyServer, передав в конструктор номер порта, по которому должен
осуществляться нужный сервис.
В нашем случае передается номер порта, равный 2323. */

#include <QtGui>
#include "MyServer.h"
#include <QApplication>
int main(int argc, char** argv)
{
   QApplication app(argc, argv);
   MyServer     server(2323);

   server.show();

   return app.exec();
}
```

Листинг П.5.3. Содержание файла myserver.h серверной части приложения

```
/*****************************************************************
В классе MyServer мы объявляем атрибут m_ptcpServer, который и является
основной частью для управления нашим сервером. Атрибут m_nNextBlockSize
служит для хранения длины следующего, полученного от сокета блока. */

#ifndef _MyServer_h_
#define _MyServer_h_
#include <QWidget>

class QTcpServer;
class QTextEdit;
class QTcpSocket;

class MyServer : public QWidget {
Q_OBJECT
private:
   QTcpServer* m_ptcpServer;
   QTextEdit*  m_ptxt;
   quint16     m_nNextBlockSize;
```

```
private:
   void sendToClient(QTcpSocket* pSocket, const QString& str);

public:
   MyServer(int nPort, QWidget* pwgt = 0);

public slots:
   virtual void slotNewConnection();
         void slotReadClient   ();
};
#endif  //_MyServer_h_
```

Листинг П.5.4. Содержание файла myserver.cpp серверной части приложения

```
#include "myserver.h"
#include "ui_myserver.h"
#include <QTcpSocket>
#include <QTcpServer>
#include <QMessageBox>
#include <QTextEdit>
#include <QVBoxLayout>
#include <QLabel>
#include <QTime>

/*****************************************************************
Для установки сервера нам необходимо вызвать в конструкторе метод listen().
В этот метод необходимо передать номер порта, который мы получили в
конструкторе.
При возникновении ошибочных ситуаций, например невозможности захвата
порта, этот метод возвратит значение false, на которое мы отреагируем показом
окна сообщения об ошибке.
Далее мы производим соединение с сигналом newConnection(), который
высылается при каждом присоединении нового клиента. Для отображения
информации мы создаем виджет многострочного текстового поля (указатель
m_ptxt) и устанавливаем в нем, вызовом метода setReadOnly(), режим,
делающий возможным только просмотр информации. */

MyServer::MyServer(int nPort, QWidget* pwgt /*=0*/) : QWidget(pwgt)
                                    , m_nNextBlockSize(0)
{
   m_ptcpServer = new QTcpServer(this);
   if (!m_ptcpServer->listen(QHostAddress::Any, nPort)) {
      QMessageBox::critical(0,
                        "Server Error",
                        "Unable to start the server:"
                        + m_ptcpServer->errorString()
```

```
                    );
        m_ptcpServer->close();
        return;
    }
    connect(m_ptcpServer, SIGNAL(newConnection()),
            this,         SLOT(slotNewConnection())
           );

    m_ptxt = new QTextEdit;
    m_ptxt->setReadOnly(true);

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(new QLabel("<H1>Server</H1>"));
    pvbxLayout->addWidget(m_ptxt);
    setLayout(pvbxLayout);
}

/*****************************************************************
Метод slotNewConnection() вызывается каждый раз при соединении с новым
клиентом.
Из этого метода мы выполняем соединения с сигналами disconnected() и
readyRead(), которые сигнализируют об отсоединении клиента и его готовности
к передаче данных соответственно.
В завершение мы вызываем метод sendToClient() для отсылки приветствия
присоединенному клиенту.
В этом методе, вторым параметром, мы передаем строку.
Внутри самого метода будет сгенерирован временной штамп, который будет
отослан клиенту вместе с переданной строкой.

Для подтверждения соединения с клиентом необходимо вызвать метод
nextPendingConnection(), который возвратит сокет, посредством которого
можно осуществлять дальнейшую связь с клиентом.
Мы соединяем сигнал disconnected(), высылаемый сокетом при отсоединении
клиента, со слотом deleteLater(), предназначенным для его последующего
уничтожения.
При поступлении запросов от клиентов высылается сигнал readyToRead(),
который мы соединяем со слотом slotReadClient(). */

/*virtual*/ void MyServer::slotNewConnection()
{
    QTcpSocket* pClientSocket = m_ptcpServer->nextPendingConnection();
    connect(pClientSocket, SIGNAL(disconnected()),
            pClientSocket, SLOT(deleteLater())
           );
```

```
    connect(pClientSocket, SIGNAL(readyRead()),
            this,          SLOT(slotReadClient())
           );

    sendToClient(pClientSocket, "Server Response: Connected!");
}

/*****************************************************************
Сначала производится преобразование указателя, возвращаемого методом
sender(), к типу QTcpSocket.
Цикл for нам нужен, так как не все высланные клиентом данные могут прийти
одновременно.
Поэтому сервер должен быть в состоянии получать как весь блок целиком, так
и только часть блока, а также и все блоки сразу. Каждый переданный сокетом
блок начинается полем, описывающим размер блока. Размер блока, который
считывается при условии того, что размер полученных данный не меньше двух
байт и атрибут m_nNextBlockSize равен нулю (то есть размер блока
неизвестен). Если доступных данных для чтения больше или равно размеру
блока, тогда производится считывание из потока данных в переменные time и
str.
После этого значение переменной time преобразуется вызовом метода toString()
в строку и используется вместе со строкой str для строки сообщения strMessage.
Строка сообщения добавляется в виджет текстового поля вызовом метода
append().
Анализ блока данных завершается присваиванием атрибуту m_nNextBlockSize
значения 0, которое говорит о том, что размер очередного блока данных
неизвестен.
Вызовом метода sendToClient() мы сообщаем клиенту о том, что нам успешно
удалось прочитать высланные им данные. */

void MyServer::slotReadClient()
{
    QTcpSocket* pClientSocket = (QTcpSocket*)sender();
    QDataStream in(pClientSocket);
    in.setVersion(QDataStream::Qt_4_2);
    for (;;) {
        if (!m_nNextBlockSize) {
            if (pClientSocket->bytesAvailable() < sizeof(quint16)) {
                break;
            }
            in >> m_nNextBlockSize;
        }

        if (pClientSocket->bytesAvailable() < m_nNextBlockSize) {
            break;
```

```
        }
        QTime   time;
        QString str;
        in >> time >> str;

        QString strMessage =
           time.toString() + " " + "Client has sent - " + str;
        m_ptxt->append(strMessage);

        m_nNextBlockSize = 0;

        sendToClient(pClientSocket,
                     "Server Response: Received \"" + str + "\""
                    );
    }
}
```

/***
В методе sendToClient() мы формируем данные, которые будут высланы клиенту.
Но есть один нюанс, который заключается в том, что нам неизвестен размер блока, и, следовательно, мы не можем записывать данные сразу в сокет, так как размер блока должен быть выслан в первую очередь. Поэтому мы прибегаем к следующим действиям.
Мы сначала создаем объект QByteArray. В него мы записываем все данные блока, причем записываем размер, равный 0. После этого мы перемещаем указатель на начало блока вызовом метода seek(), вычисляем и записываем размер блока в поток (out) (вычисляется как размер arrBlock с вычитанием из него sizeof(quint16)).
После этого созданный блок записывается в сокет вызовом метода write().

Примечание. Для пересылки обычных строк мы могли бы использовать класс потока ввода QTextStream.
Причиной, по которой используется класс QDataStream, является то, что пересылка бинарных данных представляет собой общий случай, это нам необходимо для того, чтобы переслать объект класса QTime.
То есть вы можете отправлять не только строки, но и еще растровые изображения, объекты палитры и т. д.
Далее мы будем использовать класс QDataStream. */

```
void MyServer::sendToClient(QTcpSocket* pSocket, const QString& str)
{
    QByteArray  arrBlock;
    QDataStream out(&arrBlock, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_2);
```

```
    out << quint16(0) << QTime::currentTime() << str;

    out.device()->seek(0);
    out << quint16(arrBlock.size() - sizeof(quint16));

    pSocket->write(arrBlock);
}
```

Листинг П.5.5. Содержание файла lr5_1.pro клиентской части приложения;

```
#-------------------------------------------------
# Project created by QtCreator 2019-11-02T11:15:09
#-------------------------------------------------

QT       += core gui
QT += network

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = lr5_1
TEMPLATE = app

# The following define makes your compiler emit warnings if you use
# any feature of Qt which has been marked as deprecated (the exact warnings
# depend on your compiler). Please consult the documentation of the
# deprecated API in order to know how to port your code away from it.
DEFINES += QT_DEPRECATED_WARNINGS

# You can also make your code fail to compile if you use deprecated APIs.
# In order to do so, uncomment the following line.
# You can also select to disable deprecated APIs only up to a certain version of Qt.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all
the APIs deprecated before Qt 6.0.0

CONFIG += c++11

SOURCES += \
        main.cpp \
        myclient.cpp

HEADERS += \
        myclient.h

FORMS += \
        myclient.ui
```

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

Листинг П.5.6. Содержание файла main.cpp клиентской части приложения;

```
/*****************************************************************
Для реализации клиента нужно создать объект класса QTcpSocket, а затем
вызвать метод connectToHost(), передав в него первым параметром имя
компьютера (или его IP-адрес), а вторым — номер порта сервера.
Объект класса QTcpSocket сам попытается произвести установку связи с
сервером и, в случае успеха, вышлет сигнал connected(). В противном случае
будет выслан сигнал error(int) с кодом ошибки, определенным в перечислении
QAbstractSocket::SocketError. Это может произойти, например, в том случае,
если на указанном компьютере не установлен сервер или не соответствует
номер порта.
После установления соединения объект класса QTcpSocket может высылать или
считывать данные сервера.
Высылка на сервер информации, введенной в однострочном текстовом поле
окна клиента, производится после нажатия на кнопку Send (Выслать).
В функции main() мы создаем объект клиента.
Если запуск сервера и клиента производится на одном и том же компьютере,
то в качестве имени компьютера можно передать строку localhost.
Номер порта в нашем случае равен 2323, так как это тот порт, который
используется нашим сервером. */

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

int main(int argc, char** argv)
{
   QApplication app(argc, argv);
   MyClient     client("localhost", 2323);

   client.show();

   return app.exec();
}
```

Листинг П.5.7. Содержание файла myclient.h клиентской части приложения;

```
/*****************************************************************
В классе MyClient мы объявляем атрибут m_pTcpSocket, который необходим
для управления нашим клиентом, и атрибут m_nNextBlockSize, необходимый
для хранения длины следующего полученного от сокета блока.
```

```
Остальные два атрибута — m_txtInfo и m_txtInput — используются для
отображения и ввода информации соответственно. */

#ifndef _MyClient_h_
#define _MyClient_h_

#include <QWidget>
#include <QTcpSocket>

class QTextEdit;
class QLineEdit;

class MyClient : public QWidget {
Q_OBJECT
private:
   QTcpSocket* m_pTcpSocket;
   QTextEdit*  m_ptxtInfo;
   QLineEdit*  m_ptxtInput;
   quint16     m_nNextBlockSize;

public:
   MyClient(const QString& strHost, int nPort, QWidget* pwgt = 0) ;

private slots:
   void slotReadyRead   (                            );
   void slotError       (QAbstractSocket::SocketError);
   void slotSendToServer(                            );
   void slotConnected   (                            );
};
#endif  //_MyClient_h_
```

Листинг П.5.8. Содержание файла myclient.cpp клиентской части приложения

```
#include "myclient.h"
#include "ui_myclient.h"
#include <QTcpSocket>
#include <QTcpServer>
#include <QMessageBox>
#include <QTextEdit>
#include <QVBoxLayout>
#include <QLabel>
#include <QTime>
#include <QLineEdit>
#include <QPushButton>
```

```
/*****************************************************************
В конструкторе происходит создание объекта сокета (указатель m_pTcpSocket).
Из объекта сокета вызывается метод connectToHost(), осуществляющий связь с
сервером.
Первым параметром в этот метод передается имя компьютера, а вторым —
номер порта.
Связь между сокетами асинхронна. Сокет высылает сигнал connected() как
только будет произведено соединение, а также высылает сигнал readyRead() о
готовности предоставить данные для чтения.
Мы соединяем сигналы connected(), readyRead() со слотами slotConnected() и
slotReadyRead() соответственно.
В случаях возникновения ошибок сокет высылает сигнал error(), который мы
соединяем со слотом slotError(), в котором производим отображение ошибок.
Затем создается пользовательский интерфейс программы, состоящий из
надписи, кнопки нажатия, однострочного и многострочного текстовых полей.
Сигнал clicked() виджета кнопки нажатия соединяется со слотом
slotSendToServer() класса MyClient, ответственным за отправку данных на
сервер.
Для того чтобы к аналогичному действию приводило и нажатие клавиши
<Enter>,
мы соединяем сигнал returnPressed() виджета текстового поля (m_ptxtInput)
с тем же слотом slotSendToServer(). */

MyClient::MyClient(const QString& strHost,
                   int            nPort,
                   QWidget*       pwgt /*=0*/
                  ) : QWidget(pwgt)
                    , m_nNextBlockSize(0)
{
    m_pTcpSocket = new QTcpSocket(this);
    m_pTcpSocket->connectToHost(strHost, nPort);
    connect(m_pTcpSocket, SIGNAL(connected()), SLOT(slotConnected()));
    connect(m_pTcpSocket, SIGNAL(readyRead()), SLOT(slotReadyRead()));
    connect(m_pTcpSocket, SIGNAL(error(QAbstractSocket::SocketError)),
            this,         SLOT(slotError(QAbstractSocket::SocketError))
           );

    m_ptxtInfo  = new QTextEdit;
    m_ptxtInput = new QLineEdit;

    connect(m_ptxtInput, SIGNAL(returnPressed()),
            this,        SLOT(slotSendToServer())
           );
    m_ptxtInfo->setReadOnly(true);
```

```
    QPushButton* pcmd = new QPushButton("&Send");
    connect(pcmd, SIGNAL(clicked()), SLOT(slotSendToServer()));

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(new QLabel("<H1>Client</H1>"));
    pvbxLayout->addWidget(m_ptxtInfo);
    pvbxLayout->addWidget(m_ptxtInput);
    pvbxLayout->addWidget(pcmd);
    setLayout(pvbxLayout);
}

/*****************************************************************
Слот slotReadyToRead() вызывается при поступлении данных от сервера.
Цикл for нужен, так как не все данные с сервера могут прийти одновременно.
Поэтому клиент должен быть в состоянии получить как весь блок целиком,
так и только часть блока или даже все блоки сразу.
Каждый переданный блок начинается полем, хранящим размер блока.
После того как мы будем уверены, что блок получен целиком,
то можем без опасения использовать оператор >> объекта потока QDataStream
(переменная in).
Чтение данных из сокета осуществляется при помощи объекта потока данных.
Полученная информация добавляется в виджет многострочного текстового
поля (указатель m_ptxtInfo) с помощью метода append().
В завершение анализа блока данных мы присваиваем атрибуту
m_nNextBlockSize значение 0, которое указывает на то, что размер очередного
блока данных неизвестен. */
void MyClient::slotReadyRead()
{
    QDataStream in(m_pTcpSocket);
    in.setVersion(QDataStream::Qt_4_2);
    for (;;) {
        if (!m_nNextBlockSize) {
            if (m_pTcpSocket->bytesAvailable() < sizeof(quint16)) {
                break;
            }
            in >> m_nNextBlockSize;
        }

        if (m_pTcpSocket->bytesAvailable() < m_nNextBlockSize) {
            break;
        }
        QTime   time;
        QString str;
        in >> time >> str;
```

```
        m_ptxtInfo->append(time.toString() + " " + str);
        m_nNextBlockSize = 0;
    }
}

/*****************************************************************
Слот slotError() вызывается при возникновении ошибок.
В нем мы преобразуем код ошибки в текст для того, чтобы отобразить его в
виджете многострочного текстового поля. */

void MyClient::slotError(QAbstractSocket::SocketError err)
{
    QString strError =
        "Error: " + (err == QAbstractSocket::HostNotFoundError ?
                     "The host was not found." :
                     err == QAbstractSocket::RemoteHostClosedError ?
                     "The remote host is closed." :
                     err == QAbstractSocket::ConnectionRefusedError ?
                     "The connection was refused." :
                     QString(m_pTcpSocket->errorString())
                    );
    m_ptxtInfo->append(strError);
}

/*****************************************************************
Обратите внимание на то, что мы не можем записывать данные сразу в
QTcpSocket, потому что мы не знаем размер блока, который должен быть
выслан в первую очередь.
Поэтому мы должны сначала создать объект QByteArray, для того чтобы
записывать все данные блока в него, записывая сначала размер равным 0. После
того как все необходимые данные блока записаны, мы перемещаем указатель на
начало блока и вызовом метода seek() записываем размер блока, который
вычисляется как размер arrBlock с вычитанием из него sizeof(quint16).
Это делается для исключения данных размера при подсчете байт.

После нажатия кнопки Send (Послать) вызывается слот slotSendToServer(),
который записывает в сокет строку, введенную пользователем в виджете
однострочного текстового поля (указатель m_ptxtInput). */

void MyClient::slotSendToServer()
{
    QByteArray  arrBlock;
    QDataStream out(&arrBlock, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_2);
    out << quint16(0) << QTime::currentTime() << m_ptxtInput->text();
```

```
    out.device()->seek(0);
    out << quint16(arrBlock.size() - sizeof(quint16));

    m_pTcpSocket->write(arrBlock);
    m_ptxtInput->setText("");
}

/*****************************************************************
Как только связь с сервером установлена, вызывается метод slotConnected()
и в виджет текстового поля добавляется строка сообщения. */

void MyClient::slotConnected()
{
    m_ptxtInfo->append("Received the connected() signal");
}
```

Листинг П.6.1. Содержание файла lr6.pro

```
#-------------------------------------------------
# Project created by QtCreator 2019-11-06T11:05:27
#-------------------------------------------------

QT       += core gui
QT       += core gui network
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = lr6
TEMPLATE = app

# The following define makes your compiler emit warnings if you use
# any feature of Qt which has been marked as deprecated (the exact warnings
# depend on your compiler). Please consult the documentation of the
# deprecated API in order to know how to port your code away from it.
DEFINES += QT_DEPRECATED_WARNINGS

# You can also make your code fail to compile if you use deprecated APIs.
# In order to do so, uncomment the following line.
# You can also select to disable deprecated APIs only up to a certain version of Qt.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all
the APIs deprecated before Qt 6.0.0

CONFIG += c++11
SOURCES += \
        downloader.cpp \
        main.cpp \
        widget.cpp
```

```
HEADERS += \
        downloader.h \
        widget.h

FORMS += \
        widget.ui

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

Листинг П.6.2. Содержание файла "widget.h"

```
/*****************************************************************
Заголовочный файл окна приложения. В нем мы подключим заголовочный
файл класса Downloader, который будет отвечать за скачивание данных с сайта
и сохранения их в файл.
Объявим объект данного класса Downloader.
Также присутствует объявление сигнатуры слота для чтения данных из
сохраненного файла по окончанию скачивания. */

#ifndef WIDGET_H
#define WIDGET_H
#include <QWidget>
#include <QFile>
#include <downloader.h>

namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

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

private slots:
    void readFile();

private:
    Ui::Widget *ui;
```

```
    Downloader *downloader; // Объявляем объект класса для скачивания данных
по http
};

#endif // WIDGET_H
```

Листинг П.6.3. Содержание файла widget.cpp

```
/*************************************************************************
В исходных кодах присутствует два коннекта сигналов к слотам.
Один коннект отвечает за обработку нажатия кнопки, а второй — за чтение
данных из файла по окончании скачивания файла. */

#include "widget.h"
#include "ui_widget.h"

Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    ui->setupUi(this);

    downloader = new Downloader(); // Инициализируем Downloader
    // по нажатию кнопки запускаем получение данных по http
    connect(ui->pushButton, &QPushButton::clicked, downloader,
&Downloader::getData);
    // по окончании получения данных считываем данные из файла
    connect(downloader, &Downloader::onReady, this, &Widget::readFile);

}

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

void Widget::readFile()
{
    QFile file("C:/Qt/lr6/file.xml");
    if (!file.open(QIODevice::ReadOnly)) // Открываем файл, если это возможно
        return; // если открытие файла невозможно, выходим из слота
    // в противном случае считываем данные и устанавливаем их в textEdit
    ui->textEdit->setText(file.readAll());
}
```

Листинг П.6.4. Содержание файла donwloader.h

```
/*****************************************************************
Объявляем экземпляр класса QNetworkAccesManager, а также методы для
инициализации запроса к сайту по его URL и обработки полученного ответа. */

#ifndef DOWNLOADER_H
#define DOWNLOADER_H

#include <QObject>
#include <QNetworkAccessManager>
#include <QNetworkRequest>
#include <QNetworkReply>
#include <QFile>
#include <QUrl>
#include <QDebug>

class Downloader : public QObject
{
   Q_OBJECT
public:
   explicit Downloader(QObject *parent = 0);

signals:
   void onReady();

public slots:
   void getData();     // Метод инициализации запроса на получение данных
   void onResult(QNetworkReply *reply);    // Слот обработки ответа о
полученных данных

private:
   QNetworkAccessManager *manager; // менеджер сетевого доступа
};

#endif // DOWNLOADER_H
```

Листинг П.6.5. Содержание файла downloader.cpp

```
/*****************************************************************
ВНИМАНИЕ!!!  Проверьте доступность URL перед запуском приложения. */

#include "downloader.h"

Downloader::Downloader(QObject *parent) : QObject(parent)
{
```

```
    // Инициализируем менеджер ...
    manager = new QNetworkAccessManager();
    // ... и подключаем сигнал о завершении получения данных к обработчику
полученного ответа
    connect(manager, &QNetworkAccessManager::finished, this,
&Downloader::onResult);
}

void Downloader::getData()
{
    QUrl url("http://www.mtbank.by/currxml.php"); // URL, с которого будем
получать данные
    QNetworkRequest request;    // Отправляемый запрос
    request.setUrl(url);        // Устанавлвиваем URL в запрос
    manager->get(request);      // Выполняем запрос
}

void Downloader::onResult(QNetworkReply *reply)
{
    // Если в процесе получения данных произошла ошибка
    if(reply->error()){
        // Сообщаем об этом и показываем информацию об ошибках
        qDebug() << "ERROR";
        qDebug() << reply->errorString();
    } else {
        // В противном случае создаем объект для работы с файлом
        QFile *file = new QFile("C:/Qt/lr6/file.xml");
        // Создаем файл или открываем его на перезапись ...
        if(file->open(QFile::WriteOnly)){
            file->write(reply->readAll());  // ... и записываем всю информацию со
страницы в файл
            file->close();                  // закрываем файл
        qDebug() << "Downloading is completed";
        emit onReady(); // Посылаем сигнал о завершении получения файла
        }
    }
}
```

Листинг П.7.1. Содержание файла lr7.pro

```
#-------------------------------------------------
# Project created by QtCreator 2019-11-14T16:37:20
#-------------------------------------------------

QT       += core
QT       += gui
```

```
QT       += sql

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = lr7
TEMPLATE = app
SOURCES += main.cpp\
        mainwindow.cpp \
    database.cpp \
    dialogadddevice.cpp

HEADERS  += mainwindow.h \
    database.h \
    dialogadddevice.h

FORMS    += mainwindow.ui \
    dialogadddevice.ui
```

Листинг П.7.2. Содержание файла main.cpp

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

    return a.exec();
}
```

Листинг П.7.3. Содержание файла mainwindow.h

```
/*************************************************************************
В заголовочном файле главного окна определяем слоты для запуска диалога
добавления записей, обновления модели и запуска диалога редактирования
записей.
Также имеются методы для инициализации внешнего вида окна, помимо
основных настроек, выполняемых в дизайнере интерфейсов Qt, а также метод
инициализации модели представления данных. */

#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
#include <QSqlTableModel>
#include <database.h>
#include <dialogadddevice.h>
```

```
namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:
    void on_addDeviceButton_clicked();
    void slotUpdateModels();
    void slotEditRecord(QModelIndex index);

private:
    Ui::MainWindow          *ui;
    DataBase                *db;
    QSqlTableModel          *modelDevice;
private:
    void setupModel(const QString &tableName, const QStringList &headers);
    void createUI();
};

#endif // MAINWINDOW_H
```

Листинг П.7.4. Содержание файла mainwindow.cpp

```
/*****************************************************************
В файле исходного кода основного окна производим инициализацию таблицы с данными.
Также прописываем логику поведения приложения при нажатии кнопки «Добавить», которая вызывает диалог добавления записи в таблицу.
Также этот диалог вызывается при двойном нажатии на запись в таблице данных.
В этом случае в диалог передается информация о том, какая запись была нажата и ее данные подставляются в поля для редактирования. */

#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
```

```
{
    ui->setupUi(this);
    this->setWindowTitle("QDataWidgetMapper Example");
    /* Первым делом необходимо создать объект для работы с базой данных и
инициализировать подключение к базе данных */
    db = new DataBase();
    db->connectToDataBase();
    /* Инициализируем модели для представления данных с заданием названий
колонок */
    this->setupModel(DEVICE,
                     QStringList() << trUtf8("id")
                                   << trUtf8("Имя хоста")
                                   << trUtf8("IP адрес")
                                   << trUtf8("MAC-адрес")
            );
    /* Инициализируем внешний вид таблицы с данными */
    this->createUI();
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::setupModel(const QString &tableName, const QStringList
&headers)
{
    /* Производим инициализацию модели представления данных */
    modelDevice = new QSqlTableModel(this);
    modelDevice->setTable(tableName);
    modelDevice->select();
    /* Устанавливаем названия колонок в таблице с сортировкой данных */
    for(int i = 0, j = 0; i < modelDevice->columnCount(); i++, j++){
        modelDevice->setHeaderData(i,Qt::Horizontal,headers[j]);
    }
}

void MainWindow::createUI()
{
    ui->deviceTableView->setModel(modelDevice);     // Устанавливаем модель на
TableView
    ui->deviceTableView->setColumnHidden(0, true);    // Скрываем колонку с id
записей
    // Разрешаем выделение строк
    ui->deviceTableView->setSelectionBehavior(QAbstractItemView::SelectRows);
```

```
    // Устанавливаем режим выделения лишь одной строки в таблице
    ui->deviceTableView->setSelectionMode(QAbstractItemView::SingleSelection);
    // Устанавливаем размер колонок по содержимому
    ui->deviceTableView->resizeColumnsToContents();
    ui->deviceTableView->setEditTriggers(QAbstractItemView::NoEditTriggers);
    ui->deviceTableView->horizontalHeader()->setStretchLastSection(true);

    connect(ui->deviceTableView, SIGNAL(doubleClicked(QModelIndex)), this,
SLOT(slotEditRecord(QModelIndex)));
}

/* Метод для активации диалога добавления записей */
void MainWindow::on_addDeviceButton_clicked()
{
    /* Создаем диалог и подключаем его сигнал завершения работы к слоту
обновления вида модели представления данных */
    DialogAddDevice *addDeviceDialog = new DialogAddDevice();
    connect(addDeviceDialog, SIGNAL(signalReady()), this,
SLOT(slotUpdateModels()));
    /* Выполняем запуск диалогового окна */
    addDeviceDialog->setWindowTitle(trUtf8("Добавить Устройство"));
    addDeviceDialog->exec();
}

/* Слот обновления модели представления данных */
void MainWindow::slotUpdateModels()
{
    modelDevice->select();
}

/* Метод для активации диалога добавления записей в режиме редактирования
с передачей индекса выбранной строки */
void MainWindow::slotEditRecord(QModelIndex index)
{
    /* Также создаем диалог и подключаем его сигнал завершения работы к
слоту обновления вида модели представления данных, но передаем в качестве
параметров строку записи */
    DialogAddDevice *addDeviceDialog = new DialogAddDevice(index.row());
    connect(addDeviceDialog, SIGNAL(signalReady()), this,
SLOT(slotUpdateModel()));

    /* Выполняем запуск диалогового окна */
    addDeviceDialog->setWindowTitle(trUtf8("Редактировать Устройство"));
    addDeviceDialog->exec();
}
```

Листинг П.7.5. Содержание файла database.h

```
//Вспомогательный класс для работы с базой данных

#ifndef DATABASE_H
#define DATABASE_H
#include <QObject>
#include <QSql>
#include <QSqlQuery>
#include <QSqlError>
#include <QSqlDatabase>
#include <QFile>
#include <QDate>
#include <QDebug>

/* Директивы имен таблицы, полей таблицы и базы данных */
#define DATABASE_HOSTNAME   "ExampleDataBase"
#define DATABASE_NAME       "DataBase.db"
#define DEVICE              "DeviceTable"
#define DEVICE_HOSTNAME     "Hostname"
#define DEVICE_IP           "IP"
#define DEVICE_MAC          "MAC"

class DataBase : public QObject
{
    Q_OBJECT
public:
    explicit DataBase(QObject *parent = 0);
    ~DataBase();
    /* Методы для непосредственной работы с классом.
     * Подключение к базе данных и вставка записей в таблицу */
    void connectToDataBase();
    bool inserIntoDeviceTable(const QVariantList &data);

private:
    // Сам объект базы данных, с которым будет производиться работа
    QSqlDatabase    db;

private:
    /* Внутренние методы для работы с базой данных */
    bool openDataBase();
    bool restoreDataBase();
    void closeDataBase();
    bool createDeviceTable();
};
#endif // DATABASE_H
```

Листинг П.7.6. Содержание файла database.cpp

```
#include "database.h"

DataBase::DataBase(QObject *parent) : QObject(parent)
{
}

DataBase::~DataBase()
{
}

/* Методы для подключения к базе данных */
void DataBase::connectToDataBase()
{
    /* Перед подключением к базе данных производим проверку на ее
существование.
     * В зависимости от результата производим открытие базы данных или ее
восстановление */
    if(!QFile("C:/Qt/lr7/" DATABASE_NAME).exists()){
        this->restoreDataBase();
    } else {
        this->openDataBase();
    }
}

/* Методы восстановления базы данных */
bool DataBase::restoreDataBase()
{
    if(this->openDataBase()){
        if(!this->createDeviceTable()){
            return false;
        } else {
            return true;
        }
    } else {
        qDebug() << "Не удалось восстановить базу данных";
        return false;
    }
    return false;
}

/* Метод для открытия базы данных */
bool DataBase::openDataBase()
{
```

```
    /* База данных открывается по заданному пути и имени базы данных, если
она существует */
    db = QSqlDatabase::addDatabase("QSQLITE");
    db.setHostName(DATABASE_HOSTNAME);
    db.setDatabaseName("C:/Qt/lr7/" DATABASE_NAME);
    if(db.open()){
        return true;
    } else {
        return false;
    }
}

/* Методы закрытия базы данных */
void DataBase::closeDataBase()
{
    db.close();
}
/* Метод для создания таблицы устройств в базе данных */
bool DataBase::createDeviceTable()
{
    /* В данном случае используется формирование сырого SQL-запрос с
последующим его выполнением. */
    QSqlQuery query;
    if(!query.exec( "CREATE TABLE " DEVICE " ("
                        "id INTEGER PRIMARY KEY AUTOINCREMENT, "
                        DEVICE_HOSTNAME  " VARCHAR(255)    NOT NULL,"
                        DEVICE_IP        " VARCHAR(16)     NOT NULL,"
                        DEVICE_MAC       " VARCHAR(18)     NOT NULL"
                    " )"
                ")){
        qDebug() << "DataBase: error of create " << DEVICE;
        qDebug() << query.lastError().text();
        return false;
    } else {
        return true; }
    return false;
}
/* Метод для вставки записи в таблицу устройств */
bool DataBase::inserIntoDeviceTable(const QVariantList &data)
{
    /* Запрос SQL формируется из QVariantList, в который передаются данные
для вставки в таблицу. */
    QSqlQuery query;
    /* В начале SQL запрос формируется с ключами, которые потом связываются
методом bindValue для подстановки данных из QVariantList */
```

```
    query.prepare("INSERT INTO " DEVICE " ( " DEVICE_HOSTNAME ", "
                                  DEVICE_IP ", "
                                  DEVICE_MAC " ) "
                  "VALUES (:Hostname, :IP, :MAC )");
    query.bindValue(":Hostname",    data[0].toString());
    query.bindValue(":IP",          data[1].toString());
    query.bindValue(":MAC",         data[2].toString());
    // После чего выполняется запрос методом exec()
    if(!query.exec()){
        qDebug() << "error insert into " << DEVICE;
        qDebug() << query.lastError().text();
        return false;
    } else {
        return true;
    }
    return false;
}
```

Листинг П.7.7. Содержание файла dialogadddevice.h

```
/*****************************************************************
Заголовочный файл диалога добавления и редактирования записей.
Как видно по заголовочному файлу, здесь также используется модель для
представления данных, но ее данные транслируются не в QTableView, как в
классе MainWindow, а в объект класса QDataWidgetMapper. Также здесь
переопределяется метод accept(), поскольку прежде чем закрывать окно,
необходимо удостовериться, что данные заполнены верно.
В данном проекте критерием правильности заполнения данных является
отсутствие дублирующихся записей. */

#ifndef DIALOGADDDEVICE_H
#define DIALOGADDDEVICE_H
#include <QDialog>
#include <QSqlTableModel>
#include <QDataWidgetMapper>
#include <QMessageBox>
#include <QRegExp>
#include <QRegExpValidator>
#include <database.h>

namespace Ui {
class DialogAddDevice;
}

class DialogAddDevice : public QDialog
{
```

```
    Q_OBJECT

public:
    explicit DialogAddDevice(int row = -1, QWidget *parent = 0);
    ~DialogAddDevice();

signals:
    void signalReady();

private slots:
    void on_buttonBox_accepted();
    void updateButtons(int row);

private:
    Ui::DialogAddDevice        *ui;
    QSqlTableModel             *model;
    QDataWidgetMapper          *mapper;
private:
    void setupModel();
    void createUI();
    void accept();
};

#endif // DIALOGADDDEVICE_H
```

Листинг П.7.8. Содержание файла dialogadddevice.cpp

```
/*****************************************************************
В данном файле реализована логика работы класса.
Проверка на правильность введенных данных осуществляется в момент
нажатия кнопки OK.
В случае правильно заполненных полей производится вставка записи в таблицу
или ее редактирование.
Валидация данных в полях IP адрес и MAC адрес производится с помощью
классов, описанных в заголовочном файле.
Также реализована проверка на существование записи с похожими данными. */

#include "dialogadddevice.h"
#include "ui_dialogadddevice.h"

DialogAddDevice::DialogAddDevice(int row, QWidget *parent) :
    QDialog(parent),
    ui(new Ui::DialogAddDevice)
{
    ui->setupUi(this);
```

```
    /* Метод для инициализации модели, из которой будут транслироваться
данные */
    setupModel();

    /* Если строка не задана, то есть равна -1, тогда диалог работает по принципу
создания новой записи.
     * А именно, в модель вставляется новая строка, и работа ведется с ней. */
    if(row == -1){
        model->insertRow(model->rowCount(QModelIndex()));
        mapper->toLast();
    /* В противном случае диалог настраивается на заданную запись */
    } else {
        mapper->setCurrentModelIndex(model->index(row,0));
    }
    createUI();
}
DialogAddDevice::~DialogAddDevice()
{
    delete ui;
}

/* Метод настройки модели данных и mapper */
void DialogAddDevice::setupModel()
{
    /* Инициализируем модель и делаем выборку из нее */
    model = new QSqlTableModel(this);
    model->setTable(DEVICE);
    model->setEditStrategy(QSqlTableModel::OnManualSubmit);
    model->select();

    /* Инициализируем mapper и привязываем поля данных к объектам LineEdit*/
    mapper = new QDataWidgetMapper();
    mapper->setModel(model);
    mapper->addMapping(ui->HostnameLineEdit, 1);
    mapper->addMapping(ui->IPAddressLineEdit, 2);
    mapper->addMapping(ui->MACLineEdit, 3);
    /* Ручное подтверждение изменения данных через mapper */
    mapper->setSubmitPolicy(QDataWidgetMapper::ManualSubmit);

    /* Подключаем коннекты от кнопок пролистывания к пролистыванию модели
данных в mapper */
    connect(ui->previousButton, SIGNAL(clicked()), mapper, SLOT(toPrevious()));
    connect(ui->nextButton, SIGNAL(clicked()), mapper, SLOT(toNext()));
    /* При изменении индекса в mapper изменяем состояние кнопок */
```

```
    connect(mapper, SIGNAL(currentIndexChanged(int)), this,
SLOT(updateButtons(int)));
}

/* Метод для установки валидатора на поле ввода IP и MAC адресов */
void DialogAddDevice::createUI()
{
    QString ipRange = "(?:[0-1]?[0-9]?[0-9]|2[0-4][0-9]|25[0-5])";
    QRegExp ipRegex ("^" + ipRange
                     + "\\." + ipRange
                     + "\\." + ipRange
                     + "\\." + ipRange + "$");
    QRegExpValidator *ipValidator = new QRegExpValidator(ipRegex, this);
    ui->IPAddressLineEdit->setValidator(ipValidator);

    QString macRange = "(?:[0-9A-Fa-f][0-9A-Fa-f])";
    QRegExp macRegex ("^" + macRange
                      + "\\:" + macRange
                      + "\\:" + macRange
                      + "\\:" + macRange
                      + "\\:" + macRange
                      + "\\:" + macRange + "$");
    QRegExpValidator *macValidator = new QRegExpValidator(macRegex, this);
    ui->MACLineEdit->setValidator(macValidator);
}

void DialogAddDevice::on_buttonBox_accepted()
{
    /* SQL-запрос для проверки существования записи с такими же учетными
данными.
     * Если запись не существует или находится лишь индекс редактируемой в
данный момент записи, то диалог позволяет вставку записи в таблицу данных*/
    QSqlQuery query;
    QString str = QString("SELECT EXISTS (SELECT " DEVICE_HOSTNAME "
FROM " DEVICE
                          " WHERE ( " DEVICE_HOSTNAME " = '%1' "
                          " OR " DEVICE_IP " = '%2' )"
                          " AND id NOT LIKE '%3' )")
            .arg(ui->HostnameLineEdit->text(),
                 ui->IPAddressLineEdit->text(),
                 model->data(model->index(mapper->currentIndex(),0),
Qt::DisplayRole).toString());
    query.prepare(str);
    query.exec();
    query.next();
```

```
    /* Если запись существует, то диалог вызывает предупредительное
сообщение */
    if(query.value(0) != 0){
        QMessageBox::information(this, trUtf8("Ошибка хоста"),
                                 trUtf8("Хост с таким именем или IP-адресом уже
существует"));
    /* В противном случае производится вставка новых данных в таблицу, и
диалог завершается с передачей сигнала для обновления таблицы в главном
окне */
    } else {
        mapper->submit();
        model->submitAll();
        emit signalReady();
        this->close();
    }
}
void DialogAddDevice::accept()
{
}

/* Метод изменения состояния активности кнопок пролистывания */
void DialogAddDevice::updateButtons(int row)
{
    /* В том случае если мы достигаем одного из крайних (самый первый или
самый последний) индексов в таблице данных, то мы изменяем состояние
соответствующей кнопки на состояние неактивное */
    ui->previousButton->setEnabled(row > 0);
    ui->nextButton->setEnabled(row < model->rowCount() - 1);
}
```

Учебное издание

Городничев Михаил Геннадьевич
Фатхулин Тимур Джалилевич
Джабраилов Хизар Абубакарович

Разработка кроссплатформенного программного обеспечения

Учебное пособие

Издательство «Наукоемкие технологии»
ООО «Корпорация «Интел Групп»
https://publishing.intelgr.com
e-mail: publishing@intelgr.com
Тел.: +7 (812) 945-50-63

Подписано в печать 06.11.2023.
Формат 60×84/16.
Объем 9,25 печ.л.
Тираж 500 экз.

ISBN 978-5-907804-04-3