Министерство науки и высшего образования Российской федерации

Томский государственный университет
систем управления и радиоэлектроники

К.О. Гусаченко,
А.О. Семкин

ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

Методические указания к лабораторной работе
«Библиотека Qt. Работа с файлами»

Томск
2020

УДК 004.428.4 + 519.683.8
ББК 32.972.1я73

Рецензент:
Перин А.С., доцент кафедры сверхвысокочастотной и квантовой радиотехники ТУСУР, канд. техн. наук

Гусаченко, Ксения Олеговна
Информационные технологии: Методические указания к лабораторной работе «Библиотека Qt. Работа с файлами» / К.О. Гусаченко, А.О. Семкин. – Томск: Томск. гос. ун-т систем упр. и радиоэлектроники, 2020. – 14 с.

Настоящие методические указания составлены с учетом требований федерального государственного образовательного стандарта высшего образования (ФГОС ВО).

Изложены принципы работы с файлами и директориями (создание, чтение/запись, удаление) в среде разработки *Qt Creator*. Представлены методические материалы компьютерной лабораторной работы, посвященной разработке приложений, работающих с файловой структурой компьютера.

Предназначены для студентов очной и заочной форм обучения по направлению подготовки бакалавриата 11.03.02 «Инфокоммуникационные технологии и системы связи», 11.03.01 «Радиотехника».

Одобрено на заседании каф. СВЧиКР протокол №_6__ от __30.01.2020

УДК 004.428.4 + 519.683.8
ББК 32.972.1я73

Оглавление

1. Цель работы

Целью данной работы является изучение принципов работы с файлами и директориями при помощи средств разработки *Qt*.

2. Введение

Qt представляет собой комплексную рабочую среду, предназначенную для разработки на *C++* межплатформенных приложений с графическим пользовательским интерфейсом по принципу «написал программу – компилируй ее в любом месте». *Qt* позволяет программистам использовать дерево классов с одним источником в приложениях, которые будут работать в системах *Windows*, *Mac OS X*, *Linux*. *Solaris*, *HP-UX* и во многих других.

Графический пользовательский интерфейс (*GUI — Graphical User Interface*) это средства позволяющие пользователям взаимодействовать с аппаратными составляющими компьютера комфортным и удобным для себя образом.

В рамках данной лабораторной работы будут рассмотрено создание процессов и потоков в среде *Qt Creator*.

3. Работа с файлами в *Qt*

Редко встречается приложение, которое не обращается к файлам. Работа с директориями (папками, в терминологии ОС *Windows*) и файлами — это та область, в которой не все операции являются платформонезависимыми, поэтому *Qt* предоставляет свою собственную поддержку этих операций, состоящую из следующих классов:

- *QDir* — для работы с директориями;
- *QFile* — для работы с файлами;
- *QFileInfo* — для получения файловой информации;
- *QIODevice* — абстрактный класс для ввода/вывода;
- *QBuffer* — для эмуляции файлов в памяти компьютера.

Рассмотрим классы для работы с файлами в *Qt.*

3.1. Ввод/вывод. Класс *QIODevice*

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.

Реализация конкретного устройства происходит в унаследованных классах.

В *Qt* есть четыре класса, наследующие класс *QIODevice*:

QFile — класс для проведения операций с файлами;

QBuffer — позволяет записывать и считывать данные в массив *QByteArray*, как будто бы это устройство или файл;

QAbstractSocket — базовый класс для сетевой коммуникации посредством сокетов.

QProcess — этот класс предоставляет возможность запуска процессов с дополнительными аргументами и позволяет обмениваться информацией с этими процессами посредством методов, определенных в *QIODevice*.

Для работы с устройством его необходимо открыть в одном из режимов, определенных в заголовочном файле класса *QIODevice*:

QIODevice::NotOpen — устройство не открыто (это значение не имеет смысла передавать в метод *open*());

QIODevice::ReadOnly — открытие устройства только для чтения данных;

QIODevice::writeOnly — открытие устройства только для записи данных;

QIODevice::ReadWrite — открытие устройства для чтения и записи данных (то же, что и *IO_ReadOnly | IO_WriteOnly*);

QIODevice::Append — открытие устройства для добавления данных;

QIODevice::Unbuffered — открытие для непосредственного доступа к данным, в обход промежуточных буферов чтения и записи;

QIODevice::Text — применяются преобразования символов переноса строки в зависимости от платформы. Для ОС *Windows*, например — *\r\n*, а для *MacOS X* и *UNIX — /r*;

QIODevice::Truncate — все данные устройства, по возможности, должны быть удалены при открытии.

Для того чтобы в любой момент времени исполнения программы узнать, в каком из режимов было открыто устройство, нужно вызвать метод *openMode()*.

Считывать и записывать данные можно с помощью методов *read*() и *write*() соответственно. Для чтения всех данных сразу определен метод *readAll*(), который возвращает их в объекте типа *QByteArray*. Строку или символ можно прочитать методами *readLine*() и *getChar*() соответственно.

В классе *QIODevice* определен метод для смены текущего положения *seek*(). Получить текущее положение можно вызовом метода *pos*(). Но не забывайте, что эти методы применимы только для прямого доступа к данным. При последовательном доступе, каким является сетевое соединение, они теряют смысл. Более того, в этом случае теряет смысл и метод *size*(), возвращающий размер данных устройства. Все эти операции применимы только для *QFile*, *QBuffer* и *QTemporaryFile*.

Для создания собственного класса устройства ввода/вывода, для которого *Qt* не предоставляет поддержки, необходимо унаследовать класс *QIODevice* и реализовать в нем методы *readData*() и *writeData*(). В большинстве случаев может потребоваться перезаписать методы *open*(), *close*() и *atEnd*().

Благодаря интерфейсу класса *QIODevice* можно работать со всеми устройствами одинаково, при этом не имеет значения, является ли устройство файлом, буфером или другим устройством. Выведем на консоль данные из любого устройства.

```
void print(QIODevice *pdev)
```

```
{
    char ch;
    QString str;
    pdev->open(QIODevice::ReadOnly);
    for (; !pdev->atEnd();)
    {
        pdev->getChar(&ch);
        str += ch;
    }
    pdev->close();
    qDebug() << str;
}
```

Класс *QIODevice* предоставляет ряд методов, с помощью которых можно получить информацию об устройстве ввода/вывода. Например, одни устройства могут только записывать информацию, другие — только считывать, а третьи способны делать и то, и другое. Чтобы узнать об этом, следует воспользоваться методами *isReadable*() и *isWriteable*().

3.2. Класс *QFile*

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. Можно ничего не передавать в конструкторе, а сделать это после создания объекта, вызовом метода *setName*(). Например:

```
QFile file;
file.setName("file.dat");
```

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

Продемонстрируем применение некоторых методов работы с файлами:

```
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 [1024];
while(!file1.atEnd())
{
    int nBlocksize = file1.read(a, sizeof(a));
    file2.write(a, nBlocksize);
}
```

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();
```

Примечание: Операция считывания всех данных сразу, в зависимости от размера файла, может занять много оперативной памяти, а значит, к этому следует прибегать только в случаях острой необходимости или в том случае, когда файлы занимают мало места. Расход памяти при считывании сразу всего файла можно значительно сократить при том условии, что файл содержит избыточную информацию. Тогда можно воспользоваться функциями сжатия *qCompress*() и *qUncompress*(), которые определены вместе с классом *QByteArray*. Эти функции получают, в качестве аргумента, объект класса *QByteArray* и возвращают, в качестве результата, новый объект класса *QByteArray*.

Для удаления файла класс *QFile* содержит статический метод *remove*(). В этот метод необходимо передать строку, содержащую полный или относительный путь удаляемого файла.

3.3. *Класс QBuffer*

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. Можно это так же сделать при помощи потоков, которые будут рассмотрены далее. Рассмотрим пример использования класса *QBuffer*:

```
QByteArray arr;
QBuffer buffer(&arr);
buffer.open(QIODevice::WriteOnly);
QDataStream out(&buffer);
out << QString("Message");
```

Как видно из этого примера, 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.

Класс *QBuffer* полезен для проведения операций кэширования. Например, можно считывать файлы растровых изображений в объекты класса *QBuffer*, а затем, по необходимости, получать данные из них.

3.4. *Класс QTemporaryFile*

Иногда приложению может потребоваться создать временный файл. Это может быть связано, например, с промежуточным хранением большого объема данных или передачей этих данных какой-либо другой программе.

Класс *QTemporaryFile* представляет реализацию для временных файлов. Этот класс самостоятельно создает себе имя с гарантией его уникальности, для того чтобы не возникало конфликтов, в результате которых могли бы пострадать уже существующие файлы. Сам файл будет расположен в каталоге для промежуточных данных, местонахождение которого можно получить вызовом метода *QDir::tempPath*(). С уничтожением объекта будет уничтожен и сам временный файл.

4. Информация о файлах. Класс *QFileInfo*

Задача этого класса состоит в предоставлении информации о свойствах файла, например: имя, размер, время последнего изменения, права доступа и т.д. Объект класса *QFileInfo* создается передачей в его конструктор пути к файлу, но можно передавать и объекты класса *QFile*.

Файл или каталог?

Иногда необходимо убедиться, что исследуемый объект является каталогом, а не файлом и наоборот. Для этой цели существуют методы *isFile*() и *isDir*().

В том случае, если объект является файлом, метод *isFile*() возвращает значение булевого типа *true*, иначе — *false*. Если объект является директорией, то метод *isDir*() возвращает *true*, иначе — *false*. Кроме этих методов, класс *QFileInfo* содержит метод *isSymLink*(), возвращающий *true*, если объект является символьной ссылкой (*symbolic link* или *shortcut* в ОС *Windows*).

Примечание: Символьные ссылки применяются в *UNIX* для обеспечения связи с файлами или каталогами. Создаются они при помощи команды "*ln*" с ключом "*-s*".

Путь и имя файла в *Qt*

Чтобы получить путь к файлу, нужно воспользоваться методом *absoluteFilePath*(). Для получения относительного пути к файлу следует использовать метод *filePath*(). Для получения имени файла нужно вызвать метод *fileName*(), который возвращает имя файла вместе с его расширением. Если нужно только имя файла, то следует вызвать метод *baseName*(). Для получения расширения используется метод *completeSuffix*().

Информация о дате и времени файла в *Qt*

Иногда нужно узнать время создания файла, время его последнего изменения или чтения. Для этого класс *QFileInfo* предоставляет методы *created*(), *lastModified*() и *lastRead*() соответственно. Эти методы возвращают объекты класса *QDateTime*, которые можно преобразовать в строку методом *toString*(). Например:

```
//Дата и время создания файла
fileInfo.created().toString();

//Дата и время последнего изменения файла
fileInfo.lastModified().toString();

//Дата и время последнего чтения файла
fileInfo.lastRead().toString();
```

Получение атрибутов файла в *Qt*

Атрибуты файла дают информацию о том, какие операции можно проводить с файлом.

Для их получения в классе *QFileInfo* существуют следующие методы:

isReadable() —возвращает *true*, если из указанного файла можно читать информацию;

isWriteable() —возвращает *true*, если в указанный файл можно записывать информацию;

isHidden() — возвращает *true*, если указанный файл является скрытым;

isExecutable() —возвращает *true*, если указанный файл можно исполнять. В ОС *UNIX* это определяется не на основании расширения файла, как привыкли

считать программисты в *DOS* и ОС *Windows*, а посредством свойств самого файла.

Определение размера файла в *Qt*

Метод *size*() класса *QFileInfo* возвращает размер файла в байтах. Размер файлов редко отображается в байтах, чаще используются специальные буквенные обозначения, сообщающие об его размере. Например, для килобайта — это буква *K*, для мегабайта — *M*, для гигабайта — *G*, а для терабайта — *T*. Следующая функция позволяет сопровождать буквенными обозначениями размеры, лежащие даже в терабайтном диапазоне:

```
QString fileSize(qint64 nSize)
{
    qint64 i = 0;
    for (; nSize > 1023; nSize /= 1024, ++i) { }
    return QString().setNum(nSize) + "BKMGT"[i];
}
```

5. Потоки ввода/вывода

Объекты файлов, сами по себе, обладают только элементарными методами для чтения и записи информации. Использование потоков делает запись и считывание файлов более простым и гибким. Для файлов, содержащих текстовую информацию, следует использовать класс *QTextStream*, а для двоичных файлов — класс *QDataStream*.

Применение классов *QTextStream* и *QDataStream* такое же, как и для стандартного потока в языке *C++* (*iostream*), с той лишь разницей, что они могут работать с объектами класса *QIODevice*. Благодаря этому, потоки можно использовать и для своих собственных классов, унаследованных от *QIODevice*. Для записи данных в поток используется оператор ш, а для чтения данных из потока — >>.

5.1. Класс *QTextStream*

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*(). Например:

```
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();
}
```

Методом *QTextStream::readAll*() можно считать сразу весь текстовый файл в строку. Например:

```
QFile file("myfile.txt");
QTextStream stream(&file);
QString str = stream.readAll();
```

Чтобы записать текстовую информацию в файл, необходимо создать объект класса *QFile* и воспользоваться оператором <<. Перед записью можно провести необходимые преобразования строки. Например:

```
QFile file("file.txt");
QString str = "This is a test";
if (file.open(QIODevice::WriteOnly))
{
    QTextStream stream(&file);
    stream << str.toUpper(); //Запишет-THIS IS A TEST
    file.close();
    if (stream.status() != QTextStream::Ok)
    {
        qDebug() << "Ошибка записи файла";
    }
}
```

Класс *QTextStream* создавался для записи и чтения только текстовых данных, поэтому двоичные данные при записи будут искажены. Для чтения и записи двоичных данных без искажений следует пользоваться классом *QDataStream.*

5.2. Класс *QDataStream*

Класс *QDateStream* является гарантом того, что формат, в котором будут записаны данные, останется платформонезависимым и его можно будет считать и обработать на других платформах. Это делает класс незаменимым для обмена данными по сети с использованием сокетных соединений.

Формат данных, используемый *QDataStream*, в процессе разработки версии *Qt* претерпел множество изменений и продолжает изменяться. По этой причине этот класс знаком с различными типами версий, и для того чтобы заставить его использовать формат обмена, соответствующий определенной версии *Qt*, нужно вызвать метод *setVersion*(), передав ему

идентификатор версии. Текущая версия имеет идентификатор *Qt_4_2*.

Класс поддерживает большое количество типов данных, к которым относятся: *QByteArray*, *QFont*, *QImage*, *QMap*, *QPixmap*, *QString*, *QValueList* и *Variant*.

Следующий пример записывает в файл объект точки (*QPointF*), задающей позицию растрового изображение вместе с объектом растрового изображения (*QImage*):

```
QFile file("file.bin");
if(file.open(QIODevice::WriteOnly))
{
    QDataStream stream(&file);
    stream. setVersion(QDataStream::Qt_4_2);
    stream << QPointF(30, 30) << QImage("image.png");
    if(stream.status() != QDataStream::Ok)
    {
        qDebug() << "Ошибка записи";
    }
}
file.close();
```

Для чтения этих данных из файла нужно сделать следующее:

```
QPointF pt;
QImage img;
QFile file("file.bin");
if(file.open(QIODevice::ReadOnly))
{
    QDataStream stream(&file);
    stream.setVersion (QDataStream::Qt_4_2) ;
    stream >> pt >> img;
    if(stream.status() != QDataStream::Ok)
    {
        qDebug() << "Ошибка чтения файла";
    }
file.close();
}
```

6. Методические указания по выполнению работы

Лабораторную работу необходимо выполнить в следующей последовательности:

1. Изучите теоретические разделы 2-5, выполните на компьютере все приведенные в данных разделах примеры, скомпилируйте и запустите на исполнение полученные программы. В качестве дополнительных источников информации, воспользуйтесь указанными в перечне используемой литературы (раздел 7) пособиями, а также открытыми источниками в сети Интернет.
2. Самостоятельно выполните задание, описанное ниже в разделе 6.1.
3. Подготовьте исходный код программ в Qt Creator покажите его преподавателю.
4. В случае отсутствия видимых ошибок в коде, скомпилируйте программы

и запустите их на выполнение.

5. Получите оценку.

6.1. Индивидуальное задание

В п. 5.1 содержится описание двух программ, считывающих текстовую информацию из файла и записывающих ее в файл. По аналогии с приведенными программами создайте два приложения с GUI:

1. Первое должно иметь графический интерфейс, содержащий текстовое поле и кнопку «Write». Приложение должно создавать файл, и при нажатии кнопки «Write» записывать в него текст, введенный пользователем в текстовое поле интерфейса.
2. Второе приложение должно иметь графический интерфейс, содержащий текстовое поле и кнопку «Read». Оно должно открывать ранее созданный файл и при нажатии кнопки «Read» считать количество символов текстовой информации в файле и выводить данное число в текстовое поле интерфейса.

Если приведенное задание покажется Вам недостаточно трудоемким, доработайте второе приложение так, чтобы в текстовое поле интерфейса выводилось, например, третье слово из текста, записанного в файл. Создайте дополнительное текстовое поле, в которое будет записываться количество символов данного слова.

7. Рекомендуемая литература

1. Qt 5.3. Профессиональное программирование на C++: Наиболее полное руководство / М. Шлее. - СПб. : БХВ-Петербург, 2015. – 915 с.
2. Бланшет Ж., Саммерфилд М. Qt 4: программирование GUI на C++. Пер. с англ. 2-е изд., доп. – М.: КУДИЦ-ПРЕСС, 2008. – 736 с.
3. Процессы и потоки. [Электронный ресурс]. - Режим доступа: http://qt-doc.ru/qt-process.html (дата обращения: 2.12.2019).