Работа с Файлами и Каталогами. Потоки ввода / вывода.

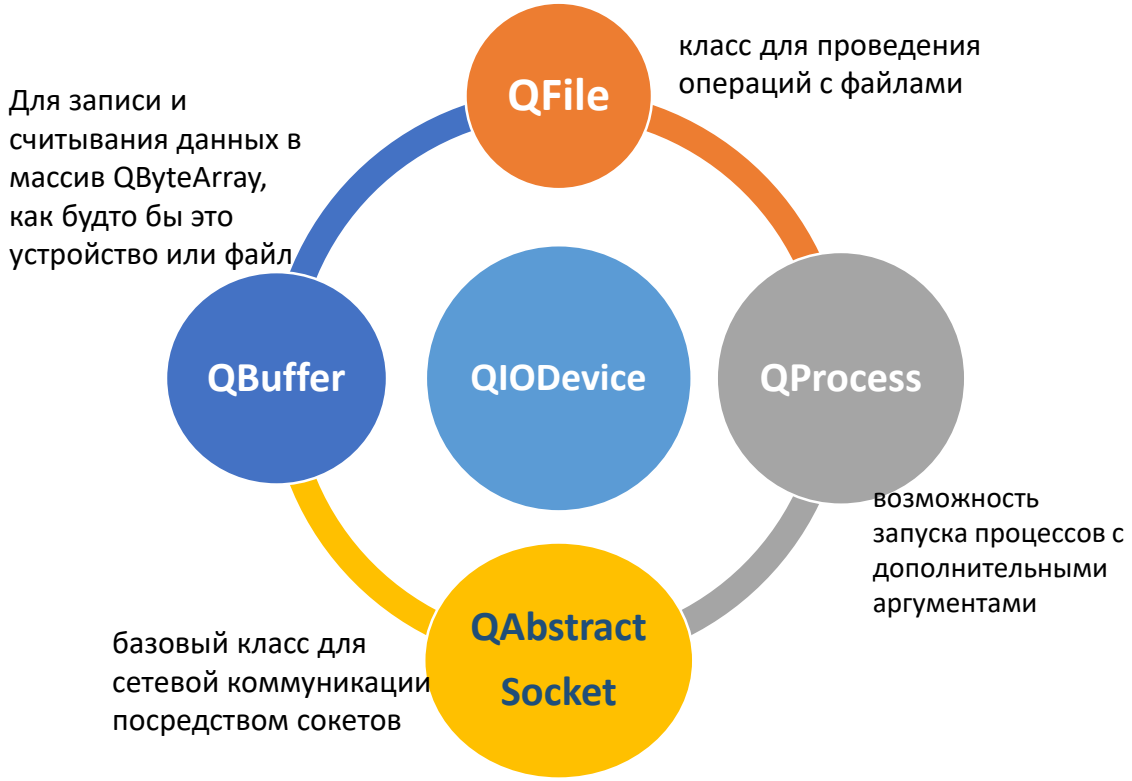
Лекция + практика

Основной функционал

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.

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.



Для работы с устройством его необходимо открыть в одном из режимов, определенных В < 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;

Для определения в каком из режимов было открыто устройство, нужно вызвать метод **openMode().**

Класс QIODevice

Основные методы:
read()/write() – считывание и запись данных;
readAll() - чтение всех данных сразу в *QByteArray*;
readLine() и getChar() – считывание строки или символа;
seek() - определен метод для смены текущего положения;
pos() - Получить значение текущего положения;
size() – размер данных устройства;
* только для прямого доступа к данным *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.

Для вывода 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;
}
```

Работа с файлами. Класс QFile

Класс *QFile* содержит методы для работы с файлами

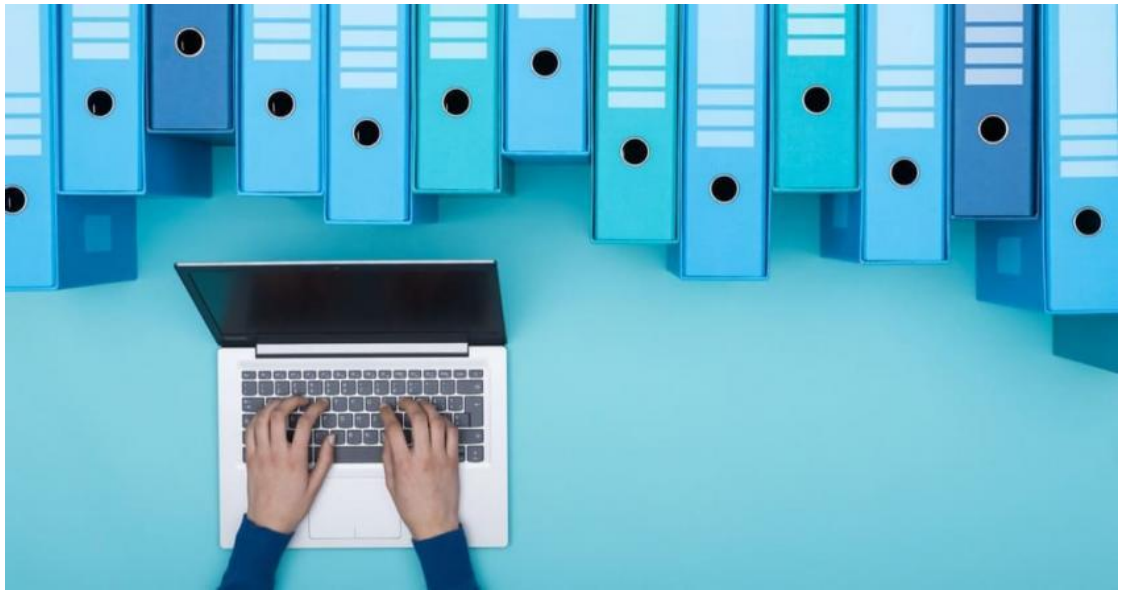
QIODevice::isOpen() – позволяет узнать , открыт файл или нет.
QFile:: close() – метод для закрытия файла с записью все данных буфера.
QFile::flush() – запись данных буфера без закрытия файла.
QFile::exists() – проверка на существование нужного вам файла;
QIODevice::read()/ QIODevice::write() - позволяют считывать и записывать файлы блоками.
QFile::remove() - статический метод для удаление файла

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

```
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);
}
```

Работа с каталогами. Класс QDir

Для представления директорий в платформо - независимой форме *Qt* предоставляет класс **QDir**.



QDir::current() — возвращает путь к директории приложения;
QDir::root () — возвращает root-директорию;
QDir::drives ()—возвращает указатель на список объектов класса *QFileinfo* с узловыми директориями (root). Для Windows это будут C:\, D:\ и т. д.;
QDir::home () — возвращает персональную директорию пользователя.
QDir::exists() – проверка существование директории.
QDir:: cd()/cdUp() - перемещаться по директориям
QDir:: makeAbsolute() – конвертация относит. пути в абс.
***QDir::mkdir()** – создание директории;
***QDir::rename()** – переименование директории;
***QDir::rmdir()** – удаление дир.;

*- возвращают true/false как результат своей работы

Класс *QDir* не предоставляет методов для определения текущего каталога приложения. Для определения, из какого каталога было запущенно приложение, то следует воспользоваться методом **QApplication::applicationDirPath(),** либо **QApplication::applicationFilePath(),-** путь к каталогу + имя приложения

Практика. Просмотр содержимого директории с помощью QDir

При помощи класса **QDir** можно получить содержимое указанной директории. При этом допускается применять различные фильтры, чтобы исключить из списка не интересующие вас файлы.
Методы класса:
entryList() –возвращает список имен элементов (*QStringList*)
entryInfoList() - информационный список (*QFileInfoList*).
count() – метод определения кол-ва элементов в дир.

Информация о файлах. Класс QFileInfo

Задача этого класса **QFileInfo** состоит в предоставлении информации о свойствах файла, например: имя, размер, время последнего изменения, права доступа и т. д.

Иногда необходимо убедиться, что исследуемый объект является каталогом, а не файлом и наоборот. Для этой цели существуют методы класс *QFileInfo*:
isFile() и isDir().
Так же, имеется метод **isSymLink()**, возвращающий true, если объект является *символьной ссылкой* (*symbolic link* или *shortcut* в ОС Windows).

```
//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();
```

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().**

Метод **size()** класса *QFileInf* о возвращает размер файла в байтах:

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

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.

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**.

```
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.

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.

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 **<<**, а для чтения данных из потока — **>>**.

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:

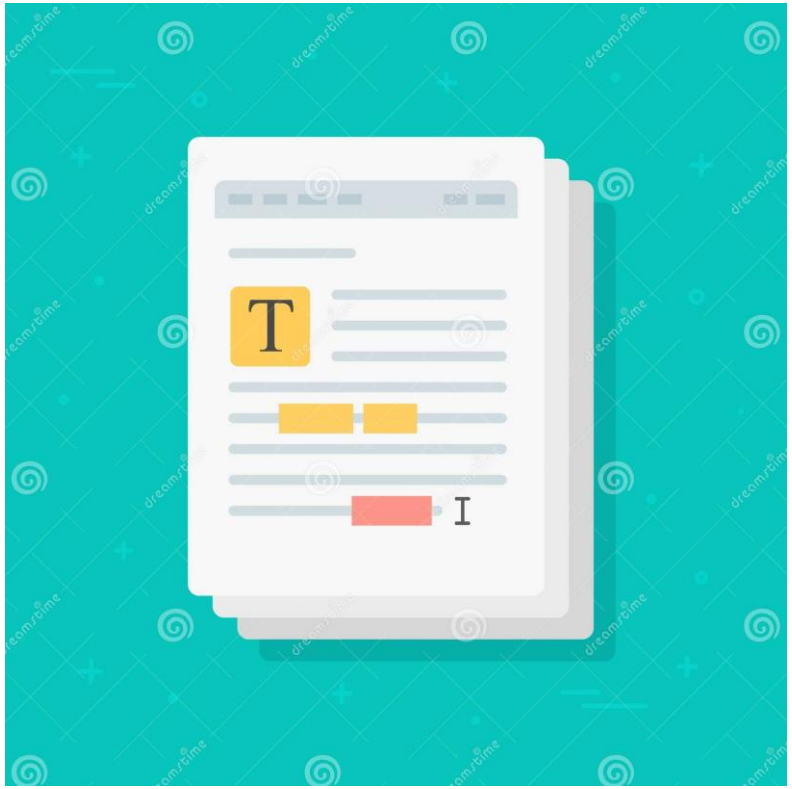
QTextStream:: readLine().

```
// Чтение файла
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();
```

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

Класс QTextStream

```
// Запись в файл
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**.

Класс QDataStream

Класс **QDateStream** является гарантом того, что формат, в котором будут записаны данные, останется платформо независимым 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.

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

```
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();
```

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

```
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();
}
```

JSON в Qt

Json текстовый формат обмена данных, использует расширение файла «.json. Может хранить сложные структуры. Часто используется для обмена данными между сервером и клиентом. Данный формат можно использовать для хранения настроек так как легко воспринимается на глаз. Важно помнить, что данный формат не предусматривает наличие комментариев в каком-либо виде.

Json основан на двух структурах данных:

- **Ключ** – З**начение** (к таким значениям можно отнести **запись, структура, словарь)**
- **Упорядоченный список значений**. К таким спискам можно отнести массив, вектор, список.

Строка – представляется так же, как и в С. Символы могут быть указаны с использованием escape -последовательности или записаны шестнадцатеричным кодом вкодировке *Unicode*.
Число – представляется так же, как и в С. Использует только десятичные системы счисления.

```
{
"hello": "world",
"t": true,
"n": null,
"i": 123,
"pi": 3.1416,
"a": [1, 2, 3, 4]
}
```

Объект – неупорядоченный набор пар «ключ : значение». Объект помещен в фигурные скобки «{ }». Каждое имя ключа и значение разделяются двоеточием «:» Пары «ключ : значение» разделяются запятой.
Массив – упорядоченная коллекция значений. Массив помещен в квадратные скобки «[ ]». Значения разделены запятой.
Значение – может быть *строкой* в двойных кавычках, *число*, *true, false, null*, *объектом* или *массивом*. Эти структуры могут быть вложенными.

Описание класса QJson

QJsonObject – Объект JSON
Список пар ключ-значение, где ключи являются уникальными строками, а значения представлены **QJsonValue**. QJsonObject может быть преобразован в/из **QVariantMap**. Узнать кол-во пар (ключ, значение) можно с помощью - **size(),** перебрать его содержимое, используя стандартный шаблон итератора C ++. **QJsonObject** можно конвертировать в текстовый JSON и из него через **QJsonDocument**.

QJsonArray - Массив JSON
Список значений. Этим списком можно манипулировать, вставляя и удаляя **QJsonValue** из массива. **QJsonArray** может быть преобразован в/из **QVariantList**. Можно запросить количество записей с помощью - size(), insert() и removeAt() и перебрать его содержимое, используя стандартный шаблон итератора C ++. **QJsonArray** является неявно разделяемым классом и делится данными с документом.

QJsonDocument
Это класс, который упаковывает полный документ JSON и может читать и записывать этот документ как из текстового представления в кодировке UTF-8, так и из собственного двоичного формата Qt. Документ JSON можно преобразовать из его текстового представления в **QJsonDocument** с помощью **JsonDocument::fromJson().** **toJson()** преобразует его обратно в текст. Парсер очень быстрый и эффективный и преобразует JSON в двоичное представление, используемое Qt. Допустимость проанализированного документа можно узнать с помощью **!IsNull().** Документ может быть запрошен относительно того, содержит ли он массив или объект, используя **isArray()** и **isObject()**. Массив или объект, содержащиеся в документе, можно извлечь с помощью **array()** или **object(),** а затем прочитать или манипулировать ими. Документ также может быть создан из сохраненного двоичного представления с использованием **fromBinaryData()** или **fromRawData()**.

QJsonParseError
Возвращает тип ошибки, которая произошла во время синтаксического анализа документа JSON.
Перечень ошибок:
QJsonParseError::NoErrorНет ошибок
QJsonParseError::UnterminatedObject Объект неправильно завершен закрывающей фигурной скобкой

QJsonValue
в JSON может быть одним из 6 основных типов.

QJsonObject::iterator
В отличии от **const_iterator** позволяет изменять значение (но не ключ), хранящееся под определенным ключом.

bool	**QJsonValue::Bool**
double	**QJsonValue::Double**
string	**QJsonValue::String**
array	**QJsonValue::Array**
object	**QJsonValue::Object**
null	**QJsonValue::Null**

Значение может представлять любой из типов данных. **QJsonValue** имеет один специальный флаг для представления неопределенных значений. Это можно сделать с помощью **isUndefined().** Тип значения можно запросить с помощью **type()** или методов доступа, таких как **isBool()**, **isString()** и так далее. Аналогично, значение может быть преобразовано в тип, сохраненный в нем, используя **toBool()**, **toString()** и так далее.

Запись в Json в Qt

```
{
   "Test1": "1",
   "Test2": 123.4,
   "Test3": 43,
   "Test4": true,
   "Test5": "1,2,3,4,5",
   "Test6": ["1", "2", "3" ]
}
```

```
QJsonObject recordObject;
recordObject.insert("Test1", QJsonValue::fromVariant("1"));
recordObject.insert("Test2", QJsonValue::fromVariant(123.4));
recordObject.insert("Test3", QJsonValue::fromVariant(43));
recordObject.insert("Test4", QJsonValue::fromVariant(true));
recordObject.insert("Test5",
QJsonValue::fromVariant(QByteArray("1,2,3,4,5")));

    QJsonArray numbersArray;
    numbersArray.push_back("1");
    numbersArray.push_back("2");
    numbersArray.push_back("3");
    recordObject.insert("Test6",numbersArray);

    QJsonDocument doc(recordObject);
    QString jsonString = doc.toJson(QJsonDocument::Indented);

    QFile file; //Записываем данные в файл
    file.setFileName("test1.json");
    file.open(QIODevice::WriteOnly | QIODevice::Text);
    QTextStream stream( &file );
    stream << jsonString;
    file.close();
```

Чтение файла Json в Qt

```
{
 "test": {
        "key1":"message1",
        "key2": "message2"
        },
"hello": "world",
 "test2":  {
             "a": [1,2,3],
             "t": true,
             "test3": {
                         "i": 123,
                         "pi": 3.14
                        }
            }
}
```

```
#include <QJsonObject>
#include <QJsonDocument>
#include <QJsonArray>
#include <QJsonValue>
#include <QJsonParseError>          0
```

```
#include <QFile>
QString val;
QFile file;
file.setFileName("test_2.json");
file.open(QIODevice::ReadOnly | QIODevice::Text);
val = file.readAll();
file.close();                                    1
```

```
QJsonParseError error;
QJsonDocument doc = QJsonDocument::fromJson(val.toUtf8(), &error);
qDebug() << "Error: " << error.errorString() << error.offset << error.error;
                                                                    2
```

```
if (doc.isObject())    {
      QJsonObject json = doc.object();  // док. превращается в объект
      QJsonValue test = json.value("test");  // считываем знач. Ключа «test»
      if (test.isObject())      {
         QJsonObject obj = test.toObject();
         QString msg1 = obj["key1"].toString();
         QString msg2 = obj["key2"].toString();
      }
      QString str = json["hello"].toString();
      QJsonValue test2 = json.value("test2");
      if (test2.isObject())   {
          QJsonObject obj2 = test2.toObject();
          QJsonArray ar = obj2["a"].toArray();
          bool t = obj2["t"].toBool();
          QJsonValue test3 = obj2.value("test3");
          if (test3.isObject()) {
             QJsonObject obj3 = test3.toObject();
             int i = obj3["i"].toInt();
             double pi = obj3["pi"].toDouble();
          }
      }
   }
```



```
{
 "test": {
       "key1":"message1",
       "key2": "message2"
       },
"hello": "world",
 "test2":  {
            "a": [1,2,3],
            "t": true,
            "test3": {
                       "i": 123,
                       "pi": 3.14
                      }
           }
}
```