GeekBrains

C++: разработка программ с графическим интерфейсом на Qt

Таймеры, процессы и потоки в Qt

Создание многопоточного приложения. Запуск процессов. Обмен сигналами в многопоточных приложениях

Дата и время. События таймера

Дата и время

Во фреймворке Qt есть классы **QDate**, **QTime** и **QDateTime**, которые предназначены для получения даты и времени и выполнения с ними различных операций (чаще всего требуется получение текущих даты и времени). Эти классы предоставляют методы для преобразования даты и времени в строку определенного формата, но также есть и методы для проведения обратного преобразования — из строки. Стоит отметить, что классы **QDate** и **QDateTime** начинают отсчитывать месяцы и дни от единицы, а не от нуля. Чтобы счет не сбивался, есть два выхода: добавить в массив пустую строку или из значения месяца вычесть единицу.

Получить текущее время можно так:

```
QTime time = QTime::currentTime();
QString tstr = "Время: " + QString::number(time.hour()) +
    ":" + QString::number(time.minute());
setWindowTitle(tstr);
```

Похожим образом получаем и текущую дату:

```
QDate date = QDate::currentDate();
   const QString month[] = {"", " января " , " февраля ", " марта ",
                            " апреля ", " мая ", " июня ",
                            " июля ", " августа ", " сентября ",
                            " октября ", " ноября ", " декабря ",
                           }; // Месяцы отсчитываются от 1, поэтому первый
                              // элемент - пустая строка
   QString dstr = QString::number(date.day()) + month[date.month()] +
QString::number(date.year());
   setWindowTitle(dstr);
```

Можно получить текущие дату и время одновременно, используя класс **QDateTime**:

```
QDateTime dateTime = QDateTime::currentDateTime();
   QTime time = dateTime.time();
   QDate date = dateTime.date();
```

События таймера

Один из важных компонентов для установки задержек, например перед показом анимации, — таймер. QTimer может работать в двух режимах:

- интервальный — срабатывает каждый раз по истечении установленного промежутка времени;
- таймер отложенного события — вызывается единожды через заданное время.

Основной сигнал таймера: **timeout()**.

```
QTimer *updDiskTime = new QTimer(this);
   connect(updDiskTime, SIGNAL(timeout()), this, SLOT(updateListDisks()));
   updDiskTime->start(5000); // Отложенное событие
```

Процессы. Запуск, работа и остановка

При создании сложного программного пакета его часто разделяют на несколько отдельных программ. В этом случае в пакете есть главное приложение, которое предоставляет графический интерфейс, и отдельные вспомогательные процессы (программы), поставляемые вместе с программным продуктом, либо сторонние, которые запускаются основным приложением для выполнения узкоспециализированных действий. Qt Creator — пример такого продукта: он запускает как собственные утилиты, так и компилятор с линковщиком. В одном из предыдущих уроков мы уже сталкивались с запуском процесса, а также команды терминала можно было выполнить и с помощью функции **system(QString)**. Процесс может генерировать несколько сигналов:

- **void finished(int exitCode)** — сигнал генерируется при завершении работы процесса с кодом;
- **void finished(int exitCode, QProcess::ExitStatus exitStatus)** — дополнительно передается статус завершения: было ли приложение завершено штатно или аварийно;
- **void error(QProcess::ProcessError error)** — сигнал при возникновении ошибки;
- **void readyReadStandardOutput(QPrivateSignal)** — сигнал при выводе процессом сообщений в консоль;
- **void stateChanged(QProcess::ProcessState state, QPrivateSignal)** — сигнал при изменении статуса приложения;
- **void started()** — сигнал при запуске процесса.

Процесс можно запускать с набором аргументов — так же, как, например, Qt Creator запускает утилиты **moc**, **uic** и другие.

Для примера запустим под ОС Windows программу Paint, под OS X — Photos, а в ОС семейства Linux — Pinta с картинкой, которую передадим через аргумент запуска процесса.

```
QString qf = QFileDialog::getOpenFileName(this, "Paint Picture", nullptr,
"Picture(*.png *.jpg, *.jpeg *.bmp)");
   // Linux
   if (qf.length() == 0) return;
   QString cmd = "pinta";
   // MS Windows
   if (QSysInfo::productType()=="windows") cmd = "mspaint";
   // macOS
   if (QSysInfo::productType()=="osx") cmd = "/Applications/Photos.app";
QProcess *p = new QProcess(this);
   QStringList list;
   if (QSysInfo::productType()=="windows")
   {
       QChar *ch = qf.data();
       for (int i = 0; i < qf.length(); i++)
       {
           if (ch[i] == '/') ch[i] = '\\';
       }
   }
   list << qf;
   p->start(cmd, list); // В консоли было бы так: mspaint <путь к файлу>
```

Допустим, одно приложение должно запустить другое через консоль, например: **qmake -project**. Тогда этот код можно записать следующим образом:

```
QString cmd = "qmake";
QProcess *p = new QProcess(this);
QStringList list;
list << "-project";
p->setArguments(list);
p->start(cmd);
```

Завершение процесса:

```
p->terminate(); // Завершение процесса
p->kill();      // Сигнал принудительного завершения процесса
```

Потоки. Многопоточность

Потоки

Поток — наименьшая единица обработки, исполнение которой может быть назначено ядром операционной системы. Существует группа задач, последовательность выполнения которых не повлияет на результат работы программного обеспеспечения, например обработка интерфейса или воспроизведение музыки, во время которого пользователь может менять настройки аудиоплеера, а сам плеер — искать музыкальные файлы на дисках компьютера. Поток может иметь собственный стек

или использовать общий стек с основной программой. Потоки могут выполняться как на одном и том же ядре с разделением по времени (мультиплексирование с разделением по времени, псевдомногопоточность, HyperThreading), либо на разных ядрах процессора параллельно. В Qt за поток отвечает класс **QThread**.

Поток выполняется с разным приоритетом по процессорному времени. Класс QThread подразумевает запуск потока с одним из восьми доступных флагов приоритета. Ниже они перечислены от наименьшего приоритета к наивысшему (за исключением восьмого, который устанавливает такой же приоритет, что и у запустившего его процесса):

- **QThread::IdlePriority;**
- **QThread::LowestPriority;**
- **QThread::LowPriority;**
- **QThread::NormalPriority;**
- **QThread::HighPriority;**
- **QThread::HighestPriority;**
- **QThread::TimeCriticalPriority;**
- **QThread::InheritPriority.**

Самый низкий приоритет у потока с флагом **QThread::IdlePriority**: он выполняется только тогда, когда другие потоки не запущены. А самый высокий — у потока с флагом **QThread::TimeCriticalPriority**, который выполняется как можно чаще по процессорному времени.

Многопоточность

Создадим программу для поиска файла на диске компьютера. Найденные файлы, подходящие под условие, будут выводиться в текстовом поле, при этом поиск можно будет остановить в любой момент. Сделаем упрощенную версию без поддержки символа *****. Создадим новый проект с главным окном на основе **QMainWindow**, а в нем — новый класс **ThreadFinder**. Наследуем методы от класса **QThread** и переопределим виртуальный метод **run()**, который и будет являтся телом потока. Основное назначение потока — выполнять код, заключенный в методе **run**. В примере тело потока будет выполнять цикл поиска файлов. Чтобы завершить поток при закрытии окна, до завершения сканирования файловой системы, добавим соответствующий флаг. Поток завершит свою работу в одном из двух случаев:

- закончится список каталогов для поиска;
- метод **findStop()** установит переменную **toWork** в **false.**

Заголовочный файл получится таким:

```
#ifndef THREADFINDER_H
#define THREADFINDER_H

```

```
#include <QObject>
#include <QThread>

class ThreadFinder: public QThread
{
   Q_OBJECT
public:
   explicit ThreadFinder(QString dir, QString File, QObject *parent = nullptr);
   void findStop() {toWork = false;} // Команда прекращения работы потока
protected:
   void run();
private:
   QString dir;
   QString file;
   bool toWork;                      // Завершение потока извне
signals:
   void stopedThread();              // Сигнал завершения поиска
   void writeFoundPath(QString);     // Вывод текущего каталога поиска
   void findFile(QString);           // Сообщение о найденном файле

};

#endif // THREADFINDER_H
```

В конструктор созданного класса добавлены два аргумента: каталог поиска (диск или каталог в зависимости от ОС) и строка из названия файла для поиска.

В цикле просматриваются каталоги и файлы, находящиесяв текущем каталоге, все каталоги добавляются в очередь вместе с полным путем до них и проверяется, совпадают ли имена файлов с именами, введенными пользователем. После этого просматриваемый каталог удаляется из очереди. Цикл будет выполняться до тех пор, пока очередь не опустеет или пока пользователь не закроет окно (в этом случае флаг **toWork** будет установлен в значение **false**).

```
#include "threadfinder.h"
#include <QDir>

ThreadFinder::ThreadFinder(QString dir, QString file, QObject *parent) : QThread
(parent)
{
   this->dir = dir;
   this->file = file;
   toWork = true;
}

void ThreadFinder::run()
{
   QStringList dirs = {dir};
   for (int i = 0; dirs.count() && toWork; ) // пока есть каталоги для поиска
файла
```

```
    {
        QDir search(dirs.at(i));
        emit writeFoundPath(dirs.at(i));
        search.setFilter(QDir::Hidden | QDir::Dirs | QDir::NoSymLinks);
        QStringList foundDirs = search.entryList();
        int amount = foundDirs.count();
        for (int j = 0; j < amount && toWork; j++)
        {
            QString newPath = dirs.at(i) + foundDirs[j] + "/";
            if (newPath.indexOf("/.") == -1) dirs << newPath; // Добавляем
                                                              // новый каталог
                                                              // для поиска
// Не забываем, что есть ссылки на каталоги текущий и возврата уровня: . и ..
        }
        search.setFilter(QDir::Hidden | QDir::Files | QDir::NoSymLinks);
        QStringList foundFiles = search.entryList();
        amount = foundFiles.count();
        for (int j = 0; j < amount && toWork; j++) // пока не просмотрены все
                                                   // файлы или пока флаг
                                                   // toWork не установлен в
                                                   // false
        {
            QString filename = foundFiles.at(j);
            if (filename.indexOf(file) != -1)
            {
                emit findFile(filename);
            }
        }
        dirs.removeAt(0);                          // Удаляем первый каталог
                                                   // поиска
    }
    emit stopedThread();                           // Сообщаем о завершении
                                                   // работы потока
}
```

Для контроля и управления потоком создадим новый класс **Controller**, базовый класс которого — **QObject**. Данный класс будет формировать результат для вывода пользователю и управлять потоком. Заголовок класса имеет следующий вид:

```
#ifndef CONTROLLER_H
#define CONTROLLER_H

#include <QObject>
#include <QSharedPointer>
#include "threadfinder.h"
class Controller : public QObject
{
    Q_OBJECT
public:
    explicit Controller(QObject *parent = nullptr);
```

```
    ~Controller();
    void startFind(QString dir, QString file);
private:
    QSharedPointer<ThreadFinder>findThread;
private:
    QString currentPath;
signals:
    void changFindPath(QString);      // Сообщаем о новом каталоге поиска и об
                                      // окончании работы
    void genPathOfFile(QString);      // Найденный файл для вывода готов
    void newFind();                   // Очистка поля вывода результата
public slots:
    void deleteThread();              // Удаляем поток при завершение работы
    void printCurrentPath(QString);   // Выводим текущий каталог поиска
    void genStringPathFile(QString); // Выводим найденный файл
    void finishThread();              // Информируем о завершении потока
};

#endif // CONTROLLER_H
```

При старте поиска создаем новый класс потока и подключаем сигналы потока к слотам.

```
#include "controller.h"

Controller::Controller(QObject *parent) : QObject(parent)
{

}

Controller::~Controller()
{
    if (findThread != nullptr)        // При завершении нужно остановить поток,
                                      // если он существует
    {
        if (findThread->isRunning()) // Проверяем, работает ли поток
        {
            findThread->findStop();  // Меняем флаг для завершения всех циклов
                                      // потока
            findThread->terminate(); // Ожидаем завершения потока
        }
        findThread.reset();           // Сбрасываем поток, вызывая деструктор
    }
}

void Controller::startFind(QString dir, QString file)
{
    if (findThread.get())
    {
        findThread->findStop();
        findThread->terminate();
        findThread.reset();
```

```
        return;
    }
    findThread = QSharedPointer<ThreadFinder>::create(dir, file);
    findThread->start(QThread::NormalPriority);
    connect(findThread.get(), SIGNAL(stopedThread()), this,
SLOT(deleteThread()));
    connect(findThread.get(), SIGNAL(writeFoundPath(QString)), this,
SLOT(printCurrentPath(QString)));
    connect(findThread.get(), SIGNAL(findFile(QString)), this,
SLOT(genStringPathFile(QString)));
    emit newFind();
}

void Controller::deleteThread()
{
    findThread.reset();
}
void Controller::printCurrentPath(QString path)
{
    currentPath = path;
    emit changFindPath(path);
}
void Controller::genStringPathFile(QString file)
{
    QString s = currentPath + file;     // Формируем полный путь к файлу
    emit genPathOfFile(s);              // Генерируем сигнал о готовности строки
                                        // файла
}

void Controller::finishThread()
{
    emit changFindPath(tr("FINISH. Find complete"));
}
```

Переходим к главному классу:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QPushButton>
#include <QLineEdit>
#include <QTextEdit>
#include <QGridLayout>
#include <QComboBox>
#include <QLabel>
#include "controller.h" // Подключаем класс-контроллер

class MainWindow : public QMainWindow
{
    Q_OBJECT
```

```
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    QLineEdit *searchEdit;
    QPushButton *startFindButton;
    QTextEdit *infoText;
    QComboBox *selDrive;
    Controller *controllerl;
    QLabel *statusLabel;
private slots:
    void findFileSlot();
    void changStatusLabel(QString);
    void printFindFile(QString);
};

#endif // MAINWINDOW_H
```

Информацию о текущем каталоге, в котором идет поиск, будем отображать в строке статуса через виджет **QLabel**

```
#include "mainwindow.h"
#include <QDir>
#include <QStatusBar>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    QWidget *centerWidget = new QWidget(this);
    setCentralWidget(centerWidget);
    QGridLayout *layout = new QGridLayout(this);
    centerWidget->setLayout(layout);
    searchEdit = new QLineEdit(this);
    layout->addWidget(searchEdit, 0, 0, 1, 3);
    //****************
    selDrive = new QComboBox(this);
    layout->addWidget(selDrive, 0, 3);
    if (QSysInfo::productType() == "windows") // Для ОС Windows
    {
        QFileInfoList infolist = QDir::drives();
        int amount = infolist.count();
        for (int i = 0; i < amount; i++)
        {
            selDrive->addItem(infolist.at(i).path());
        }

    }
    else {                                    // Для UNIX-подобных систем
        QStringList str = {"/", "/home/", "/mount/", "/opt/"};
```

```
        int amount = str.count();
        for (int i = 0; i < amount; i++)
        {
            selDrive->addItem(str.at(i));
        }
    }
    //****************
    startFindButton = new QPushButton(this);
    startFindButton->setText(tr("Find"));
    layout->addWidget(startFindButton, 0, 5);
    connect(startFindButton, SIGNAL(clicked()), this, SLOT(findFileSlot()));
    infoText = new QTextEdit(this);
    layout->addWidget(infoText, 1, 0, 10, 10);
    controllerl = new Controller(this);
    statusLabel = new QLabel(this);
    QStatusBar *statusBar = this->statusBar();
    statusBar->addWidget(statusLabel);
    connect(controllerl, SIGNAL(changFindPath(QString)), this,
SLOT(changStatusLabel(QString)));
    connect(controllerl, SIGNAL(genPathOfFile(QString)), this,
SLOT(printFindFile(QString)));
    connect(controllerl, SIGNAL((newFind())), infoText, SLOT(clear()));
}
```

Теперь добавим обработку сигналов от контроллера (класс **Controller**) потока:

```
MainWindow::~MainWindow()
{

}
void MainWindow::findFileSlot()
{
    QString linesearch = searchEdit->text();
    if (linesearch.length() == 0) return;
    controllerl->startFind(selDrive->currentText(), linesearch);
}

void MainWindow::changStatusLabel(QString line)
{
    statusLabel->setText(line);
}

void MainWindow::printFindFile(QString str)
{
    infoText->append(str);
}
```

Помимо классов Qt для реализации многопоточности можно использовать и другие библиотеки, например **OpenMP**.

Обмен сигналами и событиями

При многопоточном программировании возникает необходимость в обмене данными между потоками: например, в одном потоке создается растровое изображение, и его нужно переслать объекту другого потока. Как это можно сделать?

Каждый поток может иметь свой собственный цикл событий. Потоки могут взаимодействовать между собой и с основным потоком программы двумя способами: при помощи соединения сигналов и слотов или обмена событиями. В первом случае объекты разных потоков или виджеты с главным окном взаимодействуют при помощи метода **connect**. Этот способ знаком нам из предыдущих уроков.

Высылка событий — это еще один вариант связи между объектами. Для высылки событий есть два метода:

- **QCoreApplication::postEvent()**;
- **QCoreApplication::sendEvent()**.

Метод **postEvent()** обеспечивает потокобезопасную высылку событий, в отличие от метода **sendEvent()**. Это значит, что поток может высылать события другому потоку, который, в свою очередь, может ответить другим событием и т. д. Сами же события, обрабатываемые циклами событий потоков, будут принадлежать тем потокам, в которых они были созданы.

Создадим новый проект и новый класс **TestEvent**, с помощью которого будем формировать события. Заголовочный файл проекта содержит метод присвоения и чтения данных и метод генерации события. Хранить текст, передаваемый через создаваемое событие, будем в переменной **msg**. Можно добавить дополнительные параметры в зависимости от поставленной задачи.

```
#ifndef TESTEVENT_H
#define TESTEVENT_H
#include <QEvent>
#include <QString>

class TestEvent : public QEvent
{
public:
   enum {TypeEvent = User + 1};                    // Создаем новый тип события
   TestEvent();
   QString textMsg() const {return msg;}           // Метод чтения
   void setMsg(QString msg) {this->msg = msg;}     // Метод установки
private:
   QString msg;
};

#endif // TESTEVENT_H

// Код в testevent.cpp
```

```
#include "testevent.h"

TestEvent::TestEvent() : QEvent ((Type)TypeEvent)  // Создадим собственный тип
                                                   // события
{

}
```

Создадим класс потока, который будет искать и выводить все графические файлы:

```
#ifndef DISKVIEWER_H
#define DISKVIEWER_H
#include <QThread>
#include <QObject>

class DiskViewer : public QThread
{
public:
   DiskViewer(QObject *obj);
   void stopWork() {isWork = false;}
private:
   bool isWork;
   QStringList dirlist;
protected:
   void run() override;
};

#endif // DISKVIEWER_H
```

Добавляем вспомогательный метод и переменную для завершения работы потока:

```
#include "diskviewer.h"
#include <QDir>
#include<QApplication>
#include "testevent.h"

DiskViewer::DiskViewer(QObject *obj) : QThread(obj), isWork(true)
{
   dirlist.clear();
}

void DiskViewer::run()
{
   const QString imgFormat[] = {".png", ".bmp", ".jpeg", ".jpg"};
   const qint8 imgFmtCount = sizeof (imgFormat) / sizeof (imgFormat[0]);
   if(QSysInfo::productType() == "windows")
   {
       QFileInfoList drivers = QDir::drives();
       int amount = drivers.length();
```

```
        for(int i = 0; i < amount; i++)
        {
            dirlist << drivers[i].path();
        }
    } else dirlist << "/";
    for (;isWork && dirlist.length() > 0;) {
        QDir dir(dirlist.at(0));
        QStringList ldir = dir.entryList(QDir::Dirs | QDir::Hidden);
        int amount = ldir.length();
        for (int i = 0; i < amount; ++i) {
            dirlist << dirlist.at(0) + ldir.at(i) + "/";
        }
        QStringList imgs = dir.entryList(QDir::Files | QDir::Hidden);
        amount = imgs.length();
        for (int i = 0; i < amount; ++i) {
            for (int j = 0; j < imgFmtCount; j++)
            {
                if (imgs.at(i).indexOf(imgFormat[j], imgs.at(i).length() -
imgFormat[j].length())!= -1)
                {
                    QString l = dirlist.at(0) + imgs.at(i);
                    TestEvent *te = new TestEvent();        // Создаем событие
                    te->setMsg(l);                          // Строка пути к
                                                            // файлу
                    QApplication::postEvent(parent(), te); // Отправляем
                                                            // событие
                                                            // родительскому
                                                            // объекту
                }
            }
        }
        dirlist.removeAt(0);
    };
}
```

Вспомним пример кода, [на котором мы знакомились с потоками](). В этом примере мы также просматриваем все каталоги и файлы, но проверяем расширения файлов, а не названия. При нахождении файла с нужным расширением (графического файла) формируем строку полного имени файла (путь и название), которую отправляем через метод **postEvent** нужному объекту для обработки созданного события.

На главную форму добавим виджет **QTextEdit** и сделаем его центральным виджетом. Код обработки события, созданного объектом класса **TestEvent**, будет выглядеть так:

```
//mainwindow.h
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
```

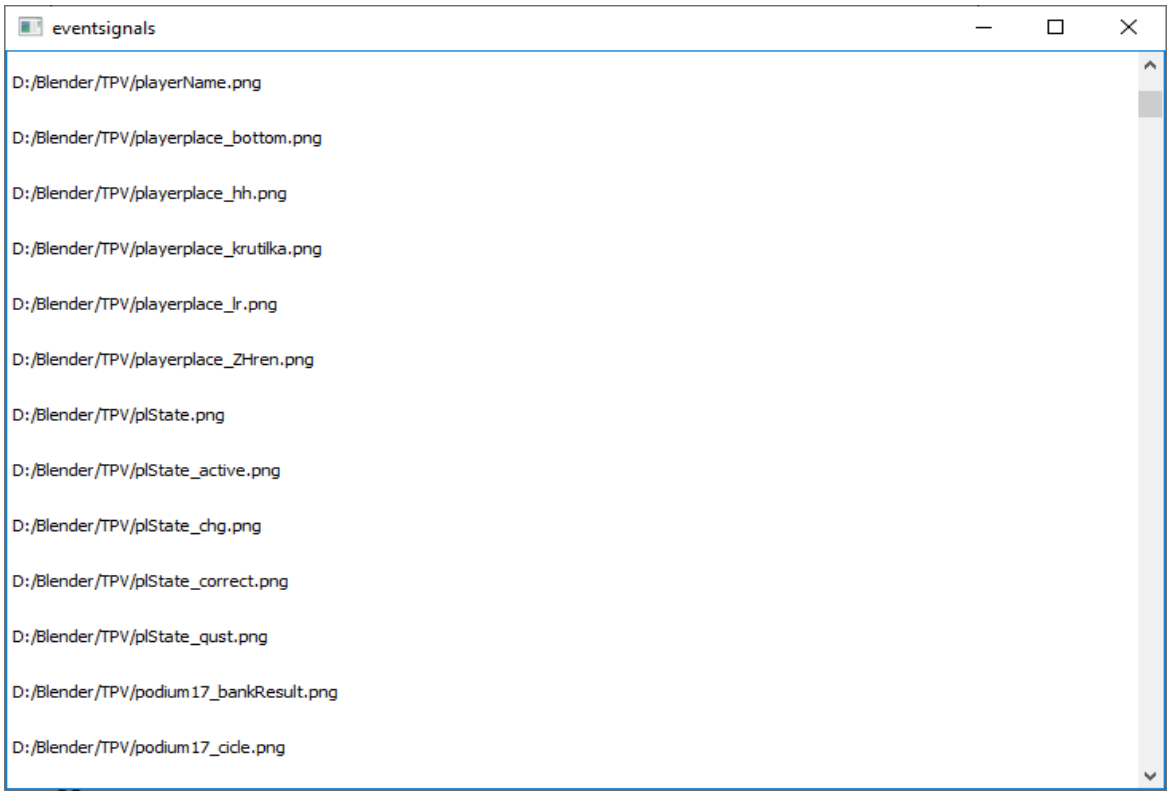
```
#include <QTextEdit>

class MainWindow : public QMainWindow
{
   Q_OBJECT

public:
   MainWindow(QWidget *parent = 0);
   ~MainWindow();
private:
   QTextEdit *textEdit;
protected:
   void customEvent(QEvent* pe) override; // Переопределяем событие
};

#endif // MAINWINDOW_H
//mainwindow.cpp
#include "mainwindow.h"
#include "testevent.h"
#include <QApplication>

MainWindow::MainWindow(QWidget *parent)
   : QMainWindow(parent)
{
   textEdit = new QTextEdit(this);
   setCentralWidget(textEdit);
}

MainWindow::~MainWindow()
{

}

void MainWindow::customEvent(QEvent* pe)
{
   if (pe->type() == TestEvent::TypeEvent) // Проверяем событие на тип
   {
       textEdit->insertHtml("<br><p>" + static_cast<TestEvent*>(pe)->textMsg() +
"</p>");                                  // Выводим строку с путем к файлу
   }
}
```

В главном файле **main.cpp** создадим поток, который будет формировать событие при нахождении графического файла (по расширению):

```
#include "mainwindow.h"
#include <QApplication>
#include "testevent.h"
#include "diskviewer.h"
```

```
int main(int argc, char *argv[])
{
   QApplication a(argc, argv);
   MainWindow w;
   DiskViewer diskviewer(&w);      // Создаем класс потока
   w.show();
   diskviewer.start();             // Запускаем поток
   int r = a.exec();               // Запускаем главный цикл программы
   diskviewer.stopWork();          // Останавливаем поток
   for(;diskviewer.isRunning();); // Ожидаем завершения работы потока
   return r;                       // Завершаем работу программы
}
```

Запустим приложение:



Синхронизация. Мьютексы

Синхронизация

При многопоточном программировании нередко возникает ситуация, когда нескольким потокам необходимо записать или считать данные, расположенные по одному адресу, то есть воспользоваться некоторыми общими ресурсами. Классы **QMutex**, **QReadWriteLock**, **QSemaphore** и **QWaitCondition** предоставляют средства синхронизации потоков. Хотя основная идея потоков состоит в том, чтобы сделать их настолько параллельными, насколько это возможно, бывают моменты, когда поток должен остановить выполнение текущих операций и подождать другие потоки. Например, если два потока

одновременно пытаются получить доступ к одной глобальной переменной, то результат обычно не определен. Принцип синхронизации заключается в проверке свободной очереди на выполнение операции, блокировке очереди, выполнении операции и разблокировке очереди.

Доступ потоков к ресурсам можно разделить на два кейса:

1. Параллельный доступ к ресурсу (например, методу).
2. Разбиение задачи для выполнения параллельными потоками. К примеру, если нужно наложить фильтр на очень большое изображение, оно разделяется на несколько частей, и на каждую из них фильтр накладывается в отдельном потоке. Основной же поток запускает задачу и ждет выполнения ее частей.

Мьютексы

При программировании многопоточных приложений во избежание ситуаций, когда два и более потоков или процессов обращаются к одной и той же ячейке памяти, причем один (или несколько) — для чтения, а другой (другие) — для записи, используют мьютексы. Ячейка памяти или группа ячеек в этом случае называется критической секцией. Критическая секция — это участок кода, в котором поток получает доступ к ресурсу (например, переменной или методу), который доступен из других потоков. Мьютексы обеспечивают взаимоисключающий доступ к ресурсам, гарантируя, что критическая секция будет обрабатываться только одним потоком. Поток, владеющий мьютексом, обладает эксклюзивным правом на использование ресурса, защищенного этим мьютексом, и другой поток не может завладеть уже занятым мьютексом. В Qt за работу с мьютексами отвечает класс **QMutex**, но также допустимо использовать и кроссплатформенные решения языка C++ из библиотек **STL** и **BOOST**.

Принцип использования мьютекса:

1. Метод **lock()** блокирует очередь или сначала ожидает разблокировки очереди другим потоком, а затем уже блокирует ее.
2. Выполняются операции внутри критической секции, например сохранение байтовой последовательности в общую память.
3. Метод **unlock()** разблокирует очередь и делает ее доступной для других потоков.

По такому же принципу работают мьютексы библиотек **STL** и **BOOST**.

Мьютекс можно использовать не только для изменения переменной или массива в памяти либо при записи в файл, но и для ожидания завершения потока. Поправим код ранее рассмотренного примера: вместо цикла, ожидающего переключения флага **isRunning()** в состояние **false** используем мьютекс. Код класса потока станет таким:

```
DiskViewer::DiskViewer(QObject *obj, QMutex *mutex) : QThread(obj), isWork(true)
{
   this->mutex = mutex; // Присваиваем переменной указатель на созданный
```

```
                        // мьютекс
    dirlist.clear();
}
void DiskViewer::run()
{
    mutex->lock();       // Блокируем очередь
    const QString imgFormat[] = {".png", ".bmp", ".jpeg", ".jpg"};
    const qint8 imgFmtCount = sizeof (imgFormat) / sizeof (imgFormat[0]);
    if(QSysInfo::productType() == "windows")
    {
        QFileInfoList drivers = QDir::drives();
        int amount = drivers.length();
        for(int i = 0; i < amount; i++)
        {
            dirlist << drivers[i].path();
        }
    }else dirlist << "/";
    for (;isWork && dirlist.length() > 0;) {
        QDir dir(dirlist.at(0));
        QStringList ldir = dir.entryList(QDir::Dirs | QDir::Hidden);
        int amount = ldir.length();
        for (int i = 0; i < amount; ++i) {
            dirlist << dirlist.at(0) + ldir.at(i) + "/";
        }
        QStringList imgs = dir.entryList(QDir::Files | QDir::Hidden);
        amount = imgs.length();
        for (int i = 0; i < amount; ++i) {
            for (int j = 0; j < imgFmtCount; j++)
            {
                if (imgs.at(i).indexOf(imgFormat[j], imgs.at(i).length() -
imgFormat[j].length())!= -1)
                {
                    QString l = dirlist.at(0) + imgs.at(i);
                    TestEvent *te = new TestEvent();
                    te->setMsg(l);
                    QApplication::postEvent(parent(), te);
                }
            }
        }
        dirlist.removeAt(0);
    };

    mutex->unlock();     // Разблокируем очередь
}
```

В этом случае **main.cpp** будет иметь следующий вид:

```
#include "mainwindow.h"
#include <QApplication>
#include "testevent.h"
#include "diskviewer.h"
```

```
int main(int argc, char *argv[])
{
   QApplication a(argc, argv);
   QMutex mutex; // Создаем мьютекс
   MainWindow w;
   DiskViewer diskviewer(&w, &mutex);
   w.show();
   diskviewer.start();
   int r = a.exec();
   diskviewer.stopWork();
   mutex.lock();
// Поток завершил свою работу
   mutex.unlock();
   return r;
}
```

Или такой:

```
#include "mainwindow.h"
#include <QApplication>
#include "testevent.h"
#include "diskviewer.h"

int main(int argc, char *argv[])
{
   QApplication a(argc, argv);
   QMutex mutex; // Создаем мьютекс
   MainWindow w;
   DiskViewer diskviewer(&w, &mutex);
   w.show();
   diskviewer.start();
   int r = a.exec();
   if (!mutex.tryLock()) {
       diskviewer.stopWork();
       mutex.lock();
   }
   mutex.unlock();
   return r;

}
```

Основное отличие метода **tryLock()** от **lock()** в том, что метод **tryLock()** не блокирует выполнение кода в потоке, где используется, а возвращает значение **false,** если мьютекс заблокирован другим потоком, и **true**, если этот метод успешно заблокировал мьютекс для текущего потока. В данном примере мы пытаемся заблокировать мьютекс, и если это не удается, выставляем флаг остановки потока и ожидаем, когда поток завершит свою работу.

Практическое задание

1. Добавить в текстовый редактор возможность вставлять дату и время в текст по позиции курсора.
2. В проект файловый менеджер добавить возможность поиска папки или файла по имени. Поиск должен выполнятся во вторичном потоке.

Дополнительные материалы

1. http://qt-doc.ru/klass-vremeni-qtime.html.
2. Форум с ответами на часто задаваемые вопросы. http://qaru.site/questions/tagged/qdatetime
3. Пример с мьютексом. http://fhoster.ru/bazaznaniy/qt/318-sinhronizatsiya-potokov-
4. QMath. https://doc.qt.io/qt-5/qtmath.html
5. Лицензия Qt. https://doc.qt.io/archives/qt-5.10/licenses-used-in-qt.html

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Официальная документация по QMutex. https://doc.qt.io/qt-5/qmutex.html
2. Официальная документация по QCoreApplication. https://doc.qt.io/qt-5/qcoreapplication.html
3. Официальная документация по QDate. https://doc.qt.io/qt-5/qdate.html
4. Официальная документация по QTime. https://doc-snapshots.qt.io/qt5-5.9/qtime.html
5. Официальная документация по QDateTime. https://doc-snapshots.qt.io/qt5-5.9/qtime.html