GeekBrains

C++: разработка программ с графическим интерфейсом на Qt

Разработка оконного интерфейса в Qt

Создание меню главного окна. Печать документов. Однодокументный и многодокументный интерфейс

Однодокументный и многодокументный интерфейс

Интерфейсы программного обеспечения, базирующиеся на работе с документами, разделяют на два типа: однодокументные и многодокументные. В однодокументных приложениях рабочая область одновременно является окном приложения, а это значит, что невозможно открыть в одном и том же приложении сразу два документа. Многодокументный интерфейс приложения представляет собой рабочую область (класса **QMdiArea**), в которой могут размещаться окна виджетов, что позволяет работать одновременно с большим количеством документов. Чтобы работать с несколькими документами в однодокументном приложении, нужно запустить несколько экземпляров программы. Создадим новый проект без формы, в заголовочном файле главного окна подключим заголовочный файл:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QMdiArea>

class MainWindow : public QMainWindow
{
   Q_OBJECT

public:
   MainWindow(QWidget *parent = 0);
   ~MainWindow();
private:
   QMdiArea *mdiArea;
};

#endif // MAINWINDOW_
```

Сам код для главного окна будет следующим:

```
#include "mainwindow.h"
#include <QGridLayout>
#include <QTextEdit>

MainWindow::MainWindow(QWidget *parent)
   : QMainWindow(parent)
{
   mdiArea = new QMdiArea(this);
   QWidget *centalW = new QWidget(this);
   setCentralWidget(centalW);
   QGridLayout *lay = new QGridLayout(this);
   centalW->setLayout(lay);
```

```
    lay->addWidget(mdiArea, 1, 0, 10, 1);
    mdiArea->addSubWindow(new QTextEdit(this));
    mdiArea->addSubWindow(new QTextEdit(this));
}
```

Создаем виджет, который будет центральным компонентом окна, затем создаем слой и прикрепляем его к центральному виджету. Создаем виджет **QMdiArea** (в Qt 4 и раннее использовался **QWorkSpace**). С помощью метода **QMdiArea::addSubWindow(QWidget *)** добавляем два виджета **QTextEdit**. Виджеты можно создать собственные, в виде текстового редактора с панелью управления форматированием и с другим функционалом или окна файловой системы для создания собственного файлового менеджера. Добавим кнопку и создадим для нее слот. При нажатии кнопки в активный документ запишется слово «Hello». Добавим кнопку в заголовочный файл **QMdiSubWindow**:

```
#include "mainwindow.h"
#include <QGridLayout>
#include <QPushButton>
#include <QMdiSubWindow>

MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    mdiArea = new QMdiArea(this);
    QWidget *centalW = new QWidget(this);
    setCentralWidget(centalW);
    QGridLayout *lay = new QGridLayout(this);
    centalW->setLayout(lay);
    lay->addWidget(mdiArea, 1, 0, 10, 5);
    QTextEdit *tedit = new QTextEdit(this);
    mdiArea->addSubWindow(tedit);
    mdiArea->addSubWindow(new QTextEdit(this));
    QPushButton *button = new QPushButton(this);
    lay->addWidget(button, 0,0,1,1);
    connect(button, SIGNAL(clicked()), this, SLOT(printToField()));
}
```

Не забудем добавить слот в объявление класса главного окна:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QMdiArea>
#include <QEvent>
#include <QTextEdit>
class MainWindow : public QMainWindow
{
    Q_OBJECT
```

```
public:
    MainWindow(QWidget *parent = 0);
    ~MainWindow();
private:
    QMdiArea *mdiArea;
    QTextEdit *curEdit;
private slots:
    void printToField();
};

#endif // MAINWINDOW_H
```

У виджета **QMdiArea** есть метод получения активного окна — **activeSubWindow()**:

```
void MainWindow::printToField()
{
    ((QTextEdit*)mdiArea->activeSubWindow()->widget())->setText("Hello");
}
// То же самое, но подробнее
void MainWindow::printToField()
{
    QMdiSubWindow *activSubWindow = mdiArea->activeSubWindow();
    QWidget *widg = activSubWindow->widget();
    QTextEdit *tedit = (QTextEdit*)widg;
    tedit->setText("Hello");
}
```

Данный пример слота полезен для реализации сохранения документа и для выполнения общих действий над активным окном (в том числе печати документа): нужно только добавить слот для обработки сигнала от главного меню, а в самом слоте получить от объекта **QMdiArea** указатель на активное окно.

Дочерние и родительские экраны. Подклассы (QDialog и др.)

Родительским экраном является любой виджет, если установлено свойство **parent = NULL**. В этом случае виджет представляет собой окно. Дочерним является окно, способное во время работы блокировать работу родительского окна. К ним относятся диалоговые окна (их также называют модальными). Подкласс **QDialog** служит для создания собственных диалоговых окон — например для заполнения формы регистрации или окон настройки приложения.

Главное окно. Панели инструментов. Строка состояния

Главное окно

Главное окно предоставляет структуру для создания пользовательского интерфейса приложения. Класс главного окна **QMainWindow** имеет свой собственный макет, к которому можно добавить **QToolBars**, **QDockWidgets**, **QMenuBar** и **QStatusBar**. Центральная область макета может быть занята любым виджетом. Центральным обычно является стандартный виджет Qt, такой как **QTextEdit** или **QGraphicsView**. Для расширенных приложений также могут быть использованы пользовательские виджеты. Центральный виджет устанавливается функцией **setCentralWidget()**. Главные окна имеют как однодокументный так и многодокументный интерфейс. В качестве центрального виджета для многодокументного интерфейса выбирается **QMdiArea**.

Панели инструментов

Панели инструментов реализованы в классе **QToolBar**. Добавить панель инструментов в главное окно можно с помощью **addToolBar()**.

Вы управляете начальной позицией панелей инструментов, назначая их определенной **Qt::ToolBarArea** (подробнее). Вы можете разделить область, вставив разрыв панели инструментов (аналогично разрыву строки при редактировании текста) с помощью **addToolBarBreak()** или **insertToolBarBreak()**. Также можно ограничить пользователю возможность размещения панелей инструментов с помощью **QToolBar::setAllowedAreas()** и **QToolBar::setMovable()**.

Размер значков панели инструментов можно получить с помощью **iconSize()**. Размеры зависят от платформы. Вы можете установить фиксированный размер функцией **setIconSize()**. Изменить внешний вид всех кнопок инструментов на панелях инструментов можно с помощью **setToolButtonStyle()**.

```
MainWindow::MainWindow(QWidget *parent)
   : QMainWindow(parent)
{
    QToolBar *tbar = addToolBar("ToolBar");
    QAction *acttool = tbar->addAction("Action панели инструментов");
}
```

Строка состояния

Строка состояния отображает маловажную информацию, например режим работы приложения, в нижней части окна. Используется она примерно так же, как панель инструментов. Для строки состояния используется метод **statusBar()**. Добавляем необходимые виджеты: например, если нужно добавить статический меняющийся текст, то добавляем **QLabel**. В объявлении класса окна

подключаем нужные классы виджетов и объявляем переменные для виджетов. Все делается также, как и для других компонентов экрана:

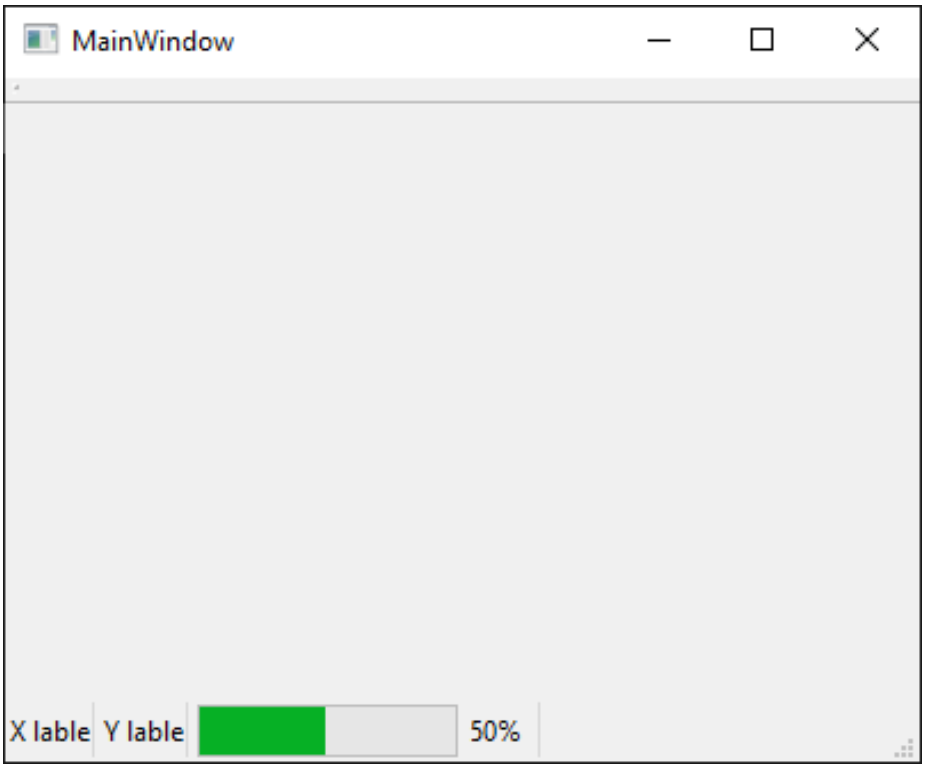
```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QLabel>
#include <QProgressBar>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
   Q_OBJECT

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

private:
   Ui::MainWindow *ui;
   QLabel *xlab, *ylab;
   QProgressBar* progrBar;
};

#endif // MAINWINDOW_H
```

Добавляем метки и прогресс-бар (примерно так же, как мы добавляли панель инструментов):

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

MainWindow::MainWindow(QWidget *parent) :
   QMainWindow(parent),
   ui(new Ui::MainWindow)
{
   ui->setupUi(this);
   QStatusBar *statusbar = this->statusBar();
   xlab = new QLabel(this);
   xlab->setText("X lable");
   ylab = new QLabel(this);
   ylab->setText("Y lable");
   progrBar = new QProgressBar(this);
   statusbar->addWidget(xlab);
   statusbar->addWidget(ylab);
   statusbar->addWidget(progrBar);
```

```
    progrBar->setValue(50);
}
```

Изменение текста или состояния индикации прогресс-бара выполняется привычным способом — вызовом методов для виджета:



Создание меню. Выпадающее меню. Контекстное меню

Рассмотрим виджеты, которые можно добавить в главное окно.

Создание меню

Меню во фреймворке реализовано в классе **QMenu**, а **QMainWindow** хранит их в **QMenuBar**. **QAction** добавляются в меню и отображаются как пункты меню. Для добавления нового меню в строку меню главного окна необходимо вызвать **menuBar()**, который возвращает **QMenuBar** для окна, а затем добавить меню с помощью **QMenuBar::addMenu()**. Элемент меню является классом **QAction**. Как и любой объект фреймворка, он содержит сигналы для подключения к слотам в программном продукте.

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent)
{
    QMenu *filemenu = menuBar()->addMenu("Файл");
    filemenu->addAction("Действие один");
```

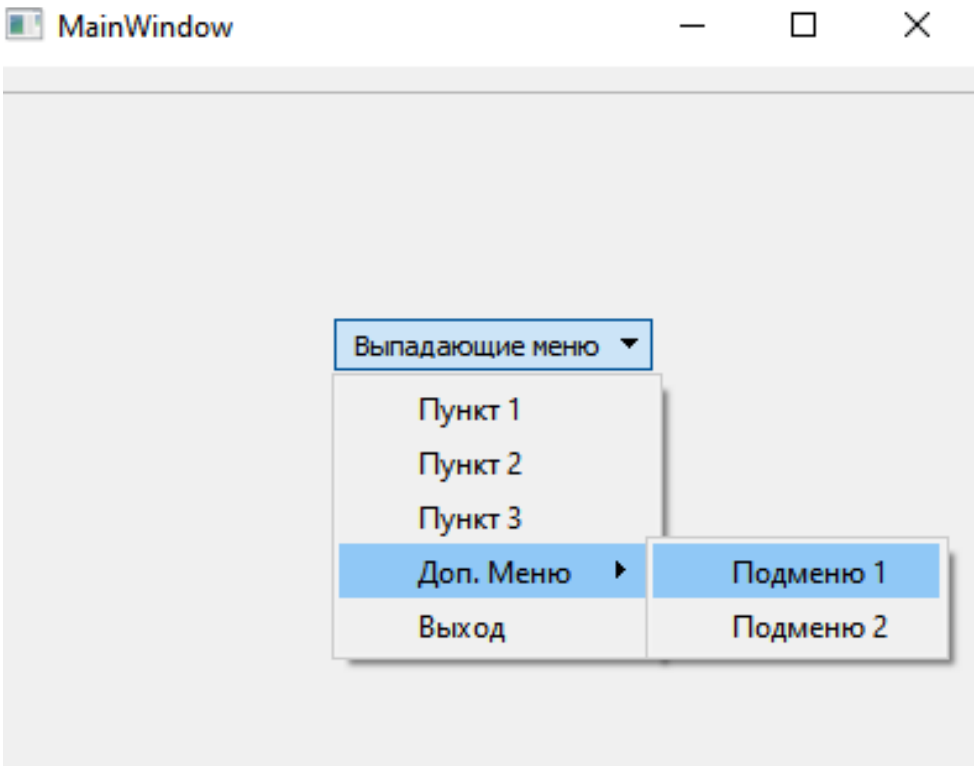
```
    filemenu->addAction("Действие два");
    QAction *quitAct = filemenu->addAction("В&ыход");
// Пример подключения элемента меню к слоту
    connect(quitAct, SIGNAL(triggered(bool)), this, SLOT(close()));
}
```

Выпадающее меню

Выпадающее меню создается так же, как и обычное, но прикрепляется к виджету кнопки — **QPushButton** или **QToolButton**:

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    menu = new QMenu(this);
    menu->addAction("Пункт 1");
    menu->addAction("Пункт 2");
    menu->addAction("Пункт 3");
    QMenu *submenu = menu->addMenu("Доп. меню");
    submenu->addAction("Подменю 1");
    submenu->addAction("Подменю 2");
    QAction *action = menu->addAction("Выход");
    ui->pushButton->setMenu(menu); // Прикрепляем меню к виджету кнопки
    connect(action, SIGNAL(triggered(bool)), this, SLOT(close()));
}
```

Получаем кнопку с выпадающим меню:



Контекстное меню

Наравне с обычным меню в программах часто используют контекстное меню. Команды контекстного меню относятся к тому объекту, над которым это меню было вызвано. Контекстное меню у виджета **QTextEdit** по умолчанию на английском языке. Мы создадим контекстное меню с пунктами на русском языке и сделаем его контекстным для виджета на базе **QTextEdit**.

Создадим новый класс C++, базовым выберем **QWidget**, сменим базовый класс на **QTextEdit**:

```
#ifndef RUSTEXTEDIT_H
#define RUSTEXTEDIT_H

#include <QMainWindow>
#include <QObject>
#include <QTextEdit>
#include <QContextMenuEvent>
#include <QMenu>

class RusTextEdit : public QTextEdit
{
   Q_OBJECT
public:
   explicit RusTextEdit(QWidget *parent = nullptr);
protected:
   void contextMenuEvent(QContextMenuEvent *event) override; // Перехватим
                                                             // событие
```

```
signals:

public slots:
    void copyText();
    void pastText();
private:
    QMenu *menu;                                                    // Для
                                                                    // контекстного
                                                                    // меню
};

#endif // RUSTEXTEDIT_H
```

Нужно перехватить вызов контекстного меню, а для этого необходимо переопределить соответствующее событие. Также мы создадим два слота для обработки копирования и вставки текста. Опишем конструктор:

```
#include "rustextedit.h"
#include <QApplication>
#include <QClipboard>

RusTextEdit::RusTextEdit(QWidget *parent) : QTextEdit(parent)
{
    menu = new QMenu(this);
    QAction *copyAction = menu->addAction("Копировать");
    QAction *pastAction = menu->addAction("Вставить");
    connect(copyAction, SIGNAL(triggered()), this, SLOT(copyText()));
// Подключаем слот к сигналу
    connect(pastAction, SIGNAL(triggered()), this, SLOT(pastText()));
}
```

Для использования буфера обмена понадобится подключить заголовочные файлы **QApplication** и **QClipboard**. Как можно увидеть из примера, обычное и выпадающее меню создаются по одному и тому же принципу:

```
void RusTextEdit::contextMenuEvent(QContextMenuEvent *event)
{
    menu->exec(event->globalPos());                 // Размещаем контекстное
                                                    // меню по экранным
                                                    // координатам
}

void RusTextEdit::copyText()
{
    QString str = this->textCursor().selectedText(); // Получаем выделенный
                                                     // текст
    qApp->clipboard()->setText(str);                 // Копируем текст
}
```

```
void RusTextEdit::pastText()
{
   QString str = qApp->clipboard()->text();          // Извлекаем из буфера
                                                     // обмена
   this->textCursor().insertText(str);               // Вставляем текст
                                                     // в позицию курсора
}
```

При обработке события вызываем созданное меню по глобальным координатам, используя методы объекта события. С помощью метода **QTextEdit::textCursor()** получаем доступ к объекту **QTextCursor**, который позволяет обрабатывать текст относительно каретки.

Диалоговые окна

Базовый класс диалогового окна — **QDialog**:

```
#include <QDialog>
class MyDialog : public QDialog
{
    Q_OBJECT
public:
  explicit MyDialog(QWidget *parent);
private:

}
```

Диалоговые окна могут вызываться двумя способами:

- метод **show()** запускает окна без блокировка главного окна;
- метод **exec()** запускает окна с блокировкой главного окна (режим модального окна).

Давайте создадим диалоговое окно для поиска строки в тексте. Для начала создадим проект Qt Widgets с базовым классом окна **QMainWindow**, после чего добавим новый класс C++:

```
#ifndef MYDIALOG_H
#define MYDIALOG_H

#include <QDialog>
#include <QWidget>
#include <QTextEdit>
#include <QGridLayout>
#include <QLineEdit>
#include <QLabel>
#include <QPushButton>
```

```
class MyDialog : public QDialog
{
   Q_OBJECT
   Q_PROPERTY(QTextEdit *textEdit WRITE setTextEdit)
public:
   explicit MyDialog(QWidget *parent = nullptr);
   virtual ~MyDialog();
   void setTextEdit(QTextEdit *);
signals:

public slots:
   void findPrev();                  // Сигнал от кнопки Поиск (до курсора
                                     // или после)
   void findNext();
private:
   QTextEdit *textEdit;
   QGridLayout *layout;
   QLabel *label;
   QLineEdit *lineEdit;              // Строка для ввода строки поиска
   QPushButton *findButtons[2];      // 2 кнопки поиска
signals:
                                     // Сигнал с указанием выделения текста
                                     // и новой позиции курсора
   void setCursorPos(int, int, int); // Начало, длина, новая позиция
};

#endif // MYDIALOG_H
```

Переходим к компоновке виджетов на форме. **QDialog** — виджет, поэтому мы можем разместить на нем слой компоновки:

```
#include "mydialog.h"

MyDialog::MyDialog(QWidget *parent) : QDialog(parent),
   textEdit(nullptr), layout(nullptr)
{
   setWindowTitle(tr("Find"));                    // Устанавливаем название
                                                  // диалогового окна
   setFixedSize(600, 100);
   layout = new QGridLayout();
   setLayout(layout);
   label = new QLabel(this);
   label->setText(tr("Find string"));
   layout->addWidget(label, 1, 1, 1, 4);
   lineEdit = new QLineEdit(this);
   layout->addWidget(lineEdit, 2, 1, 1, 7);
   findButtons[0] = new QPushButton(this);
   findButtons[1] = new QPushButton(this);
   findButtons[0]->setText(tr("Find previous"));
```

```
    findButtons[1]->setText(tr("Find next"));
    connect(findButtons[0], SIGNAL(clicked()), this, SLOT(findPrev()));
    connect(findButtons[1], SIGNAL(clicked()), this, SLOT(findNext()));
    layout->addWidget(findButtons[0], 4, 1, 1, 3);
    layout->addWidget(findButtons[1], 4, 5, 1, 3);
}

MyDialog::~MyDialog()
{
}

void MyDialog::setTextEdit(QTextEdit *textEdit)
{
    this->textEdit = textEdit;
}

void MyDialog::findPrev()
{
    QString str = lineEdit->text();
    int pos = textEdit->textCursor().position(); // Получаем позицию курсора
    QString txt = textEdit->toPlainText();       // Получаем текст
    QString p = txt.mid(0, pos);                 // Копируем текст от начала
                                                 // до курсора
    int last = -1;                               // Последний индекс с текстом
    int ps = 0;
    for (bool b = true;b;)
    {
        int index = p.indexOf(str, ps);             // Получаем индекс начала
                                                    // искомого текста
        if (index == -1) b = false;                 // Если текст не найден,
                                                    // завершаем поиск
        else {
            last = index;                           // Сохраняем последний
                                                    // найденный индекс на
                                                    // искомую строку
            ps = index + str.length();              // Смещаем поиск на длину
                                                    // искомой строки
                                                    // Продолжаем искать
                                                    // повторение строки
        }
    }
    close();                                        // Закрываем диалоговое
                                                    // окно
    if (last != -1)
    {
        emit setCursorPos(last, str.length(), last);// Посылаем позицию
                                                    // выделения и нового
                                                    // положения курсора
    }

}
void MyDialog::findNext()
```

```
{
   QString str = lineEdit->text();
   QString txt = textEdit->toPlainText();
   int pos = textEdit->textCursor().position();      // Получаем позицию
                                                     // курсора
   int index = txt.indexOf(str, pos);
   close();
   if (index != -1) {
       emit setCursorPos(index, str.length(), str.length() + index);
   }
}
```

Теперь остается добавить диалоговое окно к главной форме:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QTextEdit>
#include <QSharedPointer>
#include "mydialog.h"

class MainWindow : public QMainWindow
{
   Q_OBJECT

public:
   MainWindow(QWidget *parent = 0);
   ~MainWindow();
private:
   QTextEdit *textEdit;
   QSharedPointer<MyDialog>findDialog; // Для поискового диалога используем
                                       // "умный" указатель
private slots:
   void findText();                    // Слот вызова поиска
   void setNewPosition(int, int, int); // Слот на обработку выделения текста
                                       // и установки указателя
};

#endif // MAINWINDOW_H0
```

```
#include "mainwindow.h"
#include <QMenuBar>
#include <QMenu>
#include <QFile>
#include <QDataStream>

MainWindow::MainWindow(QWidget *parent)
   : QMainWindow(parent), textEdit(nullptr), findDialog(nullptr)
```

```
{
   textEdit = new QTextEdit(this);
   setCentralWidget(textEdit);
   QMenuBar *bar = menuBar();                                  // Создаем меню
   QMenu *menu = bar->addMenu(tr("Action"));                   // Создаем меню
   QAction *act = menu->addAction(tr("Find"));
// Подключаем сигналы кнопок к соответствующим слотам
   connect(act, SIGNAL(triggered()), this, SLOT(findText()));
   act = menu->addAction(tr("Quit"));
   connect(act, SIGNAL(triggered()), this, SLOT(close()));
}

MainWindow::~MainWindow()
{

}

void MainWindow::findText()
{
   if (!findDialog) {                                          // Если диалоговое
                                                               // окно не создано,
                                                               // создаем
       findDialog = QSharedPointer<MyDialog>::create(this);
       findDialog->setTextEdit(textEdit);                      // Передаем
                                                               // указатель на
                                                               // текстовое поле
       connect(findDialog.get(), SIGNAL(setCursorPos(int, int, int)), this,
SLOT(setNewPosition(int, int, int)));                          // Соединяем сигнал
                                                               // от диалогового
                                                               // окна к слоту
   }
   findDialog->exec();                                         // Запускаем диалог
                                                               // с блокировкой
                                                               // главного окна
}

void MainWindow::setNewPosition(int start, int lenght, int npos)
{                                                              // Если поиск
                                                               // прошел удачно,
                                                               // получаем
                                                               // координаты
                                                               // выделения текста
  QTextCursor tcursor = textEdit->textCursor();                // Копируем объект
                                                               // QTextCursor
  if (npos > start) {                                          // Если нужно
                                                               // сместить курсор
                                                               // вперед
      tcursor.setPosition(start, QTextCursor::MoveAnchor);     // Устанавливаем
                                                               // курсор в начало
                                                               // искомой строки
                                                               // по индексу
      tcursor.setPosition(npos, QTextCursor::KeepAnchor);      // Выделяем текст
```

```
                                                        // до новой
                                                        // позиции, которая
                                                        // равна сумме
                                                        // позиции начала
                                                        // искомой строки
                                                        // и ее длины
    }
    else {                                              // Если нужно
                                                        // сместить вперед
        tcursor.setPosition(start + lenght, QTextCursor::MoveAnchor);
                                                        // Берем позицию
                                                        // окончания
                                                        // искомой строки
        tcursor.setPosition(start, QTextCursor::KeepAnchor); // Переносим
                                                        // выделение на
                                                        // начало искомой
                                                        // строки
    }
    textEdit->setTextCursor(tcursor);                   // Устанавливаем
                                                        // новые настройки
                                                        // курсора
                                                        // QTextEdit
}
```

Добавим поддержку русской локализации, отредактируем файл *.pro:

```
QT       += core gui
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
TARGET = DialogExamples
TEMPLATE = app
DEFINES += QT_DEPRECATED_WARNINGS

CONFIG += c++11

SOURCES += \
        main.cpp \
        mainwindow.cpp \
        mydialog.cpp

HEADERS += \
        mainwindow.h \
        mydialog.h

TRANSLATIONS += texedit_ru.ts
CODECFORSRC     = UTF-8

qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target

RESOURCES += \
```

```
    localization.qrc
```

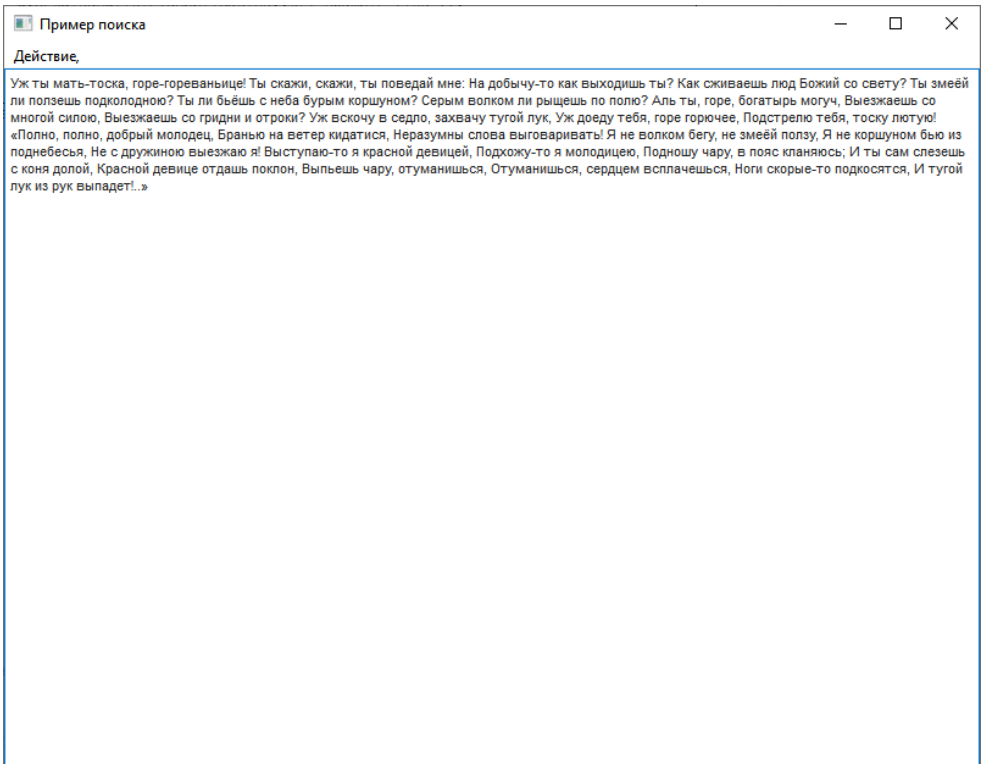
С помощью утилиты **lupdate** создадим файл локализации и отредактируем в приложении Qt Linguist. Откомпилированный файл добавим в файл ресурсов проекта **localization.qrc**.

```
<RCC>
    <qresource prefix="/">
        <file>texedit_ru.qm</file>
    </qresource>
</RCC>
```

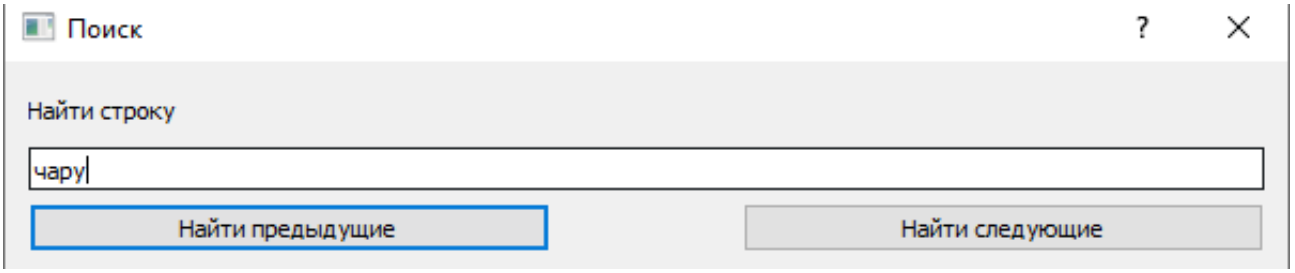
Остается добавить загрузку локализации в **main.cpp**:

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

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

    QTranslator translate;
    auto p =translate.load(":/texedit_" + QLocale::system().name());
    a.installTranslator(&translate);
    MainWindow w;
    w.setWindowTitle(QApplication::tr("Example find"));
    w.show();
    w.resize(800, 600);
    return a.exec();
}
```

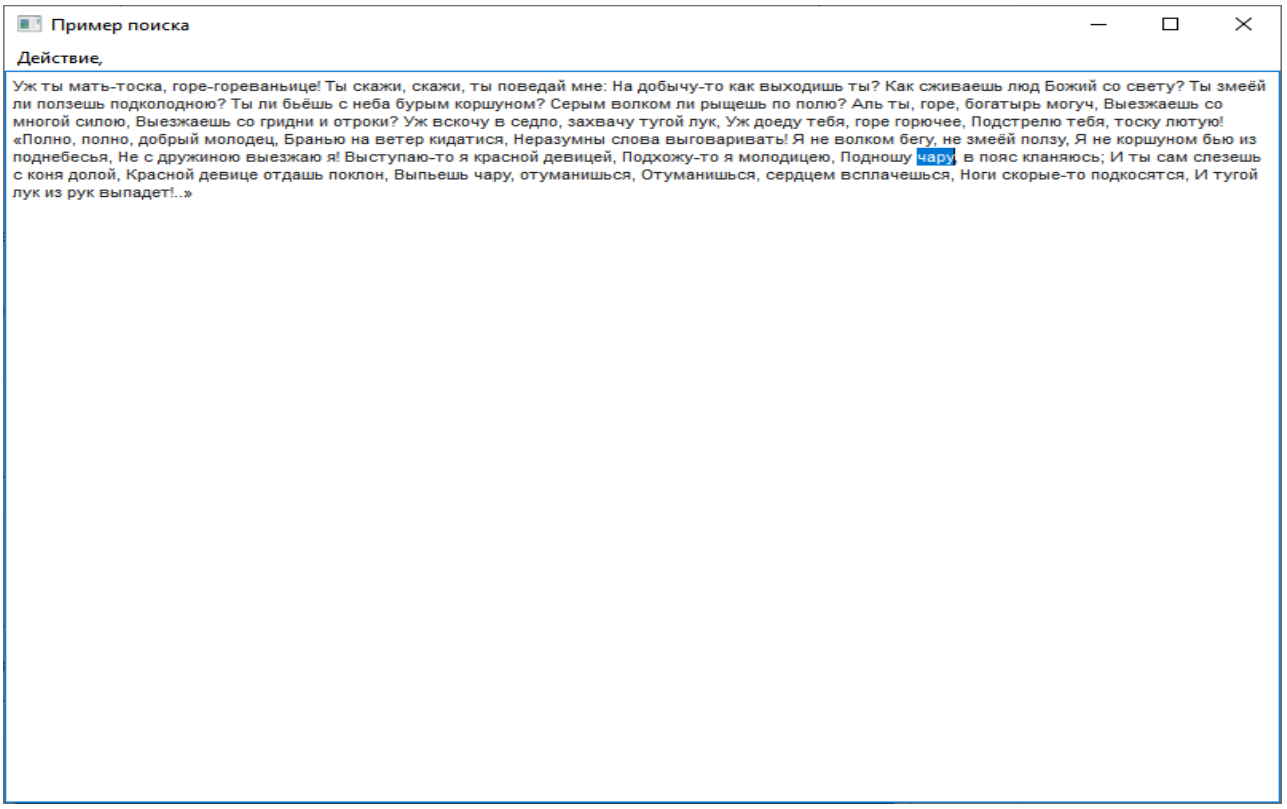
Запустим приложение и введем произвольный текст:



Запустим поиск (**Действия > Найти**), найдем слово «чару»:



Курсор был предварительно установлен в начало текста, поэтому нажмем кнопку «Найти следующие»:



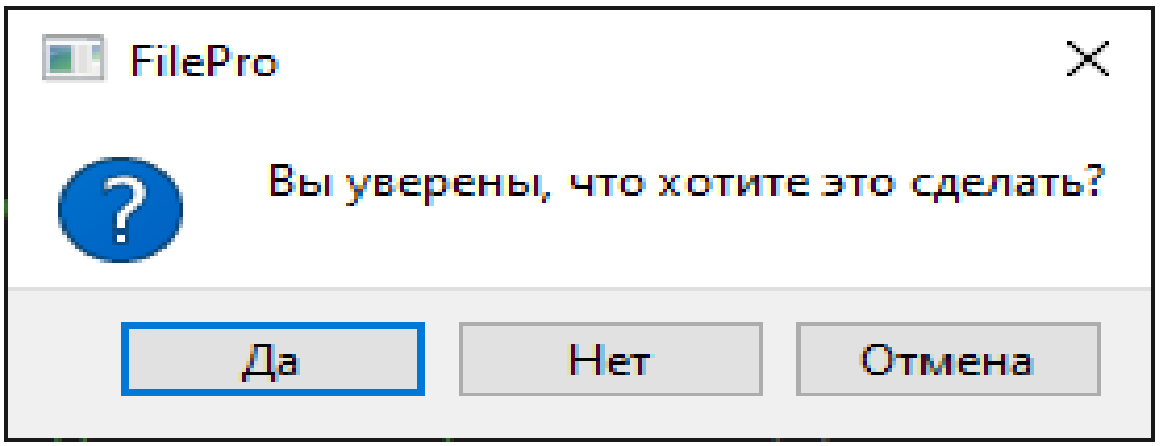
После нажатия диалоговое окно закроется, а в тексте будет выделено слово «чару».

Существуют и стандартные диалоговые окна: открытия/сохранения файлов, выбора каталога, настройки печати, выбора шрифта. Большинство стандартных классов имеют возможность вызвать диалоговые окно с помощью статического метода класса, пример которого мы рассматривали на предыдущем уроке.

В процессе выполнения программы возникает необходимость запросить у пользователя подтверждение, например, на сохранение документа при закрытии окна. Для диалоговых окон используется класс **QMessageBox**. Пример диалогового окна:

```
QMessageBox msgBox(this);
   msgBox.setText(tr("Вы уверены, что хотите это сделать?"));
   msgBox.setIcon(QMessageBox::Question);
   msgBox.addButton(tr("Да"), QMessageBox::YesRole);       // Добавлен 1-м -
                                                           // вернётся 0
   msgBox.addButton(tr("Нет"), QMessageBox::NoRole);       // Добавлен 2-м - 1
   msgBox.addButton(tr("Отмена"), QMessageBox::RejectRole);// Добавлен 3-м - 2
   int r = msgBox.exec();  // Значение r зависит от порядка добавления кнопок
   if (r == 0)
   {
       // Нажата кнопка "Да"
   }
   if (r == 1)
   {
       // Нажата кнопка "Нет"
```

```
    }
    if (r == 2)
    {
        // Нажата кнопка "Отмена"
    }
```



Возможные флаги для диалогового окна:

```
// Значки
enum Icon {
        // keep this in sync with QMessageDialogOptions::Icon
        NoIcon = 0,      // Без значка
        Information = 1, // Восклицательный знак
        Warning = 2,     // Знак предупреждения
        Critical = 3,    // Знак критической ошибки
        Question = 4     // Вопросительный знак, как в примере
    };

// Возможные возвращаемые значения при нажатии кнопки
 enum ButtonRole {
        // Влияет на порядок расположения кнопок в диалоговом окне
        InvalidRole = -1,
        AcceptRole,       // Принять
        RejectRole,       // Отказаться
        DestructiveRole,
        ActionRole,
        HelpRole,
        YesRole,          // Да
        NoRole,           // Нет
        ResetRole,        // Повторить
        ApplyRole,        // Применить

        NRoles
    };
```

Открытие, сохранение и вывод на печать документов

В современном мире документооборот происходит в цифровом виде, но порой возникает необходимость использовать бумажный вариант. Создадим редактор, который может отправить текст на печать принтеру. Для работы с печатью текста понадобится модуль **printsupport**. Добавим в файл проекта: QT += core gui printsupport.

Подключаем заголовочные файлы **QPrinter** и **QPrinterDialog**. Разместим на форме пустого проекта два объекта: **QPushButton** и **QPlainTextEdit**. По нажатию кнопки выводим диалоговое окно настройки печати:

```
void MainWindow::on_pushButton_clicked()
{
   QPrinter printer;
   QPrintDialog dlg(&printer, this);
   dlg.setWindowTitle("Print");
   if (dlg.exec() != QDialog::Accepted)
       return;
}
```

Печать происходит путем рисования на абстрактных страницах класса **QPrinter**. Для получения контекста рисования создадим объект класса **QPainter.** Текст может занимать несколько страниц и содержать абзацы. Для упрощение алгоритма разделим текст на абзацы. Для этой цели подойдет контейнер со строками. В Qt существует готовый класс-контейнер для строк — **QStringList** (подробнее в документации). Этот класс — полноценный контейнер, и в нем можно использовать итераторы. Перенесем текст из одной строки **QString** в список строк **QStringList**. Определять окончание строки в тексте будем по наличию символа переноса **\n**. Для добавления строки в список строк можно использовать оператор **<<**.

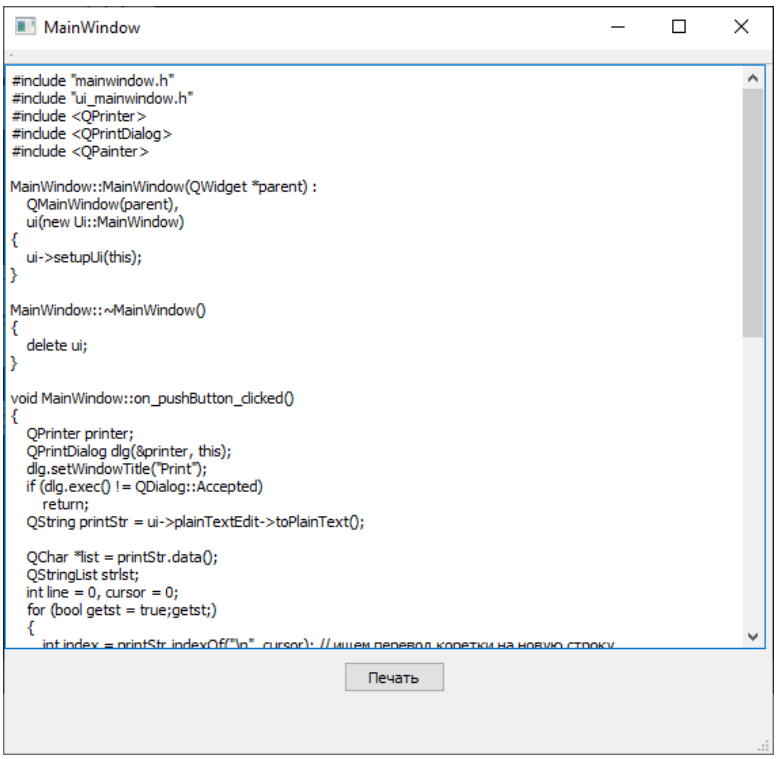
```
QString printStr = ui->plainTextEdit->toPlainText();

   QChar *list = printStr.data();
   QStringList strlst;
   int line = 0, cursor = 0;
   for (bool getst = true;getst;)
   {
       int index = printStr.indexOf("\n", cursor); // Ищем перевод каретки
                                                   // на новую строку
       QString s = "";
       if (index == -1)
       {
          getst = false;
          s.append(&list[cursor], printStr.length() - cursor);
```

```
        }
        else s.append(&list[cursor], index - cursor);
        cursor = index + 1;
        strlst << s;
    }
```

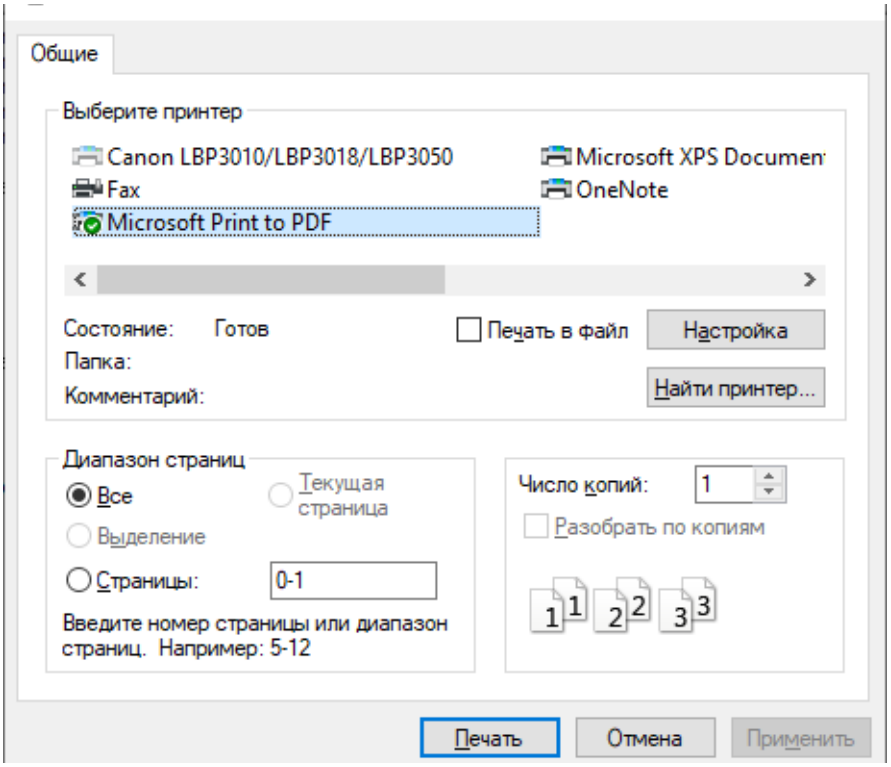
Начнем выводить текст на печать. Переменная **line** показывает позицию отрисовки текста, **cursor —** позицию маркера текста, с которого следует искать конец строки. Копируем кусок текста из исходного и помещаем строку в список строк. Если конец строки не найден, то это значит, что нам нужно скопировать остаток текста и остановить цикл выделения строк в тексте.

```
QPainter painter;
   painter.begin(&printer);
   int w = painter.window().width();
   int h = painter.window().height();
   int amount = strlst.count();
   QFont font = painter.font();
   QFontMetrics fmetrics(font);
   for (int i = 0; i < amount; i++)
   {
       QPointF pf;
       pf.setX(10);
       pf.setY(line);
       painter.drawText(pf, strlst.at(i));
       line += fmetrics.height();
       if (h - line <= fmetrics.height())
       {
           printer.newPage();
           line = 0;
       }

   }
   painter.end();
```

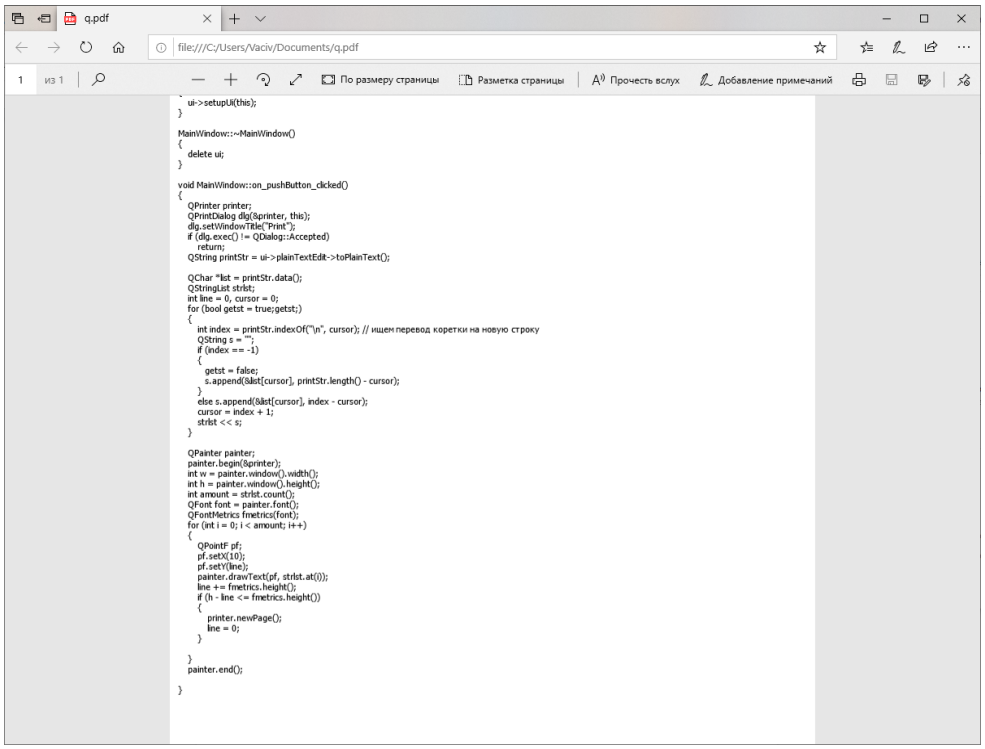
В объект **pf** заносим координаты верхнего левого угла текста, затем через метод объекта класса **QPainter** отрисовываем строку. Увеличиваем переменную с координатой по оси **y** на высоту текста, которую получаем через метод объекта класса **QFontMetrics::height()**. Методом **font()** класса **QPainter** получаем текущий шрифт: он нужен для определения размеров, которые понадобится для отрисовки текста. Если значение переменной **line** (метод **fmetrics.height()** возвращает высоту строки) достигло максимальной высоты страницы, это значит, что вся страница заполнена. Для создания новой страницы, на которой текст будет отрисовываться далее, вызываем метод **QPrinter::newPage()**. Создается чистое поле для отрисовки оставшегося текста, а страница, на которой отрисовка производилась ранее, отправляется на печать. Обнуляем **line** и продолжаем отрисовывать текст на новом листе. Запустим программу и введем в поле текст:



Нажмем кнопку **Печать**, и появится стандартное окно настройки печати:



Для экономии бумаги выберем печать в PDF. Далее укажем имя файла, в который будет напечатан документ:



Следует убедиться, что модуль **printsupport** и заголовочные файлы **QtPrintSupport/QPrinter** и **QtPrintSupport/QPrintDialog** подключены. Теперь мы можем отправить на печать содержимое виджета **QTextEdit**, вызвав метод **print(QPrinter *)**:

```
void MainWindow::on_pushButton_clicked()
{
   QPrinter printer;
   QPrintDialog dlg(&printer, this);
   dlg.setWindowTitle("Print");
   if (dlg.exec() != QDialog::Accepted)
       Return;
   textEdit->print(&printer);
}
```

Практическое задание

1. Добавить меню в текстовый редактор.
2. Добавить в текстовый редактор кнопку для вывода на печать.
3. * Добавить в текстовый редактор поддержку многодокументного интерфейса.

Дополнительные материалы

1. https://habr.com/ru/post/50765/.
2. http://doc.crossplatform.ru/qt/4.3.2/tutorial-t1.html.
3. https://www-formula.ru/2011-10-09-11-08-41.

4. https://doc.qt.io/qt-5/qtmath.html.
5. 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. https://doc.qt.io/qt-5/qtmodules.html.
2. Е. Р. Алексеев, Г. Г. Злобин, Д. А. Костюк, О. В. Чеснокова, А. С. Чмыхало. Программирование на языке C++ в среде Qt Creator.
3. https://doc.qt.io/qt-5/licenses-used-in-qt.html.