А.О. Семкин, К.В. Заичко, С.Н. Шарангович

ИНФОРМАТИКА

Руководство к лабораторной работе «Библиотека Qt. Создание главных окон программы»

2017

Министерство образования и науки Российской федерации

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ

(ТУСУР)

Кафедра сверхвысокочастотной и квантовой радиотехники
(СВЧиКР)

А.О. Семкин, К.В. Заичко, С.Н. Шарангович

ИНФОРМАТИКА

Руководство к лабораторной работе «Библиотека Qt. Создание главных окон программы»

2017

УДК 004.4+519.6

Рецензент:
профессор каф.СВЧиКР, А.Е. Мандель

А.О. Семкин, К.В. Заичко, С.Н. Шарангович

Информатика: Руководство к лабораторной работе «Библиотека Qt. Создание главных окон программы» / А.О. Семкин, К.В. Заичко, С.Н. Шарангович. – Томск: ТУСУР, 2017. – 20 с.

В данном руководстве изложены принципы работы в среде разработки *Qt Creator*. Приведено описание встроенного в *Qt* класса *QMainWindow*. Представлены методические материалы компьютерной лабораторной работы, посвященной созданию главных окон программ.

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

УДК 004.4+519.6

Оглавление

1. Цель работы

Целью данной работы является изучение принципов создания главных окон при помощи средств разработки Qt для создания законченного пользовательского интерфейса приложения, содержащего меню, панели инструментов, строку состояния и другие необходимые элементы управления.

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 рассмотрены принципы создания главных окон программ с графическим пользовательским интерфейсом.

3. Создание подкласса QMainWindow

Главные окна можно создавать при помощи *Qt Designer*, однако в рамках данной лабораторной работы будет изучен процесс создания при непосредственном программировании в виде подкласса *QMainWindow*.

Исходный код программы главного окна приложения «Электронная таблица» (в качестве примера) содержится в двух файлах: *mainwindow.h* и *mainwindow.cpp*. Сначала приведем заголовочный файл (*mainwindow.h*).

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
#include <QApplication>
#include <QAction>
#include <QMenu>
#include <QMenuBar>
#include <QStatusBar>
#include <QToolBar>
#include <QTableWidget>
#include <QLabel>
#include <QString>
#include <QStringList>
#include <QMessageBox>
#include <QFileDialog>
#include <QCloseEvent>

class MainWindow : public QMainWindow
{
Q_OBJECT

public:
    MainWindow();
protected:
```

```
    void closeEvent(QCloseEvent *event);
```

В приведенном фрагменте кода определяется класс *MainWindow* как подкласс *QMainWindow*. Он содержит макрос *Q_OBJECT*, поскольку имеет собственные сигналы и слоты.

Функция *closeEvent()* определена в *QWidget* как виртуальная функция; она автоматически вызывается при закрытии окна пользователем. Она переопределяется в *MainWindow* для того, чтобы можно было задать пользователю стандартный вопрос относительно возможности сохранения изменений («Do you want to save your changes?»).

```
private slots:
    void newFile();
    void open();
    bool save();
    bool saveAs();
```

Некоторые функции меню, как, например, File | New (Файл | Создать) или реализованы в *MainWindow* в виде закрытых слотов (см. приведенный выше фрагмент кода). Большинство слотов возвращают значение типа *void*, однако *save()* и *saveAs()* возвращают значение типа *bool*. Возвращаемое значение игнорируется при выполнении слота в ответ на сигнал, но при вызове слота в качестве функции можно воспользоваться возвращаемым значением, как это обычно делается при вызове любой обычной функции C++.

Для поддержки пользовательского интерфейса главному окну потребуется еще несколько закрытых слотов и закрытых функций.

```
    void openRecentFile();
    void updateStatusBar();
private:
    void createActions();
    void createMenus();
    void createContextMenu();
    void createToolBars();
    void createStatusBar();
    bool okToContinue();
    bool loadFile(const QString &fileName);
    bool saveFile(const QString &fileName);
    bool readFile(const QString &fileName);
    bool writeFile(const QString &fileName);
    void setCurrentFile(const QString &fileName);
    void updateRecentFileActions();
    QString strippedName(const QString &fullFileName);
```

Кроме этих закрытых слотов и закрытых функций в подклассе *MainWindow* имеется также много закрытых переменных. По мере их использования их назначение будет объяснено.

```
    QTableWidget *TableWidget;
    QLabel *columnLabel;
    QLabel *rowLabel;
    QStringList recentFiles;
    QString curFile;
```

```
    enum { MaxRecentFiles = 5 };
    QAction * recentFileActions[MaxRecentFiles];
    QAction *separatorAction;
    QMenu *fileMenu;
    QMenu *optionsMenu;
    QMenu *helpMenu;
    QToolBar *fileToolBar;
    QToolBar *optionsToolBar;
    QAction *newAction;
    QAction *openAction;
    QAction *saveAction;
    QAction *saveAsAction;
    QAction *exitAction;
    QAction *selectAllAction;
    QAction *showGridAction;
    QAction *aboutQtAction;
};

#endif // MAINWINDOW_H
```

Далее рассмотрена реализация этого подкласса (*mainwindow.cpp*).

```
#include "mainwindow.h"

MainWindow::MainWindow()
{
    TableWidget = new QTableWidget;
    this->setCentralWidget(TableWidget);
    TableWidget->setColumnCount(1000);
    TableWidget->setRowCount(1000);
    createActions();
    createMenus();
    createContextMenu();
    createToolBars();
    createStatusBar();
    setWindowIcon(QIcon(":/images/icon.png" ));
    setCurrentFile("");
}
```

Во фрагменте кода выше создается центральный виджет главного окна в виде объекта класса *QTableWidget*. Центральный виджет занимает среднюю часть главного окна (см. рис. 1). Устанавливается начальное количество строк и столбцов электронной таблицы (1000 шт.). Кроме этого вызываются закрытые функции *createActions()*, *createMenus()*, *createContextMenu()*, *createToolBars()* и *createStatusBar()* для построения главного окна в той его части, которая не занята центральным виджетом (рис. 1). Также вызывается закрытая функция *readSettings()* для чтения настроек, сохраненных в приложении.

В конце конструктора в качестве пиктограммы окна задается PNG-файл: icon.png. *Qt* поддерживает многие форматы графических файлов, включая BMP, GIF, JPEG, PNG, PNM, SVG, TIFF, XBM и XPM. Функция *QWidget::setWindowIcon()* устанавливает пиктограмму в левый верхний угол окна. К сожалению, не существует независимого от платформы способа установки пиктограммы приложения, отображаемого на рабочем столе компьютера.

Рис. 1 – Области главного окна QMainWindow

В приложениях с графическим пользовательским интерфейсом обычно используется много изображений. Существует много различных методов, предназначенных для работы приложения с изображениями. Наиболее распространенными являются следующие:

- хранение изображений в файлах и загрузка их во время выполнения приложения;
- включение файлов XPM в исходный код программы (это возможно, поскольку
- файлы XPM являются совместимыми с файлами исходного кода C++);
- использование механизма определения ресурсов, предусмотренного в Qt.

В рамках данной работы используется механизм определения ресурсов, поскольку он более удобен, чем загрузка файлов во время выполнения приложения, и он работает со всеми поддерживаемыми форматами файлов. Изображения хранятся в подкаталоге *images* исходного дерева.

Для применения системы ресурсов Qt необходимо создать файл ресурсов и добавить в файл *.pro* строку, которая задает этот файл ресурсов. В рамках данной работы используется файл ресурсов *mainwindow.qrc*, поэтому в файл *.pro* необходимо добавить следующую строку:

```
RESOURCES = mainwindow.qrc
```

Сам файл ресурсов имеет простой XML-формат. Ниже приведено его содержание.

```
<RCC>
    <qresource prefix="/">
        <file>images/icon.png</file>
        <file>images/new.png</file>
        <file>images/open.png</file>
        <file>images/save.png</file>
    </qresource>
  </RCC>
```

4. Создание меню и панелей инструментов

Большинство современных приложений с графическим пользовательским интерфейсом содержат меню, контекстное меню и панели инструментов. Меню позволяют пользователям исследовать возможности приложения и узнать новые способы работы, а контекстные меню и панели инструментов обеспечивают быстрый доступ к часто используемым функциям. На рис. 2 показаны примеры меню приложений.

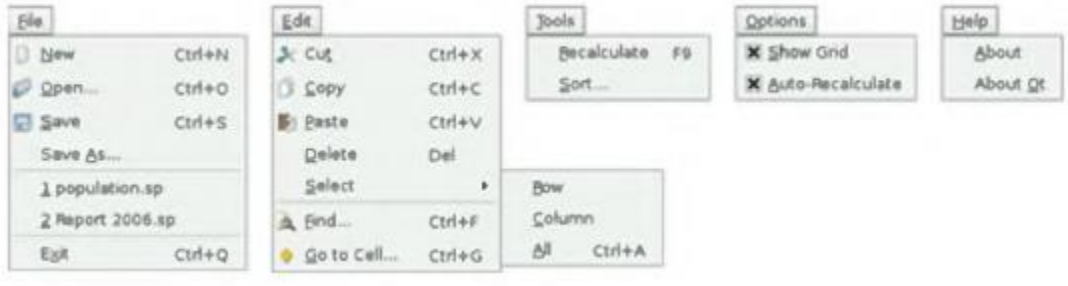

Рис. 2. Меню приложения

Использование понятия «действия» упрощает программирование меню и панелей инструментов при помощи средств разработки *Qt*. Элемент *action* (действие) можно добавлять к любому количеству меню и панелей инструментов. Создание в Qt меню и панелей инструментов разбивается на следующие этапы:

- создание и настройка действий;
- создание меню и добавление к ним действий;
- создание панелей инструментов и добавление к ним действий.

В примере приложения, рассматриваемого в данной работе, действия создаются в *createActions():*

```
void MainWindow::createActions()
{
newAction = new QAction(tr("&New"), this);
newAction->setIcon(QIcon(":/images/new.png"));
newAction->setShortcut(QKeySequence::New);
newAction->setStatusTip(tr("Create a new file"));
connect(newAction, SIGNAL(triggered()), this, SLOT(newFile()));
```

Действие *New* (создать) имеет клавишу быстрого выбора пункта меню (*New*), родительское окно (главное окно), пиктограмму, клавишу быстрого вызова команды и сообщение в строке состояния. С помощью соответствующего перечислимого значения *QKeySequence::StandardKey* можно сделать так, что *Qt* будет правильно использовать клавиши быстрого вызова той платформы, на которой запущено приложение. Далее к сигналу этого действия *triggered()* «подключается» закрытый слот главного окна *newFile()*. Это соединение гарантирует, что при выборе пользователем пункта меню File | New (файл |

создать), при нажатии им кнопки *New* на панели инструментов или при нажатии клавиш *Ctrl+N* будет вызван слот *newFile()*.

Создание действий *Open* (открыть), *Save* (сохранить), *Save As* (сохранить как) очень похоже на создание действия New:

```
openAction = new QAction(tr("&Open"), this);
openAction->setIcon(QIcon(":/images/open.png"));
openAction->setShortcut(QKeySequence::Open);
openAction->setStatusTip(tr("Open existing file"));
connect(openAction, SIGNAL(triggered()), this, SLOT(open()));
saveAction = new QAction(tr("&Save"), this);
saveAction->setIcon(QIcon(":/images/save.png"));
saveAction->setShortcut(QKeySequence::Save);
saveAction->setStatusTip(tr("Save current file"));
connect(saveAction, SIGNAL(triggered()), this, SLOT(save()));
saveAsAction = new QAction(tr("&Save as..."), this);
saveAsAction->setIcon(QIcon(":/images/saveAs.png"));
saveAsAction->setShortcut(QKeySequence::SaveAs);
saveAsAction->setStatusTip(tr("Save current file as..."));
connect(saveAsAction, SIGNAL(triggered()), this, SLOT(saveAs()));
```

Далее рассматривается создание строки «recently opened files» (недавно открытые файлы) меню *File*:

```
for (int i = 0; i < MaxRecentFiles; ++i)
    {
        recentFileActions[i] = new QAction(this);
        recentFileActions[i]->setVisible(false);
        connect( recentFileActions[i], SIGNAL(triggered()),
                 this, SLOT(openRecentFile()));
    }
```

В приведенном выше фрагменте кода заполняется действиями массив *recentFileActions*. Каждое действие скрыто и подключается к слоту *openRecentFi1e()*. Далее будет показано, как действия в списке недавно используемых файлов сделать видимыми, чтобы можно было ими воспользоваться.

```
exitAction = new QAction(tr("&Exit"), this);
exitAction->setShortcut( tr("Ctrl+Q"));
exitAction->setStatusTip(tr("Exit the application"));
connect(exitAction, SIGNAL(triggered()), this, SLOT(close()));
```

Действие *Exit* несколько отличается от действий, которые были описаны ранее. Не существует стандартизованной клавиши быстрого вызова для завершения работы приложения, так последовательность клавиш указывается явным образом. Еще одно отличие в том, что сигнал подключается к слоту *close()* окна, который предоставлен *Qt*. Далее рассмотрим действие *Select All* (выделить все), оно выглядит следующим образом:

```
selectAllAction = new QAction(tr("&Air"), this);
selectAllAction->setShortcut(QKeySequence::SelectAll);
selectAllAction->setStatusTip(tr("Select all the cells in the table"));
connect(selectAllAction, SIGNAL(triggered()),
        TableWidget, SLOT(selectAll())) ;
```

Слот *selectAll()* обеспечивается в *QAbstractItemView*, который является одним из базовых классов *QTableWidget*, поэтому нет необходимости его реализовывать. Далее рассмотрим действие *ShowGrid* (показать сетку) из меню *Options* (опции):

```
showGridAction = new QAction( tr(( "&Show Grid"), this);
showGridAction->setCheckable(true);
showGridAction->setChecked(TableWidget->showGrid());
showGridAction->setStatusTip(tr("Show or hide the table's grid"));
connect(showGridAction, SIGNAL(toggled( bool)),
          TableWidget, SLOT(setShowGrid(bool)));
```

Действие *ShowGrid* является включаемым. Включаемые действия обозначаются флажком в меню и реализуются как кнопки-переключатели на панели инструментов. Когда это действие включено, на компоненте *TableWidget* отображается сетка. После добавления этого действия к меню или панели инструментов пользователь сможет включать и выключать сетку.

Действия-переключатели, такие как *ShowGrid*, работают независимо. Кроме того, *Qt* обеспечивает возможность определения взаимоисключающих действий путем применения своего собственного класса *QActionGroup*.

```
aboutQtAction = new QAction(tr("About &Qt"), this);
aboutQtAction->setStatusTip(tr("Show the Qt library's About box" ));
connect(aboutQtAction, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
}
```

Для действия *AboutQt* (справка по средствам разработки *Qt*) используется слот *aboutQt()* объекта *QApplication*, который доступен через глобальную переменную *qApp*. Это всплывающее диалоговое окно показано на рис. 3

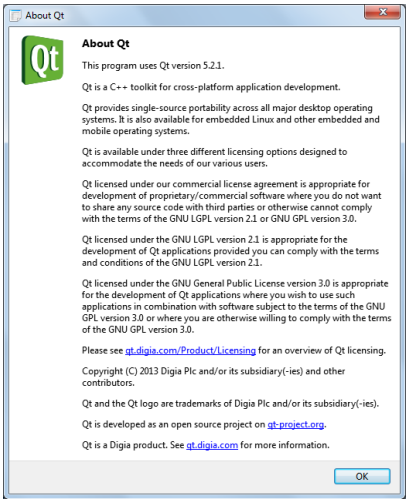

Рис. 3. Диалоговое окно About Qt

Действия созданы, далее можно перейти к построению системы меню с этими действиями.

```
void MainWindow::createMenus()
{
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(newAction);
    fileMenu->addAction(openAction);
    fileMenu->addAction(saveAction);
    fileMenu->addAction(saveAsAction);
    separatorAction = fileMenu->addSeparator();
    for (int i = 0; i < MaxRecentFiles; ++i)
        fileMenu->addAction(recentFileActions[i]);
    fileMenu->addSeparator();
    fileMenu->addAction(exitAction);
```

Сначала создается меню *File* и затем к нему добавляются действия *New*, *Open*, *Save* и *Save As*. Вставляется разделитель для визуального выделения группы взаимосвязанных пунктов меню, используется цикл *for* для добавления (первоначально скрытых) действий из массива *recentFileActions*, а в конце добавляется действие *exitAction*. Сохраняется указатель на один из разделителей. Это позволяет скрывать этот разделитель (если файлы не использовались) или показывать его, поскольку нет необходимости отображать два разделителя, когда между ними ничего нет.

Далее аналогичным образом создаются меню Options и Help:

```
    optionsMenu = menuBar()->addMenu(tr("&Options"));
    optionsMenu->addAction(showGridAction);
    menuBar()->addSeparator();
    helpMenu = menuBar()->addMenu(tr("&Help"));
    helpMenu->addAction(aboutQtAction);
}
```

Любой виджет в *Qt* может иметь связанный с ним список действий *QAction***.**

```
void MainWindow::createContextMenu()
{
    TableWidget->addAction(saveAction);
    TableWidget->addAction(showGridAction);
    TableWidget->addAction(aboutQtAction);
    TableWidget->setContextMenuPolicy(Qt::ActionsContextMenu);
}
```

Для обеспечения в приложении контекстного меню мы добавляем необходимые нам действия в виджет *TableWidget* и устанавливаем политику контекстного меню виджета на отображение контекстного меню с этими действиями.

Более сложный способ обеспечения контекстного меню заключается в переопределении функции *QWidget::contextMenuEvent()*, создании виджета *QMenu***,** заполнении его требуемыми действиями и вызове для него функции exec().

```
void MainWindow::createToolBars()
{
    fileToolBar = addToolBar(tr("&File"));
    fileToolBar->addAction(newAction);
    fileToolBar->addAction(openAction);
    fileToolBar->addAction(saveAction);
    optionsToolBar = addToolBar(tr("&Options"));
    optionsToolBar->addAction(showGridAction);
}
```

Создание панелей инструментов очень похоже на создание меню. В данной работе создается панель инструментов *File* и панель инструментов *Options***.** Как и меню, панель инструментов может иметь разделители.

5. Создание и настройка строки состояния

Обычно строка состояния содержит несколько индикаторов, например, номера столбца и строки текущей ячейки. Полоса состояния также используется для вывода подсказок и других временных сообщений.

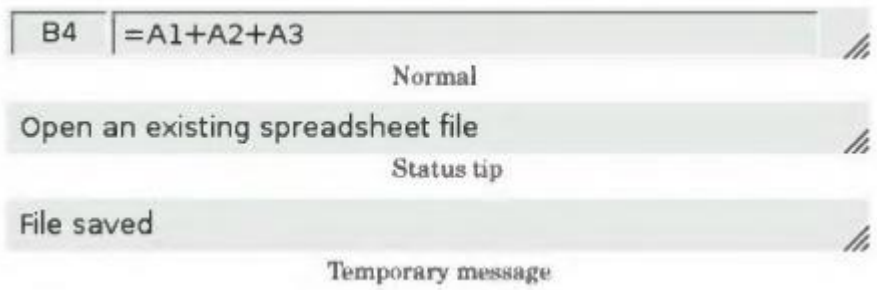

Рис. 4 – Пример строки состояния

Для создания строки состояния в конструкторе *MainWindow* вызывается функция *createStatusBar()*:

```
void MainWindow::createStatusBar()
{
    columnLabel = new QLabel("999");
    columnLabel->setAlignment(Qt::AlignHCenter);
    columnLabel->setMinimumSize(columnLabel->sizeHint());
    rowLabel = new QLabel("999");
    rowLabel->setAlignment(Qt::AlignHCenter);
    rowLabel->setMinimumSize(rowLabel->sizeHint());
    statusBar()->addWidget(columnLabel);
    statusBar()->addWidget(rowLabel);
    connect(TableWidget, SIGNAL(currentCellChanged(int, int, int, int)),
            this, SLOT(updateStatusBar()));
    updateStatusBar();
}
```

Функция *QMainWindow::statusBar()* возвращает указатель на строку состояния. (Строка состояния создается при первом вызове функции *statusBar*) В качестве индикаторов состояния просто используются текстовые метки *QLabel***,** текст которых изменяется по мере необходимости. В коде выше добавлен отступ для *columnLabel* и *rowLabel***,** чтобы указанный здесь текст

отображался с небольшим смещением от левого края. При добавлении текстовых меток *QLabel* в строку состояния они автоматически становятся дочерними по отношению к строке состояния. Когда *QStatusBar* располагает виджеты индикаторов, он постарается обеспечить «идеальный» размер виджетов, заданный функцией *QWidget::sizeHint()*, и затем растянет виджеты, которые допускают растяжение, заполняя дополнительное пространство. Идеальный размер виджета зависит от его содержания и будет сам изменяться по мере изменения содержания. Чтобы предотвратить постоянное изменение размера индикатора ячейки, его минимальный размер установлен на значение, достаточное для размещения в нем самого большого возможного текстового значения («999»), и добавлен еще немного пространства. Также параметр выравнивания установлен на значение *AlignHCenter* для выравнивания по центру текста в области индикатора. Перед завершением функции два сигнала *TableWidget* соединяются с двумя слотами главного окна *MainWindow: updateStatusBar()* и *tableWidgetModified()*.

```
void MainWindow::updateStatusBar()
{
    columnLabel->setNum(TableWidget->currentColumn());
    rowLabel->setNum(TableWidget->currentRow());
}
```

Слот *updateStatusBar()* обновляет индикаторы расположения ячейки.

6. Реализация меню File

Далее необходимо определить слоты и закрытые функции для обеспечения работы меню *File*.

```
void MainWindow::newFile()
{
    if (okToContinue())
    {
        TableWidget->clear();
        setCurrentFile("");
    }
}

bool MainWindow::okToContinue()
{
    if (isWindowModified())
    {
        int r = QMessageBox::warning(this, tr("TableWidget"),
                                     tr("the document has been modified.\n"
                                        "Do you want to save your changes?"),
                                     QMessageBox::Yes|QMessageBox::No|
                                     QMessageBox::Cancel);
        if(r==QMessageBox::Yes) return save();
        else if (r==QMessageBox::Cancel) return false;
    }
    return true;
}
```

Фрагмент кода выше содержит описание реализации слота *newFile()*, который очищает содержимое электронной таблицы с использованием встроенного метода *clear()*, а затем присваивает текущем файлу пустое имя. В реализации данного слота учитываются несохраненные изменения документа посредством использования другого слота *okToContinue()*, который контролирует, были ли внесены изменения в текущий документ (проверяя свойство *isWindowModified()*) и выводит предупреждающее сообщение пользователю, хочет ли он сохранить изменения.

QMessageBox предлагает много стандартных кнопок и автоматически пытается сделать одну из них выбранной по умолчанию (активируемой при нажатии клавиши *Enter*), а другую – кнопкой отмены (активируемой при нажатии клавиши *Esc*). Также существует возможность выбирать те кнопки, которые будут использоваться по умолчанию или будут кнопкой отмены. Кроме того, можно настраивать текст на кнопке.

Вызов функции *warning()* имеет следующий формат:

QMessageBox::warning(родительский виджет, заголовок, сообщение, кнопки);

Наряду с *warning() QMessageBox* содержит функции *information()*, *question()*, и *critical()*, каждая из которых имеет собственную пиктограмму.

Далее определяются слоты *open()*, *save()* и *saveAs()*.

```
void MainWindow::open()
{
    if (okToContinue())
    {
        QString fileName = QFileDialog::getOpenFileName(this,
                                                         tr("Open Table"),".",
                                                         tr("Table files
                                                         (*.tbl)"));
        if(!fileName.isEmpty()) loadFile(fileName);
    }
}

bool MainWindow::save()
{
    if (curFile.isEmpty())
        return saveAs();
    else return saveFile(curFile);
}

bool MainWindow::saveAs()
{
    QString fileName = QFileDialog::getSaveFileName(this,
                                                    tr("Save Table"), ".",
                                                    tr("Table files (*.tbl)"));
    if (fileName.isEmpty()) return false;
    return saveFile(fileName);
}
```

Для их корректной работы необходимо реализовать дополнительные функции для работы с файлами *loadFile()*, *readFile()* для загрузки и *saveFile()*,

writeFile() для сохранения файлов соответственно.

```
bool MainWindow::loadFile(const QString &fileName)
{
    if(!this->readFile(fileName))
    {
        statusBar()->showMessage(tr("Loading canceled"),2000);
        return false;
    }
    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File loaded"),2000);
    return true;
}

bool MainWindow::readFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QIODevice::ReadOnly))
    {
        QMessageBox::warning(this, tr("Table"),
                             tr("Cannot read file %1:\n%2.")
                             .arg(file.fileName())
                             .arg(file.errorString()));
        return false;
    }
    QDataStream in(&file);
    in.setVersion(QDataStream::Qt_5_2);

    quint16 row;
    quint16 column;
    QString str;

    QApplication::setOverrideCursor(Qt::WaitCursor);
    while (!in.atEnd())
    {
        in >> row >> column >> str;
        if (!str.isNull())
        {
            QTableWidgetItem *item = new QTableWidgetItem;
            item->setText(str);
            TableWidget->setItem(row, column, item);
        }
    }
    QApplication::restoreOverrideCursor();
    return true;
}


bool MainWindow::saveFile(const QString &fileName)
{
    if(!this->writeFile(fileName))
    {
        statusBar()->showMessage(tr("Saving canceled"), 2000);
        return false;
    }
    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File saved"), 2000);
    return true;
}
```

```
bool MainWindow::writeFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QIODevice::WriteOnly))
    {
        QMessageBox::warning(this, tr("Table"),
                             tr("cannot write file %1:\n%2.")
                             .arg(file.fileName())
                             .arg(file.errorString()));
        return false;
    }

    QDataStream out(&file);
    out.setVersion(QDataStream::Qt_5_2);

    QApplication::setOverrideCursor(Qt::WaitCursor);
    QTableWidgetItem *item;
    QString str;
    for (int row=0; row<TableWidget->rowCount(); ++row)
    {
        for (int column=0; column<TableWidget->columnCount(); ++column)
        {
            item=TableWidget->item(row,column);
            if (NULL != item)
            {
                str = item->text();
                if (!str.isNull())
                    out<<quint16(row)<<quint16(column)<<str;
            }
        }
    }
    QApplication::restoreOverrideCursor();
    return true;
}
```

В приведенном выше фрагменте кода приведен реализация функций, работающих с файлами. Они используют класс *QFile* для создания файла, а также объекты класса *QDataStream* для создания потоков ввода/вывода данных. Формат сохраняемого файла следующий: «номер строки» «номер столбца» «содержимое ячейки». Номера строк и столбцов в файле представляются в виде переменных типа *quint16*, а для сохранения содержимого ячейки используется *QString*. Каждая ячейка электронной таблицы представляется в виде объекта класса *QTableWidgetItem* и для преобразования содержимого ячейки в *QString* используется встроенный метод *text()*.

Завершает создание графического пользовательского интерфейса программы реализация необходимых слотов и функций для работы со списком ранее открытых файлов.

```
void MainWindow::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    setWindowModified(false);

    QString shownName = "Untitled";
    if (!curFile.isEmpty())
    {
        shownName = strippedName(curFile);
```

```
        recentFiles.removeAll(curFile);
        recentFiles.prepend(curFile);
        updateRecentFileActions();
    }

    setWindowTitle(tr("%1[*]-%2").arg(shownName)
                   .arg(tr("Table")));
}

QString MainWindow::strippedName(const QString &fullFileName)
{
    return QFileInfo(fullFileName).fileName();
}

void MainWindow::updateRecentFileActions()
{
    QMutableStringListIterator i(recentFiles);
    while (i.hasNext())
        if (!QFile::exists(i.next()))
            i.remove();
    for (int j=0; j<MaxRecentFiles; ++j)
    {
        if (j < recentFiles.count())
        {
            QString text = tr("&%1 %2").arg(j+1)
                    .arg(strippedName(recentFiles[j]));
            recentFileActions[j]->setText(text);
            recentFileActions[j]->setData(recentFiles[j]);
            recentFileActions[j]->setVisible(true);
        }
        else recentFileActions[j]->setVisible(false);
    }
    separatorAction->setVisible(!recentFiles.isEmpty());
}

void MainWindow::openRecentFile()
{
    if (okToContinue())
    {
        QAction *action = qobject_cast<QAction *>(sender());
        if (action)
            loadFile(action->data().toString());
    }
}
```

Дополнительно следует привести реализацию защищенной функции *closeEvent(QCloseEvent *event).*

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    if(okToContinue()) event->accept();
    else event->ignore();
}
```

На этом создание главного окна программы заканчивается.

7. Методические указания по выполнению работы

Лабораторную работу необходимо выполнить в следующей последовательности:

1. Изучите теоретические разделы 2-6, выполните на компьютере все приведенные в данных разделах примеры, скомпилируйте и запустите на исполнение полученную программу. В качестве дополнительных источников информации, воспользуйтесь указанными в перечне используемой литературы (раздел 8) пособиями, а также открытыми источниками в сети Интернет.
2. Самостоятельно выполните задание, описанное ниже в разделе 7.1.
3. Подготовьте исходный код программы в Qt Creator и файл формы в Qt Designer и покажите их преподавателю.
4. В случае отсутствия видимых ошибок в коде, скомпилируйте программу и запустите ее на выполнение.
5. Получите оценку.

7.1. Расширение функционала главного окна

Дополните полученную на предыдущих этапах работы программу двумя действиями: «Удалить строку» и «Удалить столбец». Поместите разработанные действия в контекстное меню программы (по желанию – в одно из главных меню и панель инструментов). Каждое из действий должно удалять строку (или, соответственно, столбец), к которой относится текущая выделенная ячейка.

Для удаления строки следует использовать встроенный метод *QTableWidget->removeRow*. Для удаления столбца следует использовать встроенный метод *QTableWidget->removeColumn*.

За пример для разрабатываемых действий следует взять *newAction*.

8. Рекомендуемая литература

1. Qt. Профессиональное программирование на C++: Наиболее полное руководство / М. Шлее. - СПб. : БХВ-Петербург, 2005. – 544 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.

Учебное издание

Семкин Артем Олегович

Заичко Кирилл Владимирович

Шарангович Сергей Николаевич

ИНФОРМАТИКА

Руководство к лабораторной работе «Библиотека Qt. Создание главных окон программы»

Формат 60x84 1/16. Усл. печ. л. _____.
Тираж _____ экз. Заказ_____.
Томский государственный университет систем управления и радиоэлектроники.
634050, Томск, пр. Ленина, 40. Тел. (3822) 533018.