НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
МОСКОВСКИЙ ЭНЕРГЕТИЧЕСКИЙ ИНТСТИТУТ

КАФЕДРА ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Работа по курсу «Графические системы»
студента гр. А-6-08 Гусева Михаила

«Разработка GUI с помощью кросс-платформенного инструментария Qt»

Москва 2012 г.

Оглавление

Введение

Данная работа является учебным материалом для тех, кто желает ознакомиться и изучить инструментарий Qt, а главное научиться при помощи него разрабатывать современные и технологичные пользовательские интерфейсы.

В первой части работы будут рассмотрены базовые понятия и концепции фреймворка Qt. Все они тесно связаны и широко используются при разработке GUI. В части 2 мы познакомимся с наиболее важными и уникальными особенностями Qt, уже непосредственно в контексте пользовательских интерфейсов. А в части 3 разберём создание реального приложения, с использованием технологий и приёмов, рассмотренных в первых двух частях.

1. Обзор фреймворка Qt

Qt – это кросс-платформенная библиотека, разработанная финской компанией Trolltech, ныне принадлежащей корпорации Nokia. Qt реализована на языке программирования C++ и пользуется популярностью, прежде всего, у C++-программистов, поскольку использует "родной" для них интерфейс. Возможности C++ в Qt так же значительно расширены с помощью макросов и MOC (Meta Object Compiler): добавлены сигналы/слоты, возможность создавать собственные свойства классов (propperty). Но Qt не ограничена только лишь языком C++. Для программистов Python, Ruby, Php, Perl и Java (проект Jambi) также реализованы интерфейсы взаимодействия, которые, как правило, используются для построения графического интерфейса пользователя. К сильным сторонам Qt можно отнести:

– Кроссплатформенность: Qt работает как в настольных операционных системах Windows, Linux, Mac OS и др., так и мобильных Symbian, Maemo, MeeGo.

– Быстроту: часто кросплатформенные приложения, написанные на платформах Java и .Net Framework, оказываются медлительным из-за дополнительного уровня абстракции. Программы Qt являются компилируемыми C++-приложениями, т. е. они работают также быстро как и приложения C++.

Краткая история Qt

Средства разработки Qt впервые стали известны общественности в мае 1995 года. Первоначально Qt разрабатывалось Хаарвардом Нордом (исполнительный директор) и Айриком Чеймб-Ингом (президент), которые познакомились в Норвежском институте технологии г. Тронхейм. В процессе работы над другим проектом по созданию GUI для систем Unix, Macintosh и Windows у них появилась идея о создании меж платформенной системы разработки графического пользовательского интерфейса. Так, в начале 1994 года два молодых программиста собирались выйти на установившийся рынок, не имея ни заказчиков, ни законченного продукта, ни денег. К счастью, жены обоих имели работу и могли поддержать своих мужей в течение двух лет, которых, как считали Айрик и Хаарвард, будет достаточно для разработки программного продукта, позволяющего начать зарабатывать деньги.

Необходимо было придумать название. Буква «Q» была выбрана в качестве префикса классов, поскольку эта буква имела красивое начертание в шрифте Emacs, которым пользовался Хаарвард. Была добавлена буква «t», означающая «toolkit» (инструментарий). Компания была зарегистрирована 4 марта 1994 года и по началу называлась «Quasar Technologies», затем «Troll Tech», затем «Trolltech», а теперь просто «Qt Software» после покупки компании фирмой Nokia. В апреле 1995 года норвежская компания «Metis» заключила с разработчиками контракт на разработку программного обеспечения на основе Qt, и уже спустя месяц о выпуске первой версии было объявлено на comp.os.linux.announce. Qt можно было использовать в разработках как Windows, так и Unix, причём программный интерфейс был одинаковый на обеих платформах. С первого дня предусматривались две лицензии применения Qt: коммерческая лицензия предназначалась для коммерческих разработок, и свободно распространяемая версия предназначалась для разработок Open-source проектов. Контракт с «Metis» сохранил компанию на плаву, хотя в течение долгих 10-ти месяцев не было продано ни одной коммерческой лицензии Qt. В марте 1996 года Европейское управление космических исследований стало вторым заказчиком Qt, которое приобрело десять коммерческих лицензий, а к концу этого года после выхода версии Qt 1.1 заказчиками было приобретено в общей сложности 28 лицензий. В этом году был также основан Маттиасом Эттричем проект KDE, что помогло Qt стать фактическим стандартом по разработке на C++ графического пользовательского интерфейса в системе Linux. После выхода версии 2 в 1999 году Qt выиграла премию журнала «Linux World» за лучшую библиотеку или

инструментальное средство. Qt 3.0 была выпущена в 2001 году, работая теперь в системах Windows, Mac Os X, Unix. Она содержала 42 новых класса, и объем её программного кода превышал 500 000 строк.

Летом 2005 года была выпущена Qt 4.0. Имея около 500 классов и более 9000 функций, Qt 4 оказалась больше и богаче любой предыдущей версии; она была разбита на несколько библиотек, чтобы разработчики могли использовать только нужные им части Qt. Версия Qt 4 представляла собой большой шаг вперёд по сравнению с предыдущими версиями; она содержала полностью новый набор эффективных и простых в применении классов-контейнеров, усовершенствованную функциональность архитекторы модель/представление, быстрый и гибкий фреймфорк графики 2D и мощные классы для просмотра и редактирования текста в кодировке Unicode, не говоря уже о тысячах небольших улучшений по всему спектру классов Qt.

На сегодняшний день (декабрь 2012), рабочей версией библиотек Qt является версия 4.8.3, выпущенная 13 сентября 2012 года, а концу года планируется выпуск Qt 5.0.

Особенности библиотеки Qt

Иерархия классов Qt имеет четкую внутреннюю структуру, которую важно понять, чтобы уметь хорошо и интуитивно ориентироваться в этой библиотеке. Библиотека Qt – это множество классов (более 500), которые охватывают большую часть функциональных возможностей операционных систем, предоставляя разработчику мощные механизмы, расширяющие и, вместе с тем, упрощающие разработку приложений. При этом не нарушается идеология операционной системы. Qt не является единым целым, она разбита на модули (табл. 1.1) . Любая Qt-программа так или иначе должна использовать хотя бы один из модулей, в большинстве случаев это QtCore и QtGui, поэтому эти два модуля определены в программе создания make-файлов по умолчанию.

Таблица 1.1. Полный перечень модулей библиотеки Qt 4.8.3.

QtCore	Основополагающий модуль, состоящий из классов, не связанных с графическим интерфейсом. Ядро библиотеки
QtGui	Компоненты графического интерфейса
QtMultimedia	Классы для низкоуровневой работы с мультимедиа
QtNetwork	Набор классов для сетевого программирования
QtOpenGL	Классы для работы с OpenGL

QtOpenVG	Классы для работы с OpenVG
QtScript	Модуль поддержки языка сценариев
QtScriptTools	Модуль дополнительных возможностей поддержки языка сценария
QtSql	Классы для работы с базами данных
QtSvg	Модуль для работы с SVG (Scalable Vector Graphics, масштабируемая векторная графика)
QtWebKit	Модуль для создания Web-приложений
QtXml	Классы для работы с XML
QtXmlPatterns	Поддержка XQuery и XPath для работы с и другими моделями данных
QtDeclarative	Модуль, предоставляющий декларативный фреймворк для создания динамичных, настраиваемых пользовательских интерфейсов
Phonon	Модуль для поддержки воспроизведения и записи видео и аудио, как локально, так и с устройств и по сети
Qt3Support	Модуль с классами, необходимыми для совместимости с библиотекой Qt версии 3.x.x
QtDesigner	Классы создания расширений QtDesigner'а для своих собственных виджетов
QtUiTools	Классы для обработки в приложении форм Qt Designer
QtHelp	Справочная система
QtTest	Модуль для работы с UNIT тестами (тестирование отдельных модулей)
QAxContainer	Работа с управляющими элементами ActiveX (Windows)
QAxServer	Расширение для разработки ActiveX-серверов (Windows)
QtDBus	Классы для межпроцессорного взаимодействия системы D-Bus (Unix)

Также библиотека Qt имеет ряд важных заголовочных файлов с объявлением основных типов, функций, констант и пространств имен:

Таблица 1.2. Список заголовочных файлов Qt 4.8.3.

Функции преобразования порядка байтов	Заголовочный файл <QtEndian> предоставляет функции для преобразования между прямым (little endian) и обратным порядком (big endian) представления чисел.

Базовые алгоритмы	Заголовочный файл <QtAlgorithms> включает в себя обобщенные алгоритмы, основанные на шаблонах.
Глобальные объявления Qt	Заголовочный файл <QtGlobal> включает в себя основные глобальные объявления. Он включает в себя большинство других заголовочных файлов Qt. Объявлены основные типы, функции и макросы.
Платформо-зависимые функции	Экспортируемые функции для тонкой настройки приложений Qt.

Рассмотрим особенности различных модулей для программирования приложений с Qt, а так же важные моменты и отличия от других библиотек.

QtCore

[Анг] http://doc.qt.digia.com/qt/qtcore.html
[Рус] http://doc.crossplatform.ru/qt/4.7.x/qtcore.html

Итак, базовым модулем Qt является модуль QtCore. Этот модуль является базовым для приложений и не содержит классов, относящихся к интерфейсу пользователя. Например, если требуется реализовать только консольное приложение, то, вполне возможно ограничиться одним этим модулем. В модуль QtCore входят более 200 классов, вот некоторые из них:

- Класс **QObject** - базовый класс для всех объектов Qt. Его главная особенность – наличие системы сигналов и слотов;
- Контейнерные классы QList, QVector, QMap;
- Классы строка и список строк: QString, QStringList;
- Классы для задания геометрических объектов в вещественных координатах QPointF, QLineF;
- Классы для ввода и вывода QIODevice, QTextStream, QFile;
- Классы процесса QProcess и для программирования многопоточности QThread, QWaitCondition, QMutex;
- Классы для работы с таймером QBasicTimer и QTimer;
- Классы для работы с датой и временем QDate и QTimer;
- Базовый класс событий QEvent;

• Класс для сохранения настроек приложения QSettings;
• Класс приложения QCoreApplication, из объекта которого, если требуется, можно запустить цикл событий.

Механизм сигналов и слотов

[Анг] http://doc.qt.digia.com/qt/signalsandslots.html
[Рус] http://doc.crossplatform.ru/qt/4.7.x/signalsandslots.html

Виджеты Qt имеют возможность посылать приложению сигналы, извещая его о том, что пользователь произвел какое-либо действие или о том, что виджет изменил свое состояние. Например, экземпляры класса QPushButton (обычная кнопка) посылают приложению сигнал clicked(), когда пользователь нажимает на кнопку. Сигнал может быть "подключен" к функции-обработчику, именуемой слотом. Таким образом, когда виджет посылает сигнал, автоматически происходит вызов слота. Механизм сигналов и слотов является очень гибким и мощным инструментом. Во-первых, он типобезопасен, в отличие от механизма функций обратного вызова callbacks (обратный вызов позволяет в функции исполнять код, который задаётся в аргументах при её вызове), поскольку некогда нельзя проверить, что функция обработки вызывает отзыв с правильными аргументами. Во-вторых, в отличие от стандартной жёсткой связи функции вызова и функции обработки, сигналы и слоты связаны не жёстко: класс, испускающий сигналы, не знает и не интересуется, который из слотов получит сигнал. Механизм сигналов и слотов Qt гарантирует, что, если сигнал со слотом соединён, слот будет вызываться с параметрами сигнала в нужный момент. Это является истинной инкапсуляцией информации и четко вписываются в концепцию ООП. Из-за удобства использования, сигналы и слоты находят широкое применение на практике.

Классы-контейнеры

[Анг] http://doc.qt.digia.com/qt/containers.html
[Рус] http://doc.crossplatform.ru/qt/4.3.2/containers.html

Библиотека Qt предоставляет набор основанных на шаблонах классов-контейнеров. Эти классы могут использоваться для хранения элементов указанного типа. Контейнеры оптимизированы для быстрого исполнения, низкого потребления памяти и минимального раздувания кода. Они являются альтернативой контейнерам STL (стандартной библиотека шаблонов C++). Помимо классов-контейнеров в библиотеке Qt имеются также шаблонные

классы, похожие поведением на классы-контейнеры. Всех их отличает общая простота использования и функциональность.

Так как тип T шаблона позволяет хранить информацию любого присваиваемого типа (то есть тех типов, которые имеют описанный конструктор копирования и оператор присваивания, или где компилятор их может задать по умолчанию), их целесообразно использовать для создания различных **массивов данных**. Учитывая то, что контейнеры могут быть вложенными, это делает такой механизм весьма удобным и гибким в использовании. Описание многомерных массивов без классов контейнеров потребовало бы либо сложной реализации на указателях C++, либо создания вместо одной нескольких переменных.

QtGUI

[Анг] http://doc.qt.digia.com/qt/qtgui.html
[Рус] http://doc.crossplatform.ru/qt/4.7.x/qtgui.html

В терминологии Qt, так же как и в некоторых других системах по построению GUI, все визуальные компоненты, из которых строится графический интерфейс, называются **виджетами** (widgets). Например виджеты – это кнопки, меню, полосы прокрутки, и так далее. В части 2 и части 3 данной работы будет подробно рассматриваться аспект разработки и создания графического пользовательского интерфейса с использованием Qt.

Также, в модуле QtGUI описывается один из важнейших классов, использующихся при создании приложений – QApplication. Класс QApplication управляет главным потоком и основными настройками приложения с GUI. Он содержит главный цикл обработки сообщений, где обрабатываются и пересылаются все сообщения посланные оконной системой и другими ресурсами. В нём реализованы инициализация, завершение приложения и управление сессией. Для любого приложения с GUI использующего Qt есть ровно один объект QApplication, независимо от того, имеет ли приложение 0, 1, 2 или больше окон в любое времмя. Для преложений Qt не имеющих GUI вместо него используется не связанный с библиотекой QtGui QCoreApplication.

QtOpenGL, QtOpenVG, QtSvg

[Анг] http://doc.qt.digia.com/qt/qtopengl.html
[Рус] http://doc.crossplatform.ru/qt/4.3.2/qtopengl.html

Данные модули Qt обеспечивают возможность работы с различными Графическими Системами, что может быть полезно, не только для создания различных приложений с графикой, но и при разработке особенных GUI.

OpenGL является стандартным, платформонезависимым программным интерфейсом, предназначенным для воспроизведения двухмерной и трёхмерной графики. Приложения Qt могут отображать графику 3D, используя модуль QtOpenGL, который рассчитан на применение системной библиотеки OpenGL. Этот модуль предоставляет класс QGLWidget, для которого мы можем создавать подклассы для разработки собственных виджетов, рисующихся с использованием команд OpenGL. Для многих 3D приложений это весьма существенно. Версия Qt 4 предоставляет также возможность использовать класс **QPainter** (отвечает за 2D-рисование на любых плоскостях) модуля QtGUI применительно к QGLWidget, как если бы это был простой виджет QWidget. Большое преимущество здесь состоит в том, что в итоге получается высокая производительность OpenGL для большинства операций рисования, таких как трансформации и пиксельные рисунки. Чтобы отображать выходящую за пределы экрана поверхность с использованием аппаратного ускорения, могут использоваться расширения pbuffer и framebuffer object, доступ к которым осуществляется через классы QGLPixelBuffer и QGLFramebufferObject.

Вывод графики при помощи OpenGL в приложении Qt выполняется достаточно просто: нужно создать подкласс QGLWidget, переопределить несколько виртуальных функций и собрать приложение вместе с библиотеками QtOpenGL и OpenGL.

OpenVG — это стандартный API, разработанный Khronos Group. OpenVG предназначен для аппаратно-ускоряемой двухмерной векторной графики. Он предназначается в первую очередь для мобильных телефонов и смартфонов, медиа и игровых консолей, таких как PlayStation 3, и для других электронных устройств. OpenVG поможет производителям ПО создавать более быстрые пользовательские интерфейсы, которые будут гораздо менее зависимыми от центрального процессора, что позволит экономить не только процессорное время, но и электроэнергию. OpenVG хорошо подходит как для ускорения флэш-анимации, так и SVG-графики.

QtMultimedia, Phonon

[Анг] http://doc.qt.digia.com/qt/qtmultimedia.html
[Анг] http://doc.qt.digia.com/qt/phonon-overview.html
[Рус] http://doc.crossplatform.ru/qt/4.7.x/qtmultimedia.html
[Рус] http://doc.crossplatform.ru/qt/4.5.0/phonon-module.html

Модули QtMultimedia и Phonon используются в Qt для работы с мультимедийным контентом. Так, Phonon – это кроссплатформенный мультимедийный каркас, который даёт возможность использовать аудио- и видеоконтент в приложениях Qt. Пространство имен Phonon содержит перечень всех предоставляемых модулем классов, функций и пространств имен. Например в модуле объявлены такие классы, как: Phonon::VideoPlayer (Используется для выполнения воспроизведения видео), Phonon::VideoWidget (Виджет, используемый для отображения видео), Phonon::MediaSource (Мультимедийные данные для медиа-объектов), Phonon::Effect (Используется для преобразования аудиопотоков), и т.д. Модуль QtMultimedia содержит низкоуровневые методы в своих классах по доступу к мультимедийному контенту, однако в документации Qt упоминается, что модуль Phonon в большинстве случаев использовать удобнее, т.к. он содержит методы более высокого уровня.

QtSql

[Анг] http://doc.qt.digia.com/qt/qtsql.html
[Анг] http://doc.qt.digia.com/qt/qt-sql.html *//общее описание работы с БД в Qt*
[Рус] http://doc.crossplatform.ru/qt/4.3.2/qtsql.html

Модуль QtSql обеспечивает независимый от платформы и типа базы данных интерфейс для доступа с помощью языка SQL к базам данных. Этот интерфейс поддерживается набором классов, использующих архитектуру Qt модель/представление для интеграции средств доступа к базам данных с интерфейсом пользователя. Связь с базой данных обеспечивается объектом класса QSqlDatabase. Qt использует драйверы для связи с программным интерфейсом различных баз данных. Версия Qt 4.8.3 включает в себя следующие драйверы (табл. 1.4):

Таблица 1.4. Драйверы СУБД Qt 4.8.3.

Имя драйверов	DBMS
QDB2	IBM DB2 (версия 7.1 и выше)
QIBASE	Borland InterBase
QMYSQL	MySQL
QOCI	Драйвер Oracle Call Interface
QODBC	Open Database Connectivity (ODBC) - Microsoft SQL Server и другие ODBC-совместимые базы данных
QPSQL	PostgreSQL (версия 7.3 и выше)
QSQLITE2	SQLite версии 2
QSQLITE	SQLite версии 3
QTDS	Драйвер Sybase Adaptive Server
QSYMSQL	Драйвер SQLite версии 3 для Symbian SQL Database

Отдельно, стоит отметить, что Qt поставляется вместе с SQLite - общедоступной, не нуждающейся в сервере базой данных (остальные драйвера в зависимости от версии Qt, OpenSource или коммерческой, могут включаться или не включаться). Движок этой БД не является отдельно работающим процессом, с которым взаимодействует программа. SQLite представляет собой библиотеку, с которой компонуется программа и, таким образом, движок становится составной частью программы. Такой подход для хранения данных приложения очень удобен, например в контексте разработки GUI, так как можно очень удобно хранить различные настройки пользовательского интерфейса.

Разработка в среде Qt

Фреймворк Qt всегда славился своим заботливым подходом к программистам. Например, наличие качественной документации в отличие, например, от wxWidgets, и очень удобной IDE, является весомым преимуществом при выборе средств для разработки приложений программистами. Так же, хотелось бы отметить удобные названия классов и их методов в контексте семантики Qt. Например: все методы по заданию значений полям классов, начинаются со слова set (QWidget::setGeometry(), QWidget::setEnabled()); все методы, возвращающие логическое значение

true/false и используемые для установки факта чего-либо начинаются со слова is (QObject::isWidgetType(),QAbstractButton::isChecked ()).

Наличие удобной IDE – QtCreator

[Анг] http://qt.digia.com/Product/Developer-Tools/
[Рус] http://ru.wikipedia.org/wiki/Qt_Creator

Qt Creator предоставляет кроссплатформенную, полностью интегрированную среду разработки (IDE) для создания приложений для множества настольных и мобильных платформ. Он доступен для различных версий операционных систем Linux, Mac OS X и Windows. Одним из главнейших достижений Qt Creator является то, что он позволяет команде разработчиков работать над проектом на различных платформах с использованием общих инструментов для разработки и отладки, т.е. работать над проектами. Чтобы быть в состоянии собирать и запускать приложения, Qt Creator нуждается в той же информации, которая потребуется компилятору. Эта информация указана в настройках сборки и запуска проекта, а именно в файле с расширением *.pro. Создание проекта позволяет: группировать файлы вместе, добавить собственные шаги сборки, включить формы и файлы ресурсов, указать настройки для запускаемых приложений.

Qt Creator поставляется с редактором кода Qt Designer, который раньше являлся отдельным приложением, для проектирования и сборки графических интерфейсов пользователя (GUI) из виджетов Qt. Можно использовать Qt Designer, чтобы располагать и настраивать виджеты или диалоги, и тестировать их, используя разные стили и разрешения экрана. Созданные с помощью Qt Creator виджеты и формы легко интегрируются в программный код с использованием механизма сигналов и слотов Qt, которые позволяют легко определить поведение графических элементов. Все свойства, установленные в Qt Designer, могут быть динамически изменены в коде. Более того, такие особенности как продвижение виджетов и собственные модули позволят разработчикам использовать собственные виджеты с Qt Designer.

Текстовый редактор QtCreator распознаёт языки C++, QML, а так же ещё множество других, как код, а не как простой текст. Это позволяет ему: дать вам возможность программисту писать хорошо форматированный код; угадывать что разработчик хочет написать и дополнять код; отображать сообщения об ошибках и предупреждения; дать возможность разработчику перемещаться между классами, функциями и символами; предоставлять контекстно-зависимую справку по классам, функциям и символам; и т.д.

Qt Creator интегрирован с набором полезных инструментов, такие как системы управления версиями (Git, Subversion, Perforce, CVS, Mercurial) и эмулятор Qt (для проверки приложений на мобильных устройствах).

Также, в Qt Creator имеется отладчик. Qt Creator предоставляет интерфейсы к GNU Symbolic Debugger (gdb) и Microsoft Console Debugger (CDB) для отладки обычных приложений на C++ и внутренние отладчики для Java Script.

Наличие удобной справочной документации

Справочная документация по средствам разработки Qt является важным инструментом в руках любого разработчика Qt-программ, поскольку в ней есть все необходимые сведения по любому классу и любой функции Qt. Эта документация имеется в формате HTML (каталог doc/html в системе Qt) и ее можно просматривать любым веб браузером, а также внутри QtCreatora в специальной закладке или через контекстный доступ в текстовом редакторе. QtCreator быстрее находит нужную информацию и им легче пользоваться.

В данный момент полноценная и официальная документация существует только на английском языке. Однако в сети интернет существуют различные версии перевода документации на русский (см. Литература и Интернет-источники).

Использование Qt

Лицензирование Qt

Qt имеет тройное лицензирование:

- Qt Commercial - коммерческая лицензия, после приобретения которой вы можете выпускать программный продукт под собственной лицензией.
- GNU GPL - если вы open-source разработчик это ваш выбор.
- GNU LGPL - позволяет вести разработку под собственной лицензией, однако вы не можете вносить изменения в код Qt.

Сборка библиотек

Для того, чтобы использовать фреймворк Qt, необходимо скачать с официального портала проекта (см. Литература и Интернет-источники) либо готовый SDK, либо open-source исходники определённой версии библиотек. Qt SDK включает в себя всё, что нужно для разработки Qt на многих платформах: уже собранные библиотеки, IDE Qt Creator, API для разработки под мобильные платформы, QtQuick. Всё что нужно, чтобы начать разработку с Qt SDK – это

скачать установочный файл и запустить. Однако Qt SDK является коммерческой версией продукта и предоставляется только на 30-дневный срок пользования, требуя при этом небольшой процедуры регистрации. Для open-source разработки необходимо отдельно скачать и собрать библиотеки из исходников. Эта процедура, а также список требований к установке описаны здесь: [Рус] http://doc.crossplatform.ru/qt/4.5.0/installation.html

Программы в Qt можно собирать как динамически, так и статически. Это означает, что во втором случае, единожды собрав библиотеки Qt и написав приложение, можно скомпилировать его так, чтобы в дальнейшем была возможность запускать его на определённой ОС без наличия специальной виртуальной машины, или различных библиотек.
(http://www.cyberforum.ru/qt/thread234568.html)

2. Разработка GUI в Qt

Используемый средствами разработки Qt подход к построению графического пользовательского интерфейса интуитивно понятен и очень гибок. Среди работающих в Qt программистов наиболее распространен подход, когда сначала создаются все необходимые графические элементы и затем соответствующим образом настраиваются их свойства. Программисты добавляют виджеты к компоновщикам графических элементов, которые автоматически устанавливают для них нужный размер и положение. Управление работой графического интерфейса осуществляется через взаимодействие виджетов друг с другом посредством применения механизма сигналов и слотов Qt или механизма событий.

Виджеты

Виджеты Qt

По терминологии Qt и Unix виджетом (widget) называется любой визуальный элемент графического интерфейса пользователя. Этот термин происходит от «window gadget» и соответствует элементу управления («control») и контейнеру («container») по терминологии Windows. Кнопки, меню, полосы прокрутки и фреймы являются примерами виджетов. Одни виджеты могут содержать в себе другие виджеты. Например, окно приложения обычно является виджетом,

содержащим QMenuBar (панель меню), несколько QToolBar (панель инструментов), QStatusBar (строка состояния) и некоторые другие виджеты. Большинство приложений используют QMainWindow или QDialog в качестве окна приложения, однако Qt настолько гибок, что любой виджет может быть окном, даже текстовая метка QLabel может являться окном приложения.

Виджет получает сообщения мыши (обработчики - mouseMoveEvent(), wheelEvent() и т.д.), клавиатуры (обработчики - keyPressEvent() и т.д.) и другие сообщения оконной системы, а также рисует свое представление на экране. Все виджеты прямоугольны и расположены в порядке, называемом Z-индексом. Изображение виджета может обрезаться родителем и виджетом, стоящим перед ним.

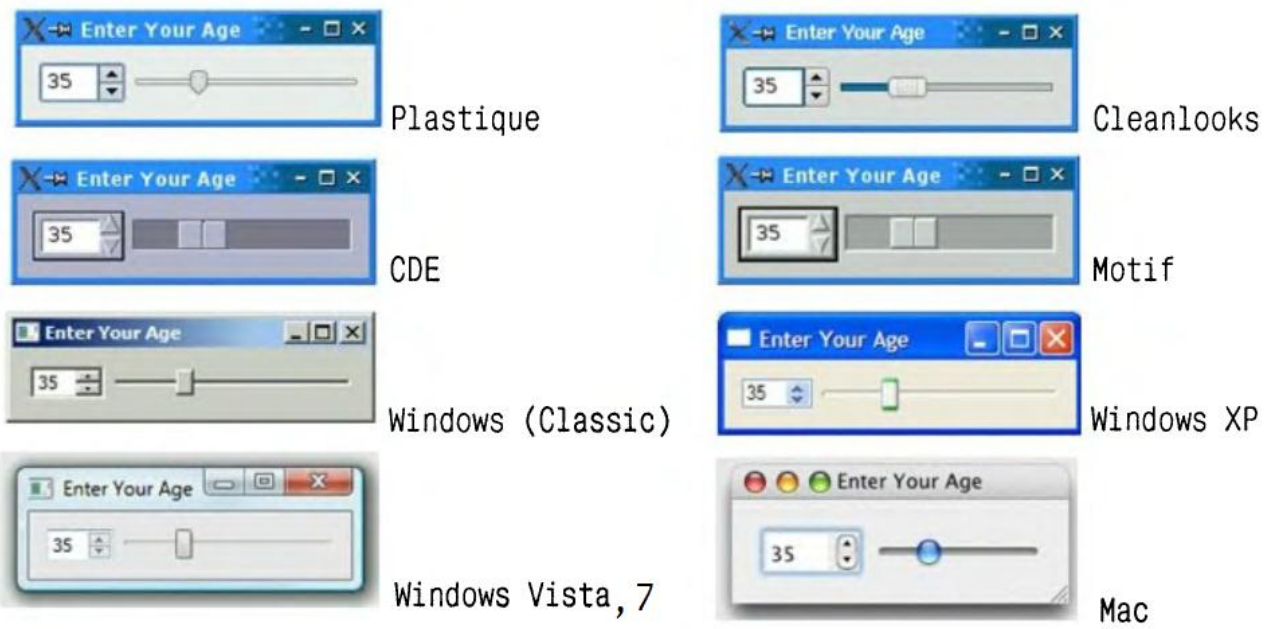
Класс QWidget

[Анг] http://doc.qt.digia.com/4.7-snapshot/qwidget.html
[Рус] http://qtdocs.narod.ru/4.1.0/doc/html/qwidget.html

Все классы элементов графического интерфейса так или иначе унаследованы от главного класса-прародителя QWidget. Он имеет множество функций-членов, некоторые из которых имеют весьма ограниченную функциональность: например, QWidget имеет свойство управляющее шрифтом, но никогда не использует его непосредственно. Им могут пользоваться все дочерние виджеты этого виджета для реального изменения своих шрифтов. Есть множество подклассов класса QWidget, которые обеспечивают реальную функциональность его функций, а также добавляют свои новые возможности, в зависимости от их назначения, такие как, например, QPushButton (кнопка), QLabel (метка). Полный перечень методов (по группам), общих для всех элементов графического интерфейса представлен в справочной документации (ссылки в начале абзаца). Список всех доступных стандартных виджетов имеется в Qt Creator.

Стили виджетов

Приложения Qt естественно выглядят на любой поддерживаемой платформе. В Qt это достигается путем эмуляции изобразительных средств данной платформы, а не путем создания «оболочки» для конкретной платформы или путем набора виджетов из инструментария. На рисунке представлены стили окон для разных ОС на примере приложения «Введите свой возраст»:



Стиль Plastique - это заданный по умолчанию стиль для приложений Qt/X11, работающих под KDE, а Cleanlooks - это стиль по умолчанию для GNOME. В этих стилях используются градиенты и сглаживание, что обеспечивает стилю современный вид. Пользователи приложения Qt могут изменить заданный по умолчанию стиль при помощи опции командной строки -style. Например, чтобы запустить приложение «Введите свой возраст» в X11 с использованием стиля Motif, необходимо просто ввести следующую команду:

./age -style motif

В отличие от других стилей, стили WindowsXP, Windows Vista, Windows 7 и Mac доступны только на своих «родных» платформах, поскольку они используют систему тем платформы. Существует также дополнительный стиль, QtDotNet, от Qt Solutions. Кроме того, существует возможность создавать собственные стили. Есть три основных подхода к переопределению внешнего вида встроенных виджетов Qt:

• Можно создать подклассы отдельных классов виджетов и переопределить их обработчики событий рисования и функций мыши. Например, различные цвета виджетов, можно изменять при помощи механизма палитр (класс QPalette). Класс QPalette содержит цветовую группу для каждого состояния виджета. Палитра состоит из трех цветовых групп: Активный, Недоступный и Неактивный. Все виджеты в Qt содержат палитру и используют ее для

отрисовки себя. Это делает пользовательский интерфейс легко настраиваевым и простым в программировании. Описание класса QPalette:
[Рус] http://qtdocs.narod.ru/4.1.0/doc/html/qpalette.html

• Можно создать подкласс класса QStyle или готового стиля, например, QWindowsStyle. Этот подход весьма мощный. Он применяется в самом Qt для того, чтобы он сохранял привычный облик на разных платформах, которые он поддерживает.

• Начиная с версии Qt 4.2, можно использовать таблицы стилей Qt, концепция которых возникла под влиянием CSS (Cascading style sheets, каскадные таблицы стилей) в HTML. Поскольку таблицы стилей представляют собой простые текстовые файлы, интерпретируемые во время выполнения, для их использования не требуется знание программирования. Использование таблиц стилей Qt:
[Рус] http://doc.crossplatform.ru/qt/4.5.0/stylesheet-reference.html;
[Анг] http://doc.qt.digia.com/4.7-snapshot/stylesheet.html

Компоновка

[Анг] http://doc.qt.digia.com/4.7-snapshot/layout.html
[Рус] http://qtdocs.narod.ru/4.1.0/doc/html/layout.html

Каждому размещаемому на форме виджету необходимо задать соответствующий размер и позицию. Существует три основных способа управления компоновкой дочерних виджетов формы в Qt: абсолютное позиционирование, ручная компоновка и применение менеджеров компоновки. Рассмотрим каждый из способов и выделим их достоинства и недостатки.

Абсолютное позиционирование является самым негибким способом компоновки виджетов. Он предусматривает жесткое кодирование в программе размеров и позиций дочерних виджетов формы и фиксированный размер самой формы. Например, код по заданию размеров и позиции некоторых элементов интерфейса может выглядеть так (используются методы класса QWidget, имеющиеся у всех элементов интерфейса):

```
…
Label->setGeometry(9, 9, 50, 25); //задаём геометрию элемента
интерфейса. Первые два параметра – положение, вторые два - размеры
LineEdit->setGeometry(65. 9, 200, 25);
```

```
CheckBox->setGeometry(9, 71, 256, 23);
tableWidget->setGeometry(9, 100, 256, 100);
closeButton->setGeometry(271, 84, 85, 32);
setFixedSize(365, 240); //задаём фиксированный размер окна
приложения, т.е. окно нельзя теперь расширить или уменьшить
…
```

Абсолютное позиционирование имеет много недостатков:
• пользователь не может изменить размер окна;
• некоторый текст может оказаться отсеченным, если пользователь выбирает необычно большой шрифт или если приложение переводится на другой язык;
• виджеты могут иметь неправильные размеры для некоторых стилей;
• расчет позиций и размеров должен производиться вручную. Этот процесс утомителен и приводит к ошибкам.

В качестве альтернативы абсолютному позиционированию используется ручная компоновка. При ручной компоновке виджетам все же придаются абсолютные позиции, но размеры виджетов становятся пропорциональны размеру окна, а не жестко кодируются в программе. Это может достигаться путем переопределения функции формы resizeEvent() в которой программист сам задаёт приращения размеров и изменение координат элементов интерфейса при изменение размеров формы. Например, рассчитывать новые размеры виджетов можно следующим образом:

```
…
int extraWidth = width() - minimumWidth(); //высчитываем
приращение размеры формы к её минимальному размеру
int extraHeight = height() - minimumHeight();
Label->setGeometry(9, 9, 50, 25); //у метки будет фиксированный
размер и положение
LineEdit->setGeometry(65, 9, 100 + extraWidth, 25); //у текстового
поля – будет изменяться только ширина
CheckBox->setGeometry(9, 71, 156 + extraWidth, 23);
tableWidget->setGeometry(9, 100, 156 + extraWidth,
50 + extraHeight); //у таблицы – оба размера
closeButton->setGeometry(171 + extraWidth, 84, 85, 32); //а у
кнопки будет фиксированный размер, но она будет перемещаться, в
зависимости от размеров окна
…
```

Такой расчёт обеспечивает плавное изменение вида формы при изменении пользователем ее размеров. Однако, точно так же как при абсолютном

позиционировании, при ручной компоновке в программе приходится жестко задавать много констант, рассчитываемых программистом. А написание подобной программы представляет собой нудное занятие, особенно если проект изменяется. При этом все-таки существует риск отсечения текста. Этого риска можно избежать, принимая во внимание идеальные размеры дочерних виджетов, но это еще больше усложняет программу.

Использование классов компоновки (менеджеров компоновки)

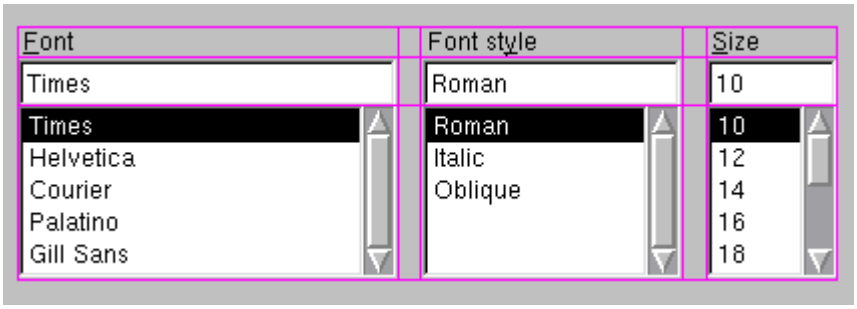
[Анг] http://doc.qt.digia.com/4.7-snapshot/qlayout.html
[Рус] http://doc.crossplatform.ru/qt/4.5.0/qlayout.html

Самый удобный метод компоновки виджетов на форме -использование менеджеров компоновки Qt. Менеджеры компоновки обеспечивают осмысленные принимаемые по умолчанию значения параметров для каждого типа виджета и учитывают идеальный размер каждого виджета, который в свою очередь обычно зависит от шрифта виджета, его стиля и содержимого. Менеджеры компоновки также учитывают максимальные и минимальные размеры и автоматически подстраивают компоновку в ответ на изменения шрифта, изменения содержимого и изменения размеров окна.

Существует три наиболее важных класса компоновки: QHBoxLayout, QVBoxLayout и QGridLayout. Эти классы происходят от QLayout, который обеспечивает основной каркас для менеджеров компоновки. Все эти три класса полностью поддерживаются Qt Designer и могут также использоваться непосредственно в программе. Например, можно скомпоновать некие элементы интерфейса следующим образом, создав диалог выбора шрифта:

```
...
QGridLayout *layout = new QGridLayout;
layout->setColumnMinimumWidth(15);
layout->addWidget(label1,0,0);
layout->addWidget(label2,0,2);
layout->addWidget(label3,0,4);
layout->addWidget(edit1,1,0);
layout->addWidget(edit2,1,2);
layout->addWidget(edit3,1,4);
layout->addWidget(list1,2,0);
layout->addWidget(list2,2,2);
layout->addWidget(list3,2,4);
...
```

Следующая иллюстрация показывает фрагмент такого диалога с сеткой, состоящей из пяти колонок и трех строк (сетка показана пурпурным цветом, на реальной форме, естественно, не отображается):



Колонки 0, 2 и 4 в данном диалоге содержат QLabel, QLineEdit и QListBox. Колонки 1 и 3 являются разделителями, заданными с помощью setColumnMinimumWidth(). Строка 0 содержит три объекта QLabel, строка 1 - из трех объектов QLineEdit, а строка 2 - из трех объектов QListBox. Использовались колонки-разделители (1 и 3) для установки правильного расстояния между колонками. Кромка по периметру диалогового окна и промежуток между дочерними виджетами устанавливаются в значения по умолчанию, которые зависят от текущего стиля виджета; они могут быть изменены, используя функции QLayout: :setContentsMargins() и QLayout::setSpacing().

Таким образом, при изменении размеров окна созданного диалога, все элементы будут перестраивать свои размеры и позиции автоматически.

Такое же диалоговое окно можно было бы создать с помощью визуальных средств разработки Qt Designer, задавая приблизительное положение дочерним виджетам, выделяя те, которые необходимо расположить рядом, и выбирая пункты меню Form| Lay Out Horizontally, Form| Lay Out Vertically или Form | Lay Out in a Grid.

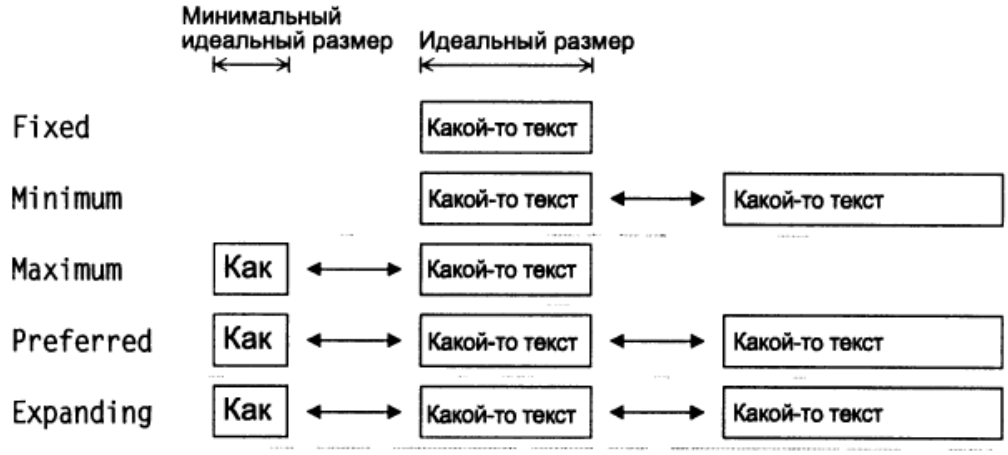
Также, использование менеджеров компоновки дает следующие дополнительные преимущества. Если добавить виджет к менеджеру или убрать виджет из него, менеджер компоновки автоматически адаптируется к новой ситуации. То же самое происходит, если вызвать метод hide() или show() для дочернего виджета. Если идеальный размер дочернего виджета изменяется, компоновка автоматически перестраивается, учитывая новый идеальный размер. Кроме того, менеджеры компоновки автоматически

устанавливают минимальный размер всей формы на основе минимальных размеров и идеальных размеров дочерних виджетов формы.

В рассмотренном примере было показано простое помещение виджета в компоновку. Иногда этого недостаточно для того, чтобы компоновка приняла нужный программисту вид. В таких ситуациях можно настроить компоновку, изменяя политику размеров и идеальные размеры размещаемых виджетов. Политика размера виджета говорит системе компоновки, как его следует растягивать или сжимать. Qt обеспечивает разумные принимаемые по умолчанию значения политик размеров для всех своих встроенных виджетов, но поскольку ни одно принимаемое по умолчанию значение не может учесть всевозможные варианты компоновки, все-таки обычной практикой для разработчиков является изменение политики размеров одного или двух виджетов формы. QSizePolicy имеет как горизонтальный, так и вертикальный компоненты. Ниже приводятся наиболее полезные значения:

• Fixed (фиксированное) означает, что виджет не может увеличиваться или сжиматься. Размер виджета всегда сохраняет значение его идеального размера.
• Minimum означает, что идеальный размер виджета является его минимальным размером. Размер виджета не может стать меньше идеального размера, но он может при необходимости вырасти для заполнения доступного пространства.
• Maximum означает, что идеальный размер виджета является его максимальным размером. Размер виджета может уменьшаться до его минимального идеального размера.
• Preferred (предпочитаемое) означает, что идеальный размер виджета является его предпочитаемым размером, но виджет может при необходимости сжиматься или растягиваться.
• Expanding (расширяемый) означает, что виджет может сжиматься или растягиваться, но в первую очередь он стремится увеличить свои размеры.

На следующем рисунке приводится иллюстрация смысла различных политик размеров, причем в качестве примера здесь используется текстовая метка QLabel с текстом «Какой-то текст».



На рисунке политики Preferred и Expanding представлены одинаково. Так в чем же их отличие? При изменении размеров формы, содержащей одновременно виджеты с политикой размера Preferred и Expanding, дополнительное пространство отдается виджетам Expanding, а виджеты Preferred по-прежнему будут иметь свой идеальный размер.

Существует еще две политики размеров: MinimumExpanding и Ignored. Первая была необходима в некоторых редких случаях для старых версий Qt, но теперь она не применяется; предпочтительнее использовать политику Expanding и соответствующим образом переопределить функцию minimumSizeHint(). Последняя напоминает Expanding, но при этом игнорируется идеальный размер виджета и минимальный идеальный его размер.

Стековая компоновка

[Анг] http://doc.qt.digia.com/4.7-snapshot/qstackedwidget.html

Класс QStackedLayout (менеджер стековой компоновки) управляет компоновкой набора дочерних виджетов или «страниц», показывая в каждый конкретный момент только одну из них и скрывая от пользователя остальные. Сам менеджер QStackedLayout невидим и не содержит внутри себя средства для пользователя по изменению страницы. Для удобства в Qt предусмотрен класс QStackedWidget, представляющий собой QWidget с встроенным QStackedLayout. Страницы в QStackedWidget нумеруются с 0. Если необходимо сделать какой-нибудь конкретный виджет видимым, можно вызвать функцию setCurrentIndex(), задавая номер страницы. Номер страницы дочернего виджета можно получить с помощью функции indexOf ().

Формы с QStackedWidget очень легко создавать при помощи Qt Designer.

1. Создайте новую форму на основе одного из шаблонов «Dialog» или на основе шаблона «Widget».
2. Добавьте в форму виджеты QListWidget и QStackedWidget.
3. Заполните каждую страницу дочерними виджетами и менеджерами компоновки. (Для создания новой страницы нажмите на правую кнопку мыши и выберите пункт меню Insert Page (вставить страницу); для перехода с одной страницы на другую щелкните по маленькой левой или правой стрелке, расположенной в верхнем правом углу виджета QStackedWidget.)
4. Расположите виджеты рядом, используя менеджер горизонтальной или люой другой компоновки.

Разделители

[Анг] http://doc.qt.digia.com/4.7-snapshot/qsplitter.html

Разделитель QSplitter представляет собой виджет, который содержит другие виджеты. Виджеты и разделители отделены друг от друга разделительными линиями. Пользователи могут изменять размеры дочерних виджетов разделителя посредством перемещения разделительных линий. Разделители могут часто использоваться в качестве альтернативы менеджерам компоновки, предоставляя пользователю больше возможностей по управлению компоновкой.

Области с прокруткой

[Анг] http://doc.qt.digia.com/4.7-snapshot/qscrollarea.html

Класс QScrollArea содержит область отображения, которую можно прокручивать, и две полосы прокрутки. Если необходимо добавить в виджет полосы прокрутки, значительно проще использовать класс QScrollArea, чем создавать свои собственные экземпляры QScrollBar и самим реализовывать функциональность скроллинга. Способ применения QScrollArea состоит в следующем: вызывается функция setWidget() с виджетом, к которому мы хотим добавить полосы прокрутки. QScrollArea автоматически делает этот виджет дочерним (если он еще не является таковым) по отношению к области отображения (он доступен при помощи функции QScrollArea::viewport()). По умолчанию полосы прокрутки видны на экране только в том случае, когда область отображения меньше дочернего виджета. Можно сделать полосы прокрутки постоянно видимыми при помощи установки следующих политик полос прокрутки:

```
scrollArea.setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
scrollArea.setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
```

Прикрепляемые окна и панели инструментов

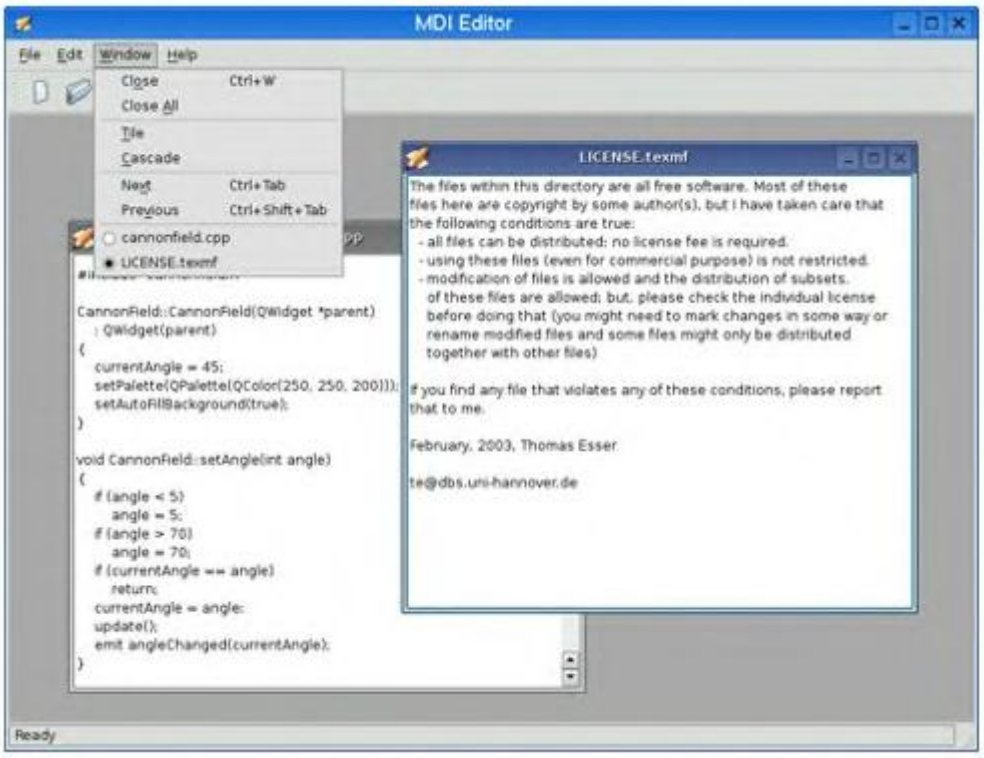
[Анг] http://doc.qt.digia.com/4.7-snapshot/qdockwidget.html

Прикрепляемыми являются окна, которые могут крепиться к определенным областям главного окна приложения, QMainWindow, или быть независимыми «плавающими» окнами. QMainWindow имеет четыре области для таких окон: одна сверху, одна снизу, одна слева и одна справа от центрального виджета. В таких приложениях, как Microsoft Visual Studio и Qt Linguist, широко используются прикрепляемые окна для обеспечения очень гибкого интерфейса пользователя. В Qt прикрепляемые окна представляют собой экземпляры класса QDockWidget.

Многодокументный интерфейс

[Анг] http://doc.qt.digia.com/4.7-snapshot/qmdiarea.html

Приложения, которые обеспечивают работу со многими документами в центральной области главного окна, называются приложениями с многодокументным интерфейсом или MDI-приложениями. В Qt MDI приложения создаются с использованием в качестве центрального виджета класса QMdiArea и путем представления каждого документа в виде дочернего окна QMdiArea. Обычно MDI-приложения содержат пункт главного меню Windows с командами по управлению как окнами, так и их списком. Активное окно отмечается галочкой. Пользователь может сделать любое окно активным, щелкая по его названию в меню Windows. В качестве примера приводится иллюстрация интерфейса написанного на Qt MDI-приложения:



Более подробная информация по работе с классом QMdiArea, а также с другими способами компоновки, и примерами реализации приведена в книге Жасмин Бланшет и Марка Саммерфилда «Qt4. Программирование GUI на C++» 2-е издание, 2008, в главе 6, части II.

Диалоговые окна

[Анг] http://doc.qt.digia.com/4.7-snapshot/dialogs.html

Диалоговые окна предоставляют пользователю возможность задавать необходимые значения параметров и выбирать определенные режимы работы. Они называются диалоговыми окнами или просто «диалогами» (dialogs), поскольку представляют собой средство, с помощью которого пользователи и приложения могут «переговариваться» друг с другом. Большинство приложений с графическим пользовательским интерфейсом имеют главное окно с панелью меню и инструментальной панелью, а также десятки диалоговых окон, естественно дополняющих главное окно. Можно также создать приложение из одного диалогового окна, которое будет непосредственно реагировать на выбор пользователя, выполняя соответствующие действия.

Класс QDialog

Класс QDialog является базовым для всех диалоговых окон, представленных в классовой иерархии Qt. Хотя диалоговое окно можно создавать при помощи любого виджета, сделав его виджетом верхнего уровня, тем не менее, удобнее воспользоваться классом QDialog, который предоставляет ряд возможностей, необходимых всем диалоговым окнам. Диалоговые окна подразделяются на две группы: модальные и немодальные. Режим модальности и немодальности можно установить и узнать при помощи методов QDialog::setModal() и QDialog::isModal() соответственно.

Модальные диалоговые окна

Такие окна обычно используются для вывода важных сообщений. Например, есть ошибки, на которые пользователь должен отреагировать, прежде чем продолжить работать с приложением. Модальные окна прерывают работу приложения и для продолжения его работы такое окно должно быть закрыто. В этих случаях модальный диалог — идеальное средство для обращения внимания пользователя на себя. Для блокировки приложения запускается цикл события только для диалогового окна, а события клавиатуры, мыши и других элементов приложения просто игнорируются. Этот цикл запускается вызовом слота exec(), который возвращает значение целого типа после закрытия диалогового окна. Это значение информирует о нажатой кнопке и может принимать значения QDialog::Accepted или QDialog::Rejected, соответствующие кнопкам Ok и Cancel (Отмена). Типичным примером модального окна является напоминание пользователю, перед закрытием приложения, о необходимости сохранения документа. В момент отображения этого окна возможность работы с самим приложением должна быть заблокирована.

Немодальные диалоговые окна

Немодальные диалоговые окна ведут себя как нормальные виджеты, не прерывая, при своем появлении, работу приложения. На первый взгляд может показаться, что применение немодальных диалоговых окон имеет больше смысла, так как в этом случае пользователь обладает большей свободой в своих действиях. Но, на самом деле, большинство приложений нуждается в остановке и ожидании решения пользователя для возобновления своих дальнейших действий. Немодальное окно может быть отображено с помощью метода show(), также как и для обычного виджета. Метод show() не возвращает никаких значений и не останавливает выполнение всей программы. Метод hide() позволяет сделать окно невидимым. Этим свойством можно

воспользоваться, и в этом случае не требуется создавать каждый раз объект диалогового окна, а при закрытии удалять его из памяти. Можно ограничиться вызовом методов show() и hide(), что даст возможность отобразить диалоговое окно, на том же месте, на котором оно было свернуто. Немодальные диалоговые окна необходимо снабжать кнопкой Close (Закрыть) для того, чтобы дать возможность пользователю закрыть его.

Стандартные классы диалоговых окон

В библиотеке Qt существуют следующие стандартные классы диалоговых окон:

● QFileDialog – предназначен для выбора одного или нескольких файлов, а также файлов, находящихся на удаленном компьютере, включает в себя возможность переименования файлов и создания директорий.

● QPrintDialog – позволяет выбрать принтер, изменить его параметры и задать диапазон страниц для печати.

● QColorDiaiog – диалоговое окно выбора цвета в Qt. Результатом выбора является класс QColor.

● QFontDialog – предназначен для выбора одного из зарегистрированных в системе шрифтов, а также для задания его стиля и размера.

● QInputDialog – диалоговое окно ввода данных можно использовать для предоставления пользователю возможности ввода строки или числа.

● QProgressDialog – информирует пользователя о начале продолжительной операции и дает возможность визуально оценить время работы.

● QMessageBox, QErrorMessage – это самый простой элемент пользовательского интерфейса, который отображает текстовое сообщение и ожидает реакции со стороны пользователя. Его основное назначение состоит в информировании о совершении определенного события.

Собственные диалоговые окна

Для создания собственного диалогового окна можно воспользоваться встроенными средствами Qt Creator и создать новый проект, выбрав в качестве исходного класса – QDialog, после чего уже размещать на нём различные

элементы интерфейса. А можно создать диалог программно, создав свой собственный класс наследованием от QDialog.

Технология drag-and-drop

[Анг] http://doc.qt.digia.com/4.7-snapshot/dnd.html
[Рус] http://qtdocs.narod.ru/4.1.0/doc/html/dnd.html

Технология «drag-and-drop» является современным и интуитивным способом передачи информации внутри одного приложения или между разными приложениями. Она часто является дополнением к операциям с буфером обмена по перемещению и копированию данных. Технология «drag-and-drop» состоит из двух действий: перетаскивание «захваченных» объектов и их «освобождение». Виджеты в Qt могут использоваться в качестве переносимых объектов, в качестве места отпускания этих объектов или в обоих качествах.

Функция dragEnterEvent() является обработчиком события и вызывается всякий раз, когда пользователь переносит объект на какой-нибудь виджет. Она обычно используется, чтобы сообщить Qt о типах данных, которые виджет готов принять. Т.е. в ней проверяют MIME-тип переносимого объекта. Например, MIME-тип text/uri-list используется для хранения списка унифицированных идентификаторов ресурсов (URI-uniform resource identifier), в качестве которых могут выступать имена файлов, адреса URL (например, адресные пути HTTP и FTP) или идентификаторы других глобальных ресурсов. Стандартные типы MIME определяются Агентством по выделению имен и уникальных параметров протоколов сети Интернет (Internet Assigned Numbers Authority - IANA). Они состоят из типа и подтипа, разделенных слешем. Буфер обмена и механизм «drag-and-drop» используют типы MIME для идентификации различных типов данных. Официальный список MIME-типов доступен по адресу http://www.iana.org/assignments/media-types/.

Функция dropEvent() – это тоже обработчик, который вызывается когда перетаскиваемый объект отпускается на неком виджете. Для того, чтобы разрешить отпускать переносимый объект на виджет, необходимо вызвать функцию acceptProposedAction() внутри dragEnterEvent(). По умолчанию виджет не смог бы принять переносимый объект. Qt автоматически изменяет форму курсора для уведомления пользователя о возможности или невозможности приема объекта виджетом.

Реализовав только dragEnterEvent() и dropEvent() можно позволить перетаскивание в виджет. Например, создав на форме объект класса-наследника от QTextEdit (текстовый виджет), и реализовав для него dragEnterEvent(), dropEvent() и функцию по загрузке данных, можно организовать «перетаскивание» текстового файла из среды Windows в программу.

Более подробная информация о технологии drag-and-drop приведена в книге Жасмин Бланшет и Марка Саммерфилда «Qt4. Программирование GUI на C++» 2-е издание, 2008, в главе 9, части II.

Интернационализация

[Анг] http://doc.qt.digia.com/4.7-snapshot/internationalization.html

Кроме латинского алфавита, используемого для английского и многих европейских языков, Qt 4 обеспечивает широкую поддержку остальным мировым системам записи:

1. Qt применяет Unicode в программном интерфейсе и во внутренних операциях. В приложении можно обеспечить всем пользователям одинаковую поддержку независимо от того, какой язык применяется в пользовательском интерфейсе;
2. Текстовый процессор Qt может работать со всеми основными нелатинскими системами записи, в том числе с арабской, китайской, кириллицей и др.; Qt учитывает различные особенности разных языков. Например процессор компоновки Qt обеспечивает компоновку справа налево для таких языков, как арабский и иврит;
3. Для определенных языков требуются специальные методы ввода текста. Такие виджеты редактирования, как QLineEdit и QTextEdit, хорошо работают в условиях применения любого метода ввода текста, существующего в системе пользователя.
4. В Qt существует возможность унифицированного перевода приложений на разные языки.

Рассмотрим эти основные возможности.

1) Unicode в Qt

Unicode является стандартной кодировкой, которая поддерживает большинство мировых систем записи. В основе кодировки Unicode лежит идея использования для хранения символов 16 бит, а не 8, и поэтому она позволяет закодировать примерно 65 000 символов (а при использовании 2-ух 16 битовых значений ещё больше) вместо только 256. Unicode содержит коды ASCII и ISO 8859-1 (Latin-1) в качестве своего подмножества с прежним их представлением. Класс QString хранит строковые значения в кодировке Unicode, а каждый символ QString имеет 16-битовый тип QChar, а не 8-битовый тип char. Любой из символов юникода можно задать константой, например латинскую букву A: str[0] = QChar(0x41); или символ Евро («€»): str[0] = QChar(0x20AC); (str – переменная типа QString). Все числовые коды, поддерживаемые кодировкой Unicode, можно найти по адресу http://www.unicode.org/standard/.

2) Текстовый процессор Qt

Текстовый процессор в Qt 4 поддерживает на всех платформах следующие системы записи: арабскую, китайскую, кириллическую, греческую, иврит, японскую, корейскую, лаосскую, латинскую, тайскую и вьетнамскую. Он также поддерживает все скрипты 4.1 в кодировке Unicode, которые не требуют специальной обработки. Кроме того, в системе X11 с Fontconfig и в последних версиях системы Windows поддерживаются следующие языки: бенгальский, деванагари, гуйарати, гурмухи, каннада, кхмерский, малайский, сирийский, тамильский, телугу, тхаана (дивехи) и тибетский. Наконец, ория поддерживается в системе X11, а монгольский и синхала поддерживаются в Windows XP. Если в системе установлен соответствующий шрифт, Qt сможет воспроизвести текст на любом из этих языков. А при установке соответствующих программ ввода текста пользователи смогут вводить в своих приложениях Qt текст на этих языках.

Некоторые языки, такие как арабский и иврит, используют запись справа налево, а не слева направо. Для таких языков общая компоновка приложения должна быть изменена на зеркальную, что делается при помощи вызова функции QApplication::setLayoutDirection(Qt::RightToLeft). Файлы перевода Qt содержат специальный маркер типа «LTR», указывающий Qt на направление записи используемого языка: слева направо или справа налево, и поэтому обычно не приходится отдельно вызывать функцию setLayoutDirection().

3) Различные методы ввода текста

При программном задании текста различных элементов интерфейса приложения, можно воспользоваться, например, статической функцией QString::fromUtf8(), аргументом которой будет строка, например на русском языке. Выведенный текст строки на экран будет отображён корректно, без символов-заменителей. Также, можно использовать класс QTextCodec:

```
QTextCodec *japaneseCodec = QTextCodec::codecForName("EUC-JP");
QTextCodec::setCodecForTr(japaneseCodec);
QString text = japaneseCodec->toUnicode("<здесь японские
иероглифы>");
```

Таким образом произойдёт смена кодировки и выводимый текст будет отображаться корректно.

4) Перевод приложений на другие языки

Разрешить пользователям вводить текст на их родном языке часто оказывается недостаточно; необходимо также перевести весь пользовательский интерфейс. В Qt это делается достаточно просто: все видимые пользователем строки обрабатываются функцией tr(), а затем используются утилиты Qt для подготовки файлов перевода на требуемый язык. Если разработчик хочет иметь многоязыковую версию своего приложения, он должен сделать две вещи:

• убедиться, что все строки, которые видит пользователь, проходят через функцию tr();
• загрузить файл перевода (.qm) при запуске приложения.

Функция tr() является статической функцией, определенной в классе QObject. При ее использовании в рамках подкласса QObject можно вызывать tr() без ограничений. Вызов tr() возвращает перевод строки, если он имеется, и первоначальный текст в противном случае.

Для подготовки файлов переводов необходимо запустить утилиту Qt lupdate. Эта утилита собирает все строковые константы, которые встречаются в вызовах tr() и формирует файлы переводов, содержащие все эти подготовленные к переводу строки. Эти файлы могут затем быть переданы переводчику для добавления к ним перевода строк. Перевод происходит при помощи утилиты Qt Linguist. После этого остается только при помощи утилиты lrelease получить двоичные файлы перевода .qm, и загрузить их в приложение. Обычно это

делается в главной функции приложения main() при помощи класса QTranslator (объект этого класса содержит набор переведённых строк приложения и предоставляет возможность навигации по ним) и QLocale (определяет пару страна-язык). Если пользователь переключается на другую локализацию во время выполнения приложения, то необходимо загрузить файлы перевода, соответствующие новой локализации, и вызвать функцию retranslateUi() для обновления интерфейса пользователя.

Более подробная информация об интернационализации приложений, а так же их конкретные примеры, приведены в книге Жасмин Бланшет и Марка Саммерфилда «Qt4. Программирование GUI на C++» 2-е издание, 2008, в главе 18, части III.

3. Пример создания GUI в Qt

Постановка задачи

В качестве примера приложения для реальной демонстрации некоторых возможностей библиотеки Qt по разработке GUI возьмём задание на курсовую работу по дисциплине «Графические системы» 9-го семестра обучения на кафедре Вычислительной Технике НИУ МЭИ.

Итак, необходимо: разработать программу в системе Unix, обладающую пользовательским интерфейсом, и обеспечивающую вывод определённой X-утилиты (xeyes) на экран, с заданными через этот интерфейс параметрами. Соответственно, интерфейс пользователя должен включать в себя: выбор цвета зрачка, цвета белка и цвета бровей «глаза», а также способ вывода утилит xeyes на экран: построчно (друг за другом заполняя некую область), крест-накрест и по спирали от центра. Программа должна также включать в себя возможность прерывания вывода X-утилит (очистка экрана).

Примечание: для разработки данного приложения использовались операционная система Linux Mint «Росинка» 11, Qt Creator 2.5.2 и версия библиотек Qt 4.8.3.

Создание проекта Qt

Для начала создадим проект в новом каталоге, например home/mikhail/Projects/qt_gui_gs. Для этого, запустив Qt Creator, перейдём на вкладку Файл → Новый файл или проект … → Приложения → GUI

Приложение Qt (Qt GUI Application) → Выбрать. Откроется диалог для задания первоначальных настроек проекта.

1) Размещаем проект в созданном нами каталоге home/mikhail/Projects/qt_gui_gs и называем его, например, gui1.

2) Выбираем модули библиотеки Qt, которые будут использоваться в проекте. В данном случае нас интересуют только QtCore и QtGui, которые выбраны по умолчанию.

3) Далее, зададим информацию о главном классе приложения. Выберем в качестве базового класса QWidget. QMainWindow является наследником QWidget, и представляет из себя «главное окно приложения», которое содержит настраиваемые главное меню, панель инструментов и т.д. QDialog – класс для создания отдельного окна-диалога с пользователем. QWidget же является исходным «чистым» классом для создания формы, который сам по себе, так же, в дальнейшем, можно встраивать в другие виджеты. Поэтому мы ограничимся выбором этого класса.
Зададим имя самого класса-наследника (Widget), имена файла-исходника (widget.cc) и заголовочного файла (widget.h), и поставим флаг «Создать форму».

4) Ещё есть возможность добавить проект в другой проект, или добавить его под контроль версий, но в данной работе это не требуется. Теперь, можно переходить, собственно, к написанию кода приложения.

Создание GUI

В данной работе мы будем рассматривать способ создания GUI программно, не используя встроенные возможности Qt Creator, т.к. эта задача сводится к интерактивному размещению на форме различных элементов интерфейса (таких как списки, кнопки и метки), путём простого перетаскивания их мышью на эскиз формы. Задание для них раскладок, настройка их внешнего вида, осуществляется путём редактирования свойств элементов в соответствующих диалогах Qt Creator-а, что мало отличается от похожего создания интерфейсов в других IDE. Создание же интерфейсов с помощью библиотеки Qt в коде программы, на ходу, просто и удобно, а также обеспечивает дополнительную гибкость в настройках элементов, и их дальнейшем манипулировании.

Итак, нам необходимо создать интерфейс пользователя, который обеспечивает возможность задавать различные параметры вывода X-утилиты Xeyes на экран, а именно: цвета зрачка глаза, белка глаза и бровей; способ вывода X-утилиты на экран: построчно, крест-накрест, по спирали. Для выбора параметра будем использовать списки, для отображения выбора – метки, а для запуска X-утилит или прерывания их вывода – кнопки. Из задания видно, что различных элементов интерфейса будет много, но некоторые из них будут обладать

одинаковыми свойствами. Поэтому объявим классы-наследники от стандартных классов Qt, которые будут представлять наши графические элементы интерфейса. Добавим в только что созданное объявление нашего класса Widget следующие строки:

```
//главный класс - виджет формы приложения
class Widget : public QWidget
{
   Q_OBJECT //макрос для объявления сигналов и слотов
   private:
    Ui::Widget *ui; //указатель на форму приложения
   public:
    explicit Widget (QWidget *parent = 0); //конструктор класса
    ~Widget(); //деструктор
    MColorListWidget *list1, *list2, *list3; //списки для выбора
цветов
    QListWidget *list4; //список для выбора типа отображения
    QPushButton *b1, *b2, *b3; //кнопки
    MHeaderLabel *ll1, *ll3, *ll5, *ll7; //метки-заголовки
    MColorLabel *ll2, *ll4, *ll6; //метки отображающие цвет
    MTypeLabel *ll8; //метка отображающая тип вывода
   public slots:
    //методы, которые вызывают кнопки:
    void mstart (); //вывод X-утилит на экран с соответствующими
параметрами
    void mbreak (); //закрытие всех X-утилит
    void mexit (); //выход из программы
    void mstartXeyes (int x, int y); //запуск одной утилиты Xeyes
};
```

QListWidget, QPushButton – это классы библиотеки Qt, описывающие элементы интерфейса Список и Кнопка соответственно. Для того, чтобы использовать эти классы, необходимо сначала добавить их в заголовочный файл директивой #include. Приведём список всех директив include, объявленных в файле widget.h:

```
#include <QWidget>
#include <QListWidget>
#include <QPushButton>
#include <QLabel>
#include <QHBoxLayout>
#include <QVBoxLayout>
#include <QErrorMessage>
#include <QFile>
#include <QProcess>
#include <QTextStream>
```

А вот классы MColorListWidget, MHeaderLabel, MColorLabel, MTypeLabel – это наша задумка. Их мы создадим наследованием от стандартных классов Qt и, для того чтобы они отвечали нашим требованиям, зададим для них свои

собственные конструкторы, новые методы и поля. Сигналы и слоты объявляются как показано в Части 1. Условимся называть все наши методы и классы, начиная с буквы m, чтобы отличать их от стандартных методов Qt. Добавим, выше определения класса Widget определения классов новых элементов интерфейса:

```
//метка-заголовок
class MHeaderLabel: public QLabel
{
    public:
     MHeaderLabel (QWidget *parent, QLayout *layout, QString
text); //конструктор инициализации
};


//метка отображающая цвет
class MColorLabel: public QLabel
{
    Q_OBJECT
    public:
     MColorLabel (QWidget *parent, QLayout *layout); //конструктор
инициализации
    public slots:
     void mchangeColor (QColor color, QString colorname); //меняет
цвет метки (а именно задаёт новую палитру) + добавляет название
};


//метка отображающая тип вывода X-утилит
class MTypeLabel: public QLabel
{
    Q_OBJECT
    public:
     MTypeLabel (QWidget *parent, QLayout *layout); //конструктор
инициализации
    public slots:
     void mchangeType (int num); //задаёт тип вывода X-утилит в
зависимости от выбранного из списка
};


//список для выбора цветов
class MColorListWidget: public QListWidget
{
    Q_OBJECT
    private:
     QList <QColor> colors; //массив кодов цветов из списка для
изменения цвета метки. Элементы массива соответствуют элементам
списка
    private slots:
     void mselectFromList (int num); //слот вызывается сигналом
нажатия на элемент списка
```

```
    public:
     MColorListWidget (QWidget *parent, QLayout *layout);
//конструктор инициализации
     bool mgetRgb (QString path); //заполняет список цветами из
файла rgb.txt
    signals:
     void mgetColor(QColor color, QString colorname); //Выбирает
цвет из массива по номеру + задаёт имя цвета
};
```

Разберём каждый класс отдельно и опишем его методы в файле widget.cc. Не следует забывать, что каждый элемент интерфейса в Qt – это тоже виджет, которому присущи так же как и всем некие общие свойства графического элемента, такие как, например, виджет-родитель, или возможность добавления в разметку.

Класс MHeaderLabel описывает заголовочную метку. Все заголовочные метки имеют один внешний вид, который описывается в конструкторе класса. Через параметр меняется только текст метки, а так же задаётся виджет-родитель и раскладка для этой метки (рассмотрим ниже). Многие параметры не задаются в конструкторе, например шрифт метки по умолчанию равен шрифту виджета-родителя.

```
//конструктор класса метки-заголовка
MHeaderLabel::MHeaderLabel(QWidget *parent, QLayout *layout,
QString text)
{
    setParent(parent); //задание класса-родителя для виджета.
Виджет будет размещён на нём.
    layout->addWidget(this); //добавляем виджет в заданную
параметром разметку
    setText(text); //задаём текст из параметра в конструкторе
    setAlignment(Qt::AlignCenter); //содержимое метки расположено
по-середине
}
```

Класс MColorLabel описывает метку для отображения цвета и его названия, выбранного в соответствующем списке. В конструкторе класса настроим внешний вид цветовой метки и также зададим для неё родительский виджет и разметку.
Метод mchangeColor () является слотом и вызывается в тот момент, когда испускается сигнал MColorListWidget:: mgetColor(). Их связь происходит в конструкторе главного класса Widget для каждого объекта этих классов. Т.е. при выборе из второго списка цветов, будет меняться цвет второй метки, соответствующей выбору цвета белка глаза. Код цвета и его название передаются через параметры.
Цвет метки задаётся палитрой (Часть 2, раздел «Виджеты»).

```
//конструктор класса метки цвета
MColorLabel::MColorLabel(QWidget *parent, QLayout *layout):
QLabel()
{
    setParent(parent);
    layout->addWidget(this);
    setAutoFillBackground(true); //фон метки будет заполняться
цветом
    setFrameShape(QFrame::Panel); //рамка метки
    setText("     "); //пустой текст - получится прямоуголник
    setAlignment(Qt::AlignCenter);
}


//метод-СЛОТ меняющий цвет метки
void MColorLabel::mchangeColor(QColor color, QString colorname)
{
    QPalette pal; //цвет меняется через смену палитры виджета
    pal.setBrush(QPalette::Background,color); //изменяем палитру:
роль - задний фон, цвет - задан параметром
    setPalette(pal);
    setText(colorname); //выводим название цвета в метку
}
```

Класс MTypeLabel описывает метку, отображающую тип вывода X-утилит. Сигнал выбора элемента из списка связывается со слотом отображения типа mchangeType() также в конструкторе класса Widget.

```
//конструктор метки типа вывода X-утилит
MTypeLabel::MTypeLabel(QWidget *parent, QLayout *layout): QLabel()
{
    setParent(parent);
    layout->addWidget(this);
    setText("1");
}


//метод-СЛОТ: отображение номера типа вывода X-утилит на метке
void MTypeLabel::mchangeType(int num)
{
    setText(QString::number(num+1)); //номер выбранного из списка
элемента - и есть тип отображения (+1)
}
```

Класс MColorListWidget описывает элемент интерфейса список, для выбора цвета определённой части X-утилиты. Объектов этого класса будет всего 3. В конструкторе класса помимо задания различных параметров внешнего вида списка, список заполняется строками с названиями цветов, которые берутся из стандартного файла цветов Linux rgb.txt. Для этого создан метод MColorListWidget::mgetRgb (). Если взять цвета из этого файла не удаётся, то список заполняется тремя цветами по умолчанию. Коды цветов сохраняются в

массив QList <QColor> colors, который представляет из себя объект шаблонного класса библиотеки Qt (Часть 1), и используются для задания цвета цветовой метке.
Объясним ещё раз связь событий и механизма сигналов/слотов для выбора цвета метки в нашей программе, поскольку это является важным моментом. Последовательность вызовов такова: Событие выбора определённого элемента списка → Срабатывает стандартный сигнал QListWidget::currentRowChanged(int) → Вызывается слот MColorListWidget::mselectFromList(int) → Внутри него испускается сигнал MColorListWidget::mgetColor(QColor, QString) → Он в свою очередь вызывает слот MColorLabel::mchangeColor(QColor,QString). Стандартный сигнал класса currentRowChanged() нельзя напрямую соединить со слотом задания цвета метки из-за разных параметров, но из-за того что последовательность вызовов сигналов/слотов может быть любой – эта проблема решается.
В методе mgetRgb() происходит загрузка списка цветов из файла rgb.txt при помощи класса QFile библиотеки Qt. Он обеспечивает простоту и скорость доступа к различным видам файлов, а также их обработки. Например, использование его совместно с классом QTextStream позволяет удобно считывать текстовый файл по строкам и манипулировать ими.

```
//конструктор списка цветов
MColorListWidget::MColorListWidget(QWidget *parent, QLayout
*layout): QListWidget()
{
    QErrorMessage *mes = new QErrorMessage;
    setParent(parent);
    layout->addWidget(this);
    setFont(parent->font());
    setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
    setMinimumWidth(150);
    if (mgetRgb("rgb.txt") == false)
    {
        colors.clear();
        addItem("Red");
        colors.append(Qt::red);
        addItem("Green");
        colors.append(Qt::green);
        addItem("Blue");
        colors.append(Qt::blue);
        mes->showMessage(QString::fromUtf8("Файл rgb.txt не найден
или некорректен. Заданы цвета по умолчанию!"));
    }
    //далее соединяем событие выбора элемента из списка со слотом,
который инициирует изменение цвета метки при помощи другого
сигнала
    connect(this, SIGNAL(currentRowChanged(int)), this,
SLOT(mselectFromList(int)));
}
```

```
//метод-СЛОТ: испускание сигнала, связанного со методом-слотом
изменения цвета метки
void MColorListWidget::mselectFromList(int num)
{
    //здесь num - уже текущий номер элемента из списка
    emit mgetColor(colors.at(num),this->item(num)->text());
//параметры передаются слоту
}


//заполняем список цветами из файла rgb.txt, учитывая его
структуру
bool MColorListWidget::mgetRgb(QString path)
{
    QString line; //объявляем временные переменные
    QColor color(0,0,0);
    QString name = "";
    QString col = "";
    QFile file(path);
    if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
        return false; //если файл не найден, или при открытии
возникли ошибки - возвращаем ложь
    QTextStream in(&file);
    line = in.readLine(); //считываем первую строку файла rgb.txt
с заголовком
    while (!file.atEnd()) //далее считываем все последующие строки
и выделяем из них код цвета и название
    {
        line = in.readLine();
        for (int k=-1; k<=7; k=k+4)
        {

col.append(line.at(k+1)).append(line.at(k+2)).append(line.at(k+3))
;
            col = col.trimmed(); //убираем пробелы, если они были
вместо цифры
            switch (k) //определяем какую из составляющих цвета
присваиваем сейчас
            {
                case -1: color.setRed(col.toInt()); break;
//преобразуем красную составляющую цвета из строки в число
                case 3: color.setGreen(col.toInt()); break;
                case 7: color.setBlue(col.toInt()); break;
            }
            col = "";
        }
        for (int k=11; k<line.size(); k++)
        {
            if (line.at(k).isLetterOrNumber())
name.append(line.at(k)); //накапливаем название цвета, добавляя по
букве/цифре
        }
```

```
            colors.append(color);
            addItem(name);
            name = "";
        }
        return true;
}
```

Опишем конструктор главного класса Widget. В нём создадим все разметки, участвующие в проектировании расположения элементов интерфейса программы. Их иерархия выглядит так: Основная горизонтальная разметка растянута на весь виджет. В неё по очереди, слева-направо добавляются вертикальные разметки содержащие в себе виджеты, расположенные сверху-вниз.
Сигнал класса MColorListWidget и слот MColorLabel соединяются также здесь, и их нельзя соединить в конструкторах этих классов, т.к. мы не знаем какие из объектов классов надо подсоединить друг к другу.
Методы по выводу X-утилит на экран, прерыванию их вывода и выходу из программы также являются слотами и подсоединены к сигналам clicked() нажатия кнопок.

```
Widget::Widget(QWidget *parent): QWidget(parent), ui(new
Ui::Widget)
{
    ui->setupUi(this); //устанавляваем интерфейс пользователя

    //определяем шрифт для всего виджета
    QFont font;
    font.setFamily("Segoe UI");
    font.setPointSize(12);
    this->setFont(font);

    //(1) Создаём основную разметку для виджета - горизонтальную
    QHBoxLayout* hor_main = new QHBoxLayout(this); //родительский
виджет разметки - главный виджет окна
    hor_main->setContentsMargins(10,10,10,10); //задаём отступы от
краёв содержимого разметки
    hor_main->setSpacing(10); //задаём промежутки между виджетами
внутри разметки

    //(2) Каждая группа виджетов будет находиться в своей разметке
- вертикальной,
    //// которая в свою очередь находится в главной горизонтальной
    QVBoxLayout* ver_1 = new QVBoxLayout(this);
    ver_1->setContentsMargins(0,0,0,0);
    ver_1->setSpacing(5);

    //(3) Создаём и размещаем на родительском виджете все дочерние
виджеты
    ll1 = new MHeaderLabel(this,ver_1,QString::fromUtf8("Цвет
зрачка:")); //используем констоукторы наших классов
    ll2 = new MColorLabel(this,ver_1);
```

```
    list1 = new MColorListWidget(this,ver_1);
    connect(list1, SIGNAL(mgetColor(QColor,QString)), ll2,
SLOT(mchangeColor(QColor,QString)));
    list1->setCurrentRow(0); //зададим выбор цвета по умолчанию
    hor_main->addLayout(ver_1); //добавляем вертикальную разметку
с только что созданными виджетами в горизонтальную


    //далее аналогично для остальных групп виджетов
    QVBoxLayout* ver_2 = new QVBoxLayout(this);
    ver_2->setContentsMargins(0,0,0,0);
    ver_2->setSpacing(5);

    ll3 = new MHeaderLabel(this,ver_2,QString::fromUtf8("Цвет
белка:"));
    ll4 = new MColorLabel(this,ver_2);
    list2 = new MColorListWidget(this,ver_2);
    connect(list2, SIGNAL(mgetColor(QColor,QString)), ll4,
SLOT(mchangeColor(QColor,QString)));
    list2->setCurrentRow(1);
    hor_main->addLayout(ver_2);


    QVBoxLayout* ver_3 = new QVBoxLayout(this);
    ver_3->setContentsMargins(0,0,0,0);
    ver_3->setSpacing(5);

    ll5 = new MHeaderLabel(this,ver_3,QString::fromUtf8("Цвет
бровей:"));
    ll6 = new MColorLabel(this,ver_3);
    list3 = new MColorListWidget(this,ver_3);
    connect(list3, SIGNAL(mgetColor(QColor,QString)), ll6,
SLOT(mchangeColor(QColor,QString)));
    list3->setCurrentRow(2);
    hor_main->addLayout(ver_3);


    QVBoxLayout* ver_4 = new QVBoxLayout(this);
    ver_4->setContentsMargins(0,0,0,0);
    ver_4->setSpacing(5);

    ll7 = new MHeaderLabel(this,ver_4,QString::fromUtf8("Тип
заполнения:"));
    ll8 = new MTypeLabel(this,ver_4);
    list4 = new QListWidget(this); //у этого списка уже
стандартный тип QListWidget, поэтому его заполнение делаем
отдельно
    ver_4->addWidget(list4);
    list4->addItem(QString::fromUtf8("Построчно"));
    list4->addItem(QString::fromUtf8("Крест-накрест"));
    list4->addItem(QString::fromUtf8("По спирали"));
    connect(list4, SIGNAL(currentRowChanged(int)), ll8,
SLOT(mchangeType(int)));
```

```
    list4->setCurrentRow(0); //зададим выбор типа вывода X-утилит
по умолчанию
    hor_main->addLayout(ver_4);


    QVBoxLayout* ver_5 = new QVBoxLayout(this);
    ver_5->setContentsMargins(0,0,0,0);
    ver_5->setSpacing(5);
    ver_5->setAlignment(Qt::AlignCenter); //добавляем: содержимое
разметки будет центрироваться

    b1 = new QPushButton(this);
    ver_5->addWidget(b1);
    b1->setText(QString::fromUtf8(" Запуск "));
    connect(b1, SIGNAL(clicked()), this, SLOT(mstart()));
//сигналы нажатия кнопок соединяем со слотами их обработки
    b2 = new QPushButton(this);
    ver_5->addWidget(b2);
    b2->setText(QString::fromUtf8(" Прерывание "));
    connect(b2, SIGNAL(clicked()), this, SLOT(mbreak()));
    b3 = new QPushButton(this);
    ver_5->addWidget(b3);
    b3->setText(QString::fromUtf8(" Выход "));
    connect(b3, SIGNAL(clicked()), this, SLOT(mexit()));
    hor_main->addLayout(ver_5);
}
```

Опишем методы нажатия на различные кнопки. Для вызова bash-команд из C++ кода Qt будем использовать статический метод QProcess::startDetached(), в параметре которого строкой указываем bash-команду с её параметрами. В ОС Linux этот метод создаёт отдельный процесс, который ведёт себя как daemon.

```
//вывод утилиты Xeyes с координатами x и y
void Widget::mstartXeyes(int x, int y)
{
    QProcess::startDetached( //X-утилита запустится в ОТДЕЛЬНОМ
процессе bash-командой xeyes с параметрами
     QString("xeyes -geometry 200x150+%1+%2 -fg %3 -center %4 -
outline %5") //высота утилиты Xeyes = 150 пикселей, длина = 200
     .arg(x).arg(y).arg(ll2->text()).arg(ll4->text()).arg(ll6-
>text())); //в формальные параметры строки подставляются
фактические имена цветов
}


//вывод X-утилит на экран с соответствующими параметрами
void Widget::mstart()
{
    int i=0, j=0;
    int type = ll8->text().toInt();
    //Будем заполнять "массив мест" X-утилит, размером 5x5
    if (type == 1) //заполняем построчно
```

```
    {
        for (i=0; i<5; i++)
        for (j=0; j<5; j++)
            mstartXeyes(i*200,j*150+j*28); //т.к. размер X-утилиты
- это размер окна без "шапки", надо добавить высоту шапки = 28
пикселей
    }
    else if (type == 2) //заполняем крест-накрест
    {
        for (i=0; i<5; i++) //просматривая весь массив проверяем,
...
        for (j=0; j<5; j++)
            if (i == j) mstartXeyes(i*200,j*150+j*28);
//...принадлежит ли элемент главной диагонали
            else if (j == (4 - i)) mstartXeyes(i*200,j*150+j*28);
//принадлежит ли элемент побочной диагонали
    }
    else if (type == 3) //заполняем по спирали. Порядок обхода:
слева-направо, сверху-вниз. Получится спираль от центра.
    {
        for (i=0; i<=4; i++) mstartXeyes(i*200,0);
        i--; //т.к. цикл for изменяет итератор ещё 1 раз после
завершения
        for (j=1; j<=4; j++) mstartXeyes(i*200,j*150+j*28);
        j--;
        for (i=3; i>=0; i--) mstartXeyes(i*200,j*150+j*28);
        i++;
        for (j=3; j>=2; j--) mstartXeyes(i*200,j*150+j*28);
        j++;
        for (i=1; i<=2; i++) mstartXeyes(i*200,j*150+j*28);
    }
}


//закрытие всех X-утилит
void Widget::mbreak()
{
    QProcess::startDetached("killall xeyes"); //прерываем все
процессы X-утилит bash-командой killall
}


//выход из программы
void Widget::mexit()
{
    this->close(); //закрываем виджет, а => и всё приложение
}
```

Главная программа

Файл, который так и называется main.cc, представляет из себя описание функции main, главной функции программы на C++. В нём происходит подключение заголовочного файла с нашим разработанным виджетом-формой, а так же первоначальные настройки окна приложения.

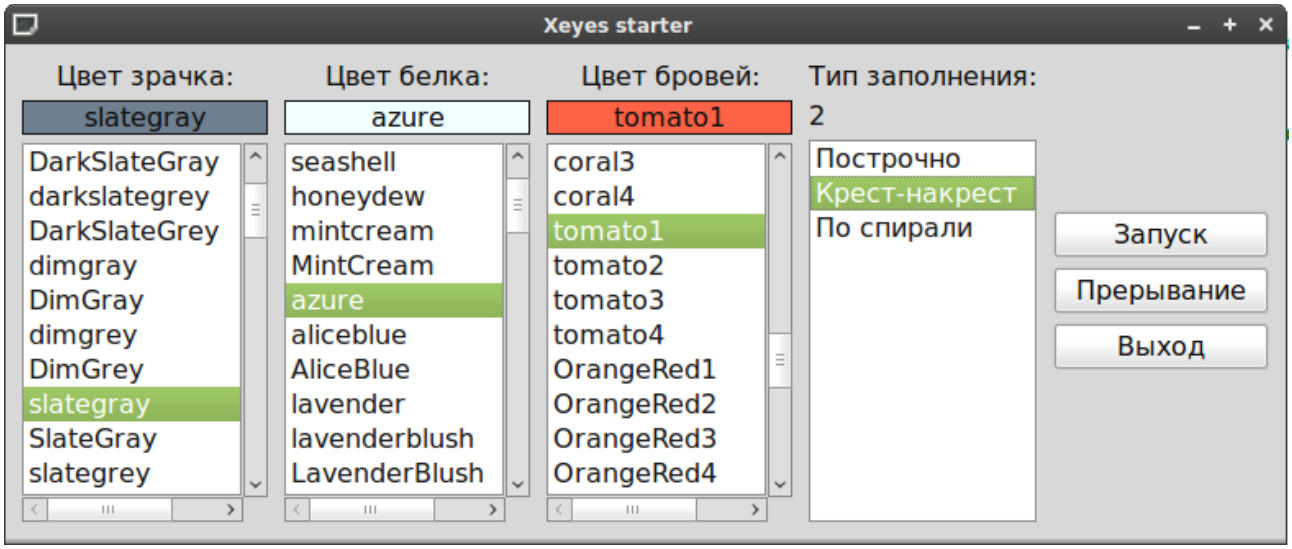
```
#include <QApplication>
#include "widget.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.setWindowTitle("Xeyes starter"); //задаём заголовок окна
    w.show();

    return a.exec();
}
```

Результат работы программы

Итак, в результате запуска программы мы получим следующий вид пользовательского интерфейса:



Запустив кнопкой «Запуск» вывод утилит Xeyes, мы увидим, что они выводятся с требуемыми параметрами: крест-накрест и соответствующими цветами глаз.

Кнопкой «Прерывание» можно завершить все созданные процессы xeyes, а по кнопке «Выход» - выйти из программы.

Литература и Интернет-ресурсы

1. Жасмин Бланшет, Марк Саммерфилд. Qt 4: Программирование GUI на C++, издание 2-е, дополненное, 2008. — 718 с.

2. http://qt.digia.com/ - официальный портал фреймворка Qt. Основная информация и свежие новости проекта. Здесь же можно скачать Qt SDK.

3. http://qt-project.org/downloads - официальная страница для скачивания open-source версии библиотеки и IDE Qt Creator.

4. http://doc.qt.digia.com/4.7-snapshot/index.html - официальная документация библиотеки Qt.

5. http://qtdocs.narod.ru/4.1.0/doc/html/index.html - неофициальный перевод документации на русский язык. Версия Qt 4.1. Перевод в процессе.

6. Макс Шлее, Qt 4.5, Профессиональное программирование на C++, 2010. — 884 с.

7. http://qt-doc.ru/ - множество хороших статей по разработке в Qt.

8. http://doc.crossplatform.ru/ - ещё один неофициальный перевод документации на русском языке. Переведено частично.