# Создание QApplication

первым делом инициализируются ресурсы

Q\_INIT\_RESOURCE(name) - call qInitResources\_<name>()

далее создаётся экземпляр QApplication - класс, обслуживающий GUI-приложение, поддерживающий жизненный цикл системных сообщений

QApplication инициализирует QCoreApplication, который сам инициализируется от QApplicationPrivate.

В конструкторе QApplicationPrivate ничего особого не происходит

Далее конструктор QCoreApplication - класса, являющимся наследником QObject, инициализирует последнего от QApplicationPrivate, а затем происходит иницаиализация диспетчера событий -- Qt реализовал его для платформ wintel, unix, symbian. При этом все диспетчеры событий наследуются от абстрактного диспетчера -- QAbstractEventDispatcher.

Класс QAbstractEventDispatcher осуществляет регистрацию сообщений, позволяет навешивать обработчики на сообщения конкретного типа, устанавливать на каждый тип сообщений таймер прослушивания.

Разговор о классе QObject пойдёт несколько позднее.

Примечание:

Для того, чтобы библиотеки Qt были бинарно совместимыми от версии к версии, Qt использует паттерн D-pointer. Суть паттерна в том, что бинарная совместимость нарушается, когда экспортируемые библиотеки меняют свой размер -- сдвигаются указатели на функции и всё ломается. Эту проблему можно обойти, если в экспортируемом классе хранить только один указатель неизменного размера -- d-pointer -- который ссылается на основную содержательную структуру. В Qt внутренние структуры экспортируемых классов снабжаются постфиксом Private, поэтому у всех выводимых наружу классов внутри будут располагаться Private-структуры, несущие основную смысловую нагрузку. Таким образом, можно, не меняя интерефейса, от версии к версии безболезненно менять реалиазцию. Поскольку этот паттерн используется во всех внешних классах Qt, то в дальнейшем описании я не буду упоминать конструирование Private-классов.

После инициализации менеджера графического приложения QApplication, создаётся, собственно, класс текстового редактора TextEdit. Он является наследником класса окна - QMainWindow.

Замечание. С точки зрения отношений классов весь редактор является окном. Такое решение является, безусловно, далеко не лучшим, поскольку нарушает принцип модульности -- нет даже простого разделения на модули логики работы с текстом и интерфейса. Это демонстрирует жёсткость связи, один из признаков простой архитектуры.

Итак, при конструировании TextEdit вызывается конструктор QMainWindow. QMainWindow в свою очередь наследуется от QWidget -- базового класса пользовательского интерфейса, который умеет рендерить в клиентскую область и получать сообщению от устройств ввода. В конструкторе виджета производится настройка параметров отображения -- фокусировка на окне, размер рабочей области, родительский виджет и т. п. Самым главным событием является посылка сообщения Create. Далее в теле конструктора QMainWindow прописывается layout -- размещение объектов интерфейса (меню, статус бар, тулбар, центральный виджет, вспомогательные виджеты) внутри данного виджета. Ясно, что топология структуры виджетов может быть произвольной - каждый виджет может включать в себя сколь угодно много других.

После вызова конструктора QMainWindow следует серия инициализаций пунктов панели меню -- за каждый пункт меню отвечает экземпляр класса QMenu -- и элементов тулбара, который во многом повторяет пункты меню -- за каждый элемент отвечает экземпляр класса QToolBar. Для каждой кнопки инициализируются QAction -- кликабельные элементы пользовательского интерфейса, которые можно подключить к конкретным пользовательским методам. Но как это делать? Пришло время рассказать про систему сигналов-слотов -- killing feature Qt.

Суть системы в следующем: если сделать класс наследником QObject'a и прописать в класс дополнительно несколько функций, необходимых для метакомпилятора (обёрнуты в макрос Q\_OBJECT), то мы получаем расширение для языка C++ - понятие слота и сигнала. Слот бывает публичным и приватным (public slot, private slot), сигнал всегда публичный. Сигнал -- это метод класса, не имеющий реализацию. Все сигналы объявляются под ключевым словом "signals:". Сигнал можно излучать (emit). Например, если в классе есть сигнал void Signal0( int a, float b ), то его можно излучить, например, в таком фрагменте кода:

void f( bool flag)

{

if (flag)

emit Signal0(30, 23.9f);

}

Излучённый сигнал попадает в слот -- нечто сродни callback-функции. При этом сигнатура сигнала и слота должна быть согласована: список аргументов сигнала всегда совпадает со списком аргументов слота, дабы метакомпилятор мог успешно передать параметры в слот.

Сигналы и слоты необходимо соединять. Для этого существует статический метод класса QObject QObject::connect. Он принимает указатель на объект, членом которого является сигнал, указатель на сигнал, указатель на объект, членом которого является слот, в который придёт излучаемый сигнал, и указатель на слот, а также тип подключения – синхронный или асинхронный.

У класса QAction есть сигнал triggered(), который излучается отслеживающим клики мыши экземпляр класом QMenu, в котором и сложены все экземпляры QAction, привязанные к данному меню. К каждой кнопке меню привязывается метод класса TextEdit, исполняющий соответствующее действие.

На каждый элемент панели меню написан свой метод

-- на пункт File и сопряжённые с ним элементы тулбара -- setupFileActions;

-- на пункт Edit и сопряжённые с ним элементы тулбара -- setupEditActions.

-- на пункт Format и сопряжённые с ним элементы тулбара -- setupTextActions.

Затем создаётся экземпляр QTextEdit -- это класс, который представляет собой виджет, позволяющий отображать plain-text и rich-text (с HTML-форматированием) и происходит связывание сигналов QTextEdit, излучаемых при изменении формата текста и позиции курсора, с соответствующими слотами.

QTextEdit является наследником QAbstractScrollArea – низкоуровневая абстракция прокручиваемой области. QAbstractScrollArea в свою очередь наследуется от QFrame – виджета с рамкой заданного стиля

При инициализации QTextEdit происходит инициализация класса QTextEditControl, который позволяет производить операции вставки и создания текста (класс QMimeData), соединение сигналов, отвечающих за фокусировку, изменения размеров текст и тд, с соответствующими слотами. Также создаётся экземпляр класса QTextDocument, внутри которого вызывается метод setHtml. Внутри данного метод создаётся экземпляр класса QTextHtmlImporter, отнаследованный от QTextHtmlParser. В конструкторе импортёра вызывается HTML-парсер, который строит DOM-дерево: дерево XML-узлов (QTextHtmlParserNode). Каждый узел имеет вид:

<имя\_тега атрибут1=»значение1» атрибут2=»значение2» атрибут3=»значение3»>текст</имя\_тега> -- *парный тег*

<имя\_тега атрибут1=»значение1» атрибут2=»значение2» атрибут3=»значение3» /> -- непарный тег

QTextCursor!

QTextDocumentFragment: rich text <-> plain text