1.2 Структура проекта Qt. Общие рекомендации по разработке проектов с использованием библиотеки Qt

В проект Qt входят файл конфигурации проекта (*.pro), исходные файлы (*.cpp) и заголовочные файлы (*.h). В начале файла конфигурации производиться подключение необходимых модулей. Для подключения модуля используется конструкция `QT +=` (например, `QT += coreguiwidgets`). В рамках данного пособия мы будем подробно говорить о следующих модулях Qt:

- **QtCore** – ядро Qt. Базовый модуль, который содержит класс QObject, контейнерные классы, классы для ввода и вывода, классы моделей интервью, классы для работы с датой и временем и др., но не содержит классов относящихся к интерфейсу пользователя.
- **QtGui** – модуль базовых классов для программирования пользовательского интерфейса. Содержит класс QGuiApplication, который поддерживает механизм цикла событий, позволяет получить доступ к буферу обмена, позволяет изменять форму курсора мыши и т.д.
- **QtWidgets** – модуль содержащий классы, которые являются “деталями” при реализации пользовательского интерфейса. В его состав входит класс QWidget – базовый класс всех элементов управления библиотеки Qt, классы автоматического размещения элементов, классы меню, классы диалоговых окон и окон сообщений, классы для рисования, а также классы всех стандартных элементов управления.
- **QtSql** – модуль отвечающий за поддержку работы с базами данных. Кроме перечисленных выше модулей библиотека Qt поддерживает также: QtNetwork для работы с сетями, QtOpenGL для работы с графикой OpenGL, QtTest содержащий вспомогательные классы для тестирования кода, QtXML отвечающий за поддержку XML и многие другие.

Далее в файле конфигурации проекта можно задать следующие параметры (в виде ПАРАМЕТР = ЗНАЧЕНИЕ):

TARGET – имя приложения. Если данное поле не заполнено, то название программы будет соответствовать имени проектного файла;

TEMPLATE – указывает тип проекта. Например: app – приложение, lib – библиотека.

FORMS –список файлов с расширением ui. Эти файлы создаются программой QtDesigner и содержат описание интерфейса пользователя в формате XML.

Кроме того, в файле проекта указывается список заголовочных файлов (ключевое слово HEADERS) и файлов исходного кода (ключевое слово SOURCES) в виде КЛЮЧЕВОЕ_СЛОВО += ИМЯ_ФАЙЛА1 ИМЯ_ФАЙЛА2 и т.д. При добавлении в проект файлов, QtCreator автоматически вносит изменения в файл проекта. Например, при добавлении в пустой проект HelloWorldфайла`main.cpp`, в файле HelloWorld.proпоявится строка `SOURCES += main.cpp`.

Также в файле проекта допустимо устанавливать следующие параметры:

LIBS – список библиотек, которые должны быть подключены для создания исполняемого модуля;

CONFIG – настройки компилятора;

DESTDIR – директория, в которую будет помещен исполняемый файл;

INCLUDEPATH – каталог, содержащий заголовочные файлы; и другие.

При разработке проекта файлы классов лучше всего разбивать на две отдельные части. Часть определения класса помещается в заголовочный файл *.h, а реализация класса – в файл с расширением .cpp. В заголовочном файле с определением класса необходимо указывать директиву препроцессора `#ifndef`или`#pragmaonce`. Например,

```
#ifndef _MyClass_h_
#define _MyClass_h_
classMyClass { ...
};
#endif //_MyClass_h_
```

Такой подход применяется во избежание конфликтов в случае, когда заголовочный файл включается в исходные файлы более одного раза. По традиции заголовочный файл, как правило, носит имя содержащегося в нем класса.

Для ускорения компиляции, при использовании в заголовочных файлах указателей на типы данных рекомендуется предварительно объявлять эти типы, а не напрямую включать их определения посредством директивы #include.

В случае, когда предполагается использование механизма сигналов и слотов или приведение типов при помощи функции `qobject_cast<T>()`, рекомендуется в начале определения класса указать макрос`Q_OBJECT`. Например,

```
classMyClass : publicQObject {
    Q_OBJECT public:
```

```
    MyClass();
    ...
};
```

Основную программу рекомендуется размещать в отдельном файле. В этом файле должна быть реализована функция `main`(). В связи с этим, в качестве имени для такого файла зачастую выбирают `main.cpp`.

1.3 Первая программа на QtПриступим к написанию первой программы на Qt. Создайте новый проект («Другой проект» –> «Emptyqmakeproject»). Проект назовите

HelloWorld, выберите компилятор и завершите создание проекта. В файл HelloWorld.pro запишите следующий код:

```
QT += coreguiwidgets

TARGET = HelloWorld
TEMPLATE = app
```

В контекстном меню проекта нажмите «Добавить новый» и выберите C++ SourceFile. Назовите его main. В проекте должен появиться каталог исходных файлов, содержащий main.cpp. Обратите внимание на то, что в файл HelloWorld.pro автоматически добавились следующие строки:

```
SOURCES += \
main.cpp
```

Это означается, что файл main.cpp, находящийся в корне каталога проекта, включен в проект. Теперь наполните файл main.cpp следующим содержимым:

```
#include<QtWidgets>
intmain(intargc, char** argv){
QApplicationapp(argc, argv);
QLabel* lbl = newQLabel("HelloWorld!");      lbl-
>show();
returnapp.exec();
}
```

В первой строке функции `main` создается объект класса `QApplication`, который осуществляет контроль и управление

приложением. Любая использующая Qt-программа с графическим интерфейсом должна создавать только один объект этого класса, и он должен быть создан до использования операций, связанных с пользовательским интерфейсом.

Затем создается объект класса `QLabel`. После создания элементы управления Qt по умолчанию невидимы, и для их отображения необходимо вызвать метод `show()`. Объект класса `QLabel` является основным управляющим элементом приложения, что позволяет завершить работу приложения при закрытии окна элемента.

Наконец, в последней строке программы приложение запускается вызовом `QApplication::exec()`. С его запуском приводится в действие цикл обработки событий, который передает получаемые от системы события на обработку соответствующим объектам. Он продолжается до тех пор, пока либо не будет вызван статический метод `QCoreApplication::exit()`, либо не закроется окно последнего элемента управления. По завершению работы приложения метод `QApplication::exec()` возвращает значение целого типа, содержащее код, информирующий о его завершении.

2 ПРОЕКТИРОВАНИЕ ГРАФИЧЕСКОГО ИНТЕРФЕЙСА ПОЛЬЗОВАТЕЛЯ

2.1 Механизм сигналов и слотов

Во время своей работы, приложение реагирует на различные события. Для того чтобы реализовать необходимую “реакцию” на определенное событие могут быть использованы различные подходы. В библиотеке Qt таким подходом является механизм сигналов и слотов. Сигналами в Qt являются методы, которые предназначены для пересылки сообщений. Слоты – это методы, которые описывают “реакцию” приложения на сигналы. Для использования механизма сигналов и слотов в описании класса, сразу на следующей строке после ключевого слова class, должен находится макрос `Q_OBJECT`, а сам класс должен быть унаследован от `QObject`. После макроса `Q_OBJECT` не должно стоять точки с запятой.

Сигналы описываются как пустые методы класса после ключевого слова “signals”. Например,

```
signals:
classMySignalClass : publicQObject { Q_OBJECT
```

```
    ... signals:
    voidmySignal();
    ...
};
```

Для подачи сигнала используется ключевое слово `emit`. Например, `emitmySignal()`.

Слоты в отличие от сигналов могут быть определены как `public`, `private` или `protected`. Соответственно, перед каждой группой слотов необходимо указать: `privateslots:`, `protectedslots:` или `publicslots:`. Например, `publicslots:`

```
classMySlotClass : publicQObject {
    Q_OBJECT …
    publicslots:
    voidmySlot() {
        …
    }
};
```

Каждый сигнал может быть связан с произвольным количеством слотов, а каждый слот – с произвольным количеством сигналов. Для связи сигнала со слотом используется метод `connect()` класса `QObject`.

```
QObject::connect(constQObject*   sender, constchar*
signal, constQObject* receiver, constchar*   slot,
    Qt::ConnectionTypetype = Qt::AutoConnection
    );
```

`sender` – указатель на объект, отправляющий сигнал; `signal` – это сигнал, с которым осуществляется соединение. Прототип (имя и аргументы) метода сигнала должен быть заключен в специальный макрос `SIGNAL(method());` `receiver` – указатель на объект, который имеет слот для обработки

сигнала; `slot` – слот, который вызывается при получении сигнала. Прототип слота должен быть заключен в специальном макросе `SLOT(method());`

`type`– управляет режимом обработки.

Пример: `QObject::connect(MySignalClass, SIGNAL(mySignal()), MySlotClass, SLOT(mySlot()));`

Если вызов метода `connect()` происходит из потомка класса `QObject`, тогда `QObject::` можно опустить. Если сигнал или слот находится в классе,

из которого происходит вызов connect(), то можно опустить первый или третий параметр соответственно. Например, если указанное выше соединение осуществляется в классе MySlotClass, то вызов connect() можно записать в следующем виде: connect(MySignalClass, SIGNAL(mySignal()), SLOT(mySlot())).

В некоторых ситуациях возникает необходимость в разъединение сигналов и слотов, для этих целей может быть использован статический метод disconnect() класса QObject, набор параметров которого аналогичен параметрам метода connect().

2.2 Класс QWidget

При проектировании графического пользовательского интерфейса с использованием библиотеки Qt основным “строительным материалом” являются виджеты. За работу с виджетами отвечает класс QWidget. Он унаследован от класса QObjectи является предком для таких элементов графического интерфейса как: главное и диалоговые окна приложения, кнопки, флажки, переключатели, слайдеры, полосы прокрутки, меню, выпадающие списки и многие другие. Наследование класса QObjectпозволяет использовать механизм сигналов и слотов, а также организовывать объектные иерархии. Элементы расположенные на виджете считаются его потомками. В иерархии такого рода виджеты, которые не являются потомками, называются виджетами верхнего уровня и обладают собственными окнами.

Конструктор класса QWidgetвыглядит следующим образом:

```
QWidget(QWidget* pwgt = 0, Qt::WindowFlags f = 0)
```

Первый параметр задает предка создаваемого виджета в объектной иерархии. Второй параметр отвечает за внешний вид окна. Туда могут быть переданы объединенные с типом окна побитовой операцией “|”. Для задания типа окна используются следующие константы: Qt::Window, Qt::Tool, Qt::ToolTip, Qt::Popup, Qt::SplashScreen; а в качестве модификаторов: Qt::WindowSystemMenuHint, Qt::FramelessWindowHint, Qt::WindowMinimizeButtonHintи другие. Как можно заметить, для получения виджета верхнего уровня мы можем использовать конструктор без параметров.

Основные методы класса QWidget:

- `setWindowTitle()` – устанавливает заголовок окна;
- `size()`, `height()` и `width()` – возвращают размеры виджета;
- `resize()` – позволяет установить или изменить размеры виджета;
- `pos()` – возвращает позицию окна;
- `move()` – перемещает окно;
- `setGeometry()` – объединяет методы `move`и `resize`.

2.3 Размещение элементов. Менеджеры компоновки

Для построения гибкого и легко масштабируемого графического интерфейса удобно использовать менеджеры компоновки, которые управляют размещением виджетов на окне. Корневым в иерархии классов отвечающих за работу менеджеров компоновок является `QLayout`. От него унаследованы классы `QBoxLayout`и`QGridLayout`. `QBoxLayout`является предком для классов `QHBoxLayout` и `QVBoxLayout`, которые отвечают за горизонтальное и вертикальное размещение виджетов соответственно. Класс `QGridLayout`в свою очередь обеспечивает размещение виджетов в виде таблицы.

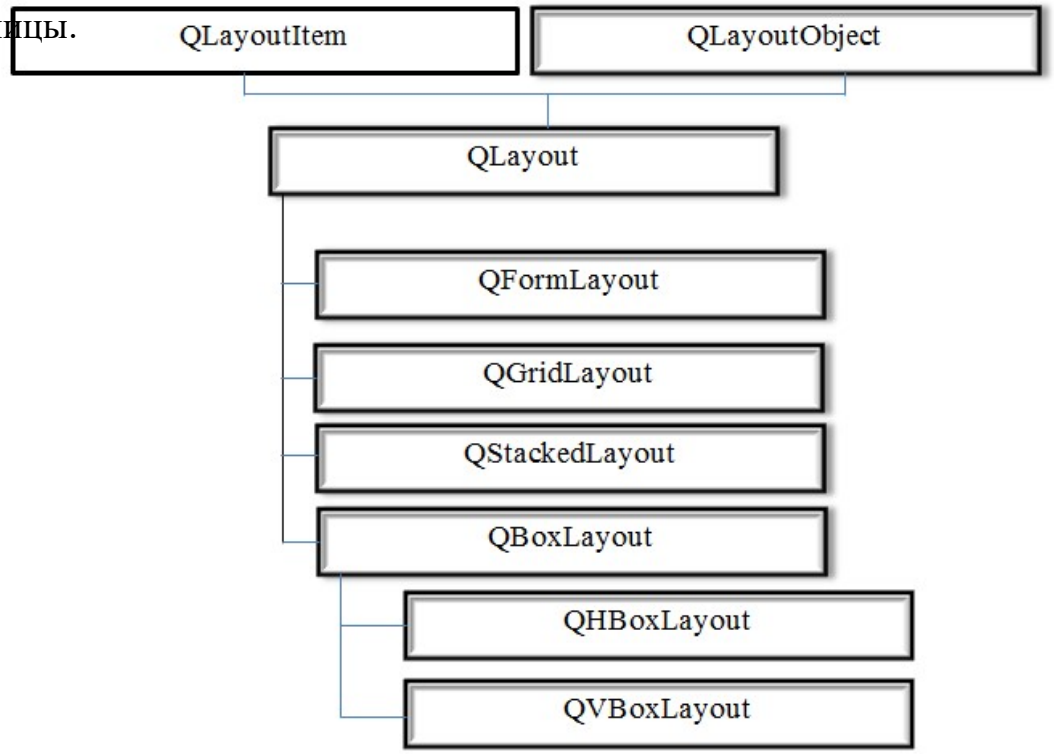

Рис 2.3.1 Иерархия классов менеджеров компоновок.

Некоторые общие методы менеджеров компоновок:

- `setSpacing()` устанавливает расстояние между виджетами;

• setMargin() устанавливает отступ виджетов от границы области;

addWidget() добавляет виджет в компановку;

• addLayout() позволяет встраивать другие менеджеры компоновок.

При использовании класса QBoxLayoutили его потомков, для указания способа размещения первым параметром в конструктор можно передать одно из следующих значений:

• LeftToRight для горизонтального размещения слева направо;

• RightToLeft для горизонтального размещения справа налево;

TopToBottom для вертикального размещения сверху вниз;

BottomToTop для вертикального размещения снизу вверх.

В табличном размещении (QGridLayout) при добавлении элемента указывается его позиция (номер строки и столбца). Для этого применяется метод addWidget(QWidget * widget, intfromRow, intfromColumn, introwSpan, intcolumnSpan, Qt::Alignmentalignment = 0). Параметры fromRowи fromColumnсодержат информацию о том, куда необходимо добавить элемент, параметры rowSpanи columnSpan – о количестве строк и столбцов, занимаемых данным элементом, а параметр alignment – о выравнивании.

2.4 Стандартные визуальные компоненты

Можно выделить следующие типы визуальных компонентов: элементы отображения, кнопки, флажки, переключатели, элементы настройки, элементы ввода и элементы выбора.

Элементы отображения. Основным классом для создания элемента отображения является QLabel. Отображаемая этим классом информация может быть текстовой, графической или анимацией. Для задания содержимого определены методы setText(),setPixmap() иsetMovie().

Кнопки, флажки и переключатели. Базовым классом для всех видов кнопок является класс QAbstractButton. От него наследуются следующие классы:

• QPushButton – кнопка, на которую можно нажать.

• QCheckBox – флажок, который имеет два состояния: выбран или не выбран.

• QRadioButton – переключатель, который также как и предыдущий имеет два состояния. Однако переключатели обычно формируются группами

по два и более и, при выборе одного из них, остальные автоматически входят в состояние “не выбран”.

• QToolButton – кнопка быстрого доступа. Обычно используется для добавления на панель инструментов.

Установить текст на кнопке можно либо передав его в конструктор, либо с помощью метода setText(). Кроме текста кнопке можно задать растровое изображение при помощи метода setIcon().

Класс QAbstractButton предоставляет следующие сигналы:

• pressed() – отправляется при нажатии на кнопку;

• released() –отправляется, когда “отпускаем” кнопку;

• clicked() – отправляется после того, как на кнопку нажали и отпустили;

• toggled() – отправляется при изменении состояния кнопки (только для кнопок имеющих статус переключателя).

Для класса QPushButton определен метод setCheckable(), который позволяет создать так называемый выключатель (togglebutton).

Для класса QCheckBox существует два основных вида флажков – обычный флажок (normalcheckbox) и флажок с неопределенным состоянием (tristatecheckbox). Состояние задается с помощью метода setChecked(). Флажки с неопределенным состоянием задаются при помощи метода setTristate(), а третье состояние задается передачей константы Qt::PartiallyChecked в метод setCheckState().

Как уже оговаривалось выше, переключатели QRadioButton группируются по два и более. Для подобного рода группировки удобно использовать класс QGroupBox.

Полосы прокрутки, слайдеры, установщики. Базовым для данных элементов настройки является класс QAbstractSlider. Для изменения ориентации следует применить слот setOrientation() и передать в него константу Qt::Horizontal или Qt::Vertical. Для установки диапазона значений используется метод setRange(), передав через запятую минимальное и максимальное значение, или методами setMinimum() и setMaximum(). Величина единичного шага задается методом setSingleStep(), а страничного – методом setPageStep(). Для получения значения слайдера применяется метод value(), а для установки значения – setValue(). Сигналы valueChanged() и sliderMoved() отправляются при изменении значения слайдера. От

QAbstractSliderунаследованы классы QSlider (ползунок), QScrollBar (полоса прокрутки) и QDial
(установщик).

Класс QSlider позволяет отображать метки (шкалы), которые дают пользователю визуальное представление о значении слайдера. Позиция меток задается методом setTickPosition() с помощью следующих констант: NoTicks (без меток), TicksAbove (сверху), TicksBelow (снизу) и TicksBothSides(сверху и снизу). Шаг для прорисовки меток устанавливается методом setTickInterval(). Применение класса QDial схоже с QSlider, но визуально первый представляет собой круглый регулятор.

Элементы ввода. Обычное однострочное поле ввода определяется классом QLineEdit, а многострочное – QPlainTextEdit. Получить текст из поля ввода можно методом text(), установить – методом setText(). При изменении текст отправляется сигнал textChanged().

Поля ввода со стрелками, предназначенными для увеличения или уменьшения значения, будем называть счетчиками. Абстрактным классом для счетчиков является QAbstractSpinBox. От него унаследованы классы QSpinBox (счетчик для целых значений), QDoubleSpinBox (счетчик для значений типа double) и QDateTimeEdit (счетчик для даты). Для всех видов счетчиков реализованы методы stepUp() и stepDown(), которые эмулируют нажатие на кнопки стрелок.

Элементы выбора. К элементам выбора относят простые и выпадающие списки. Простой список реализован в классе QListWidget. Элементы списка могут содержать текст и растровые изображения. Для добавления элемента в список можно воспользоваться методом addItem(). Элементы списка представлены классом QListWidgetItem. В список можно добавить сразу несколько текстовых элементов, передав объект класса QStringList, содержащий список строк, в метод insertItems(). Для вставки одного элемента (текстового или объекта класса QListWidgetItem) в произвольное место определен метод insertItem(). Стоит отметить, что элементами списка могут служить и другие виджеты. Для этой цели в классе QListWidgetопределены методы setItemWidget() и itemWidget().

2.5 Класс действия QAction. Панель инструментов. Меню. Главное окно приложения

Зачастую, при проектировании современного графического интерфейса, многие операции (например, копирование, вставка, сохранение и др.) доступны одновременно через главное меню, панель инструментов, контекстное меню, комбинации горячих клавиш и т.д. Для упрощения разработки подобных приложений в библиотеке Qt существует класс `QAction`. Этот класс хранит следующую информацию о действии:

- название операции (установка – метод `setText`);
- текст всплывающей подсказки (`setToolTip`);
- текст подсказки “Что это” (`setWhatsThis`);
- “горячие” клавиши (`setShortcut`);
- ассоциированные пиктограммы (`setIcon`);
- текст подсказки в строке состояния (`setStatusTip`).

Панель инструментов предназначена для быстрого доступа к необходимому действию. Класс панели инструментов в Qt – `QToolBar`. Установка панели инструментов осуществляется при помощи метода `addToolBar()`. Добавить действие на панель инструментов можно при помощи метода `addAction()`, а виджет – методом `addWidget()`.

За создание меню в Qt отвечает класс QMenu. Основные методы:

- addAction – добавить действие;
- addSeparator – добавить разделитель; addMenu – добавить подменю.

После создания меню, его можно поместить стандартную панель QMenuBar (метод addMenu).

Класс `QMainWindow` позволяет создавать главное окно приложения. Он содержит такие элементы как: главное меню, секции для панели инструментов, строку состояния, главную рабочую область. Указатель на виджет главного меню можно получить с помощью метода `menuBar()`, а на рабочую область – метод `centralWidget()`. Установить основной виджет рабочей области можно методом `setCentralWidget()`.

2.6 Лабораторная работа № 1 «Графический интерфейс»

Разработайте приложение, которое средствами библиотеки Qt предоставляет пользователю интерфейс и решает задачу вашего варианта. Для выбора действия пользователю должны быть предоставлены пункты меню и

кнопки панели инструментов. Эти элементы управления должны определяться объектами класса `QAction`.

Решение задачи оформить в виде отдельной функции, которая все необходимые ей данные получает в качестве параметров и не использует функций библиотеки Qt. Получение данных введѐнных пользователем и представление результата следует (но не обязательно) оформить в виде отдельных подпрограмм.

Функции Windows API не использовать и заголовочный файл windows.h не подключать.

Варианты заданий.

1. Даны три целых числа: *A*, *B* и *C*. Найдите среди них
   • наибольшее значение, • наименьшее значение.
2. Даны целые числа *A* и *B*. Не используя операторов «/», «%» необходимо вычислить
   • неполное частное, • остаток от их деления.
3. Даны комплексные числа *A* + *B***i** и *C* + *D***i**. Необходимо найти их
   • сумму, • разность.
4. Даны целые числа *A* и *B*. Необходимо вычислить их • наибольший общий делитель,
   • наименьшее общее кратное.
5. Даны целые числа *A*, *B* и *C*, проверить, можно ли из них составить арифметическую прогрессию. Если да, составьте из них
   • возрастающую,
   • убывающую прогрессию.
6. Даны дроби *A/B* и *C/D* . Необходимо найти их • произведение,
   • частное.
7. Даны вектора (*A*, *B*) и (*C*, *D*). Необходимо найти их
   • сумму, • разность.
8. Даны три целых числа: *A*, *B* и *C*. Найдите среди них
   • количество положительных значений,
   • количество отрицательных значений.
9. Даны целые числа *A* и *B*. Необходимо

• не используя оператор “*” вычислить их произведение,
• не используя функции “pow” вычислить A^B.

10. Даны целые числа A, B и C, проверить, можно ли из них составить геометрическую прогрессию. Если да, составьте из них
• возрастающую прогрессию,
• убывающую прогрессию.

11. Даны дроби A/B и C/D . Необходимо найти их
• сумму,
• разность.

12. Даны комплексные числа $A + B\mathbf{i}$ и $C + D\mathbf{i}$. Необходимо найти их
• произведение,
• частное.

13. Даны три положительных числа A, B и C. Проверьте, существует ли треугольник, у которого длины сторон равны данным числам. Если такой треугольник существует, то найдите его
• площадь,
• периметр.

2.7 Пример

Решим следующую задачу:

Даны целые числа A и B. Необходимо вычислить их
• *сумму,*
• *разность,*
• *произведение,*
• *частное.*

Создайте пустой проект с именем Lab1-example. В файл проекта Lab1-example.pro добавьте следующий код:

```
QT += coreguiwidgets

TARGET = Lab1-example
TEMPLATE = app
```

Основной функционал реализуем в классе Calculator. Добавьте к проекту заголовочный файл Calculator.h. В этот файл запишите следующий код:

```
#ifndef CALCULATOR
#define CALCULATOR

#include<QMainWindow>
//Перечисление используемых классов
classQLineEdit; classQComboBox;
classQLabel; classQPushButton;
classQWidget;
// ============================================
classCalculator : publicQMainWindow {
Q_OBJECT
private:
//Основной виджет приложения
QWidget *centralWidget;
//Поля для ввода чисел
QLineEdit* pleFirstNum;
QLineEdit* pleSecondNum;
//Выпадающий список для выбора операции
QComboBox* pcbAction; //Кнопка
«Выполнить»
QPushButton* ppbCalc;
//Статический текс с результатом вычислений QLabel*
plResult;
public:
Calculator(QWidget* pwgt = 0);
//Функции выполняющие вычисления
doubleadd(double a, double b);
doublesub(double a, double b);
doublemul(double a, double b);
doublediv(double a, double b);
publicslots:
//Слот нажатия на кнопку voidslotCalcClicked();
};
#endif //CALCULATOR
```

Добавьте к проекту файл Calculator.cpp, который будет содержать реализацию методов класса Calculator.

```
#include<Calculator.h>
#include<QtWidgets>
```

```
//Конструктор
Calculator::Calculator(QWidget* pwgt) :
QMainWindow(pwgt)
{
//Инициализация полей ввода     pleFirstNum
= newQLineEdit();     pleFirstNum-
>setMaximumWidth(30);     pleSecondNum =
newQLineEdit();     pleSecondNum-
>setMaximumWidth(30); //Инициализация
выпадающего списка действий     pcbAction =
newQComboBox();     pcbAction-
>setMaximumWidth(30);     pcbAction-
>addItems(QStringList() << "+" << "-" << "/"
<< "*");
//Инициализация кнопки     ppbCalc =
newQPushButton("=");     ppbCalc-
>setMaximumWidth(30);     connect(ppbCalc,
SIGNAL(clicked(bool)),
SLOT(slotCalcClicked()));
//Инициализация надписи для результата
plResult = newQLabel("0");     plResult-
>setMaximumWidth(60);
//Настройка размещения и установка центрального
виджета
QVBoxLayout* phbxLayout = newQVBoxLayout(this);
phbxLayout->addWidget(pleFirstNum);     phbxLayout-
>addWidget(pcbAction);     phbxLayout-
>addWidget(pleSecondNum);     phbxLayout-
>addWidget(ppbCalc);     phbxLayout-
>addWidget(plResult);
phbxLayout->setContentsMargins(100, 100, 100,
100);
centralWidget = newQWidget();     centralWidget-
>setLayout(phbxLayout);
setCentralWidget(centralWidget);
}
//Методы для вычисления результата
doubleCalculator::add(double a, double b)
{
return a + b;
}
doubleCalculator::sub(double a, double b)
```

```
{
return a - b;
}
doubleCalculator::mul(double a, double b)
{
return a * b;
}
doubleCalculator::div(double a, double b)
{
return a / b;
}
//Слот нажатия кнопки
voidCalculator::slotCalcClicked()
{
//Считывание информации из полей ввода double a
= pleFirstNum->text().toDouble();     double b =
pleSecondNum->text().toDouble();
doubleresult = 0; //Получение выбранной операции
QStringact = pcbAction->currentText();
//Вызов соответствующего метода     if (act
== "+") result = add(a, b);     elseif (act
== "-") result = sub(a, b);     elseif (act
== "/") result = div(a, b);     elseif (act
== "*") result = mul(a, b);     else {
plResult->setText("err");         return;
    }
//Отображение результата
plResult->setText((newQString)->setNum(result));
}
```

Теперь создадим файл `main.cpp`, содержащий функцию `main()`.

```
#include<QtWidgets>
#include "Calculator.h"
```

```
intmain(intargc, char** argv)
{
QApplicationapp(argc, argv);

Calculatorcalculator;
calculator.setWindowTitle("Calculator");
calculator.resize(300,300);
calculator.show();
returnapp.exec();
}
```

Запустите приложение и проверьте его работу.

Теперь реализуем поддержку действий, меню и панели инструментов. Для этого добавим в файл Calculator.h предварительное описание следующих классов:

```
classQAction;
classQMenu;
classQToolBar;
```

Кроме того, добавим описание следующих элементов:

```
//Описание объектов действий
QAction* pactAdd;
QAction* pactSub;
QAction* pactMul;
QAction* pactDiv;
//Описание объектов пунктов меню
QMenu* pmnuAdd;
QMenu* pmnuSub;
QMenu* pmnuMul;
QMenu* pmnuDiv;
QMenu* pmnuActions;
//Описание объекта панели инструментов
QToolBar* ptbTools;
```

и описание слотов для работы с действиями:

```
voidslotAdd();
voidslotSub();
voidslotMul();
voidslotDiv();
```

Теперь в реализацию конструктора класса Calculator добавим инициализацию действий, меню и панели инструментов.

```
//Сумма
pactAdd = newQAction(this);     pactAdd-
>setText("&+");
pactAdd->setShortcut(QKeySequence("CTRL++"));
pactAdd->setToolTip("Сумма");     connect(pactAdd,
SIGNAL(triggered()),
SLOT(slotAdd())); //Разность
pactSub = newQAction(this);     pactSub->setText("&-
");
pactSub->setShortcut(QKeySequence("CTRL+-"));
pactSub->setToolTip("Разность");     connect(pactSub,
SIGNAL(triggered()),
SLOT(slotSub())); //Произведение
pactMul = newQAction(this);     pactMul-
>setText("&*");
pactMul->setShortcut(QKeySequence("CTRL+*"));
pactMul->setToolTip("Произведение");
connect(pactMul, SIGNAL(triggered()),
SLOT(slotMul())); //Частное
pactDiv = newQAction(this);     pactDiv-
>setText("&/");
pactDiv->setShortcut(QKeySequence("CTRL+/"));
pactDiv->setToolTip("Частное");     connect(pactDiv,
SIGNAL(triggered()),
SLOT(slotDiv())); //Создание
меню
pmnuActions = newQMenu("&Действия");
pmnuActions->addAction(pactAdd);
pmnuActions->addAction(pactSub);
pmnuActions->addAction(pactMul);
pmnuActions->addAction(pactDiv);
menuBar()->addMenu(pmnuActions);
//Создание панели инструментов
ptbTools = newQToolBar();     ptbTools-
>addAction(pactAdd);     ptbTools-
>addAction(pactSub);     ptbTools-
>addAction(pactMul); ptbTools-
>addAction(pactDiv);
addToolBar(Qt::LeftToolBarArea, ptbTools);
```

Нам осталось только реализовать слоты для работы действий. Для этого мы воспользуемся уже реализованным слотом `slotCalcClicked()`, перед вызовом которого будем менять значение выпадающего списка `pcbAction` на соответствующее необходимому действию.

```
voidCalculator::slotAdd()
{
pcbAction->setCurrentText("+");
slotCalcClicked();
}
voidCalculator::slotSub()
{
pcbAction->setCurrentText("-");
slotCalcClicked();
}
voidCalculator::slotDiv()
{
pcbAction->setCurrentText("/");
slotCalcClicked();
}
voidCalculator::slotMul()
{
pcbAction->setCurrentText("*");
slotCalcClicked();
}
```