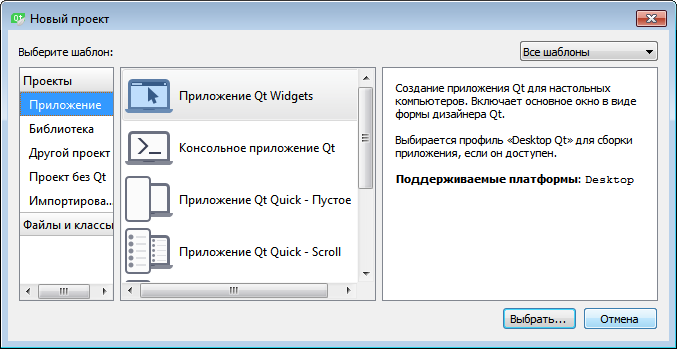
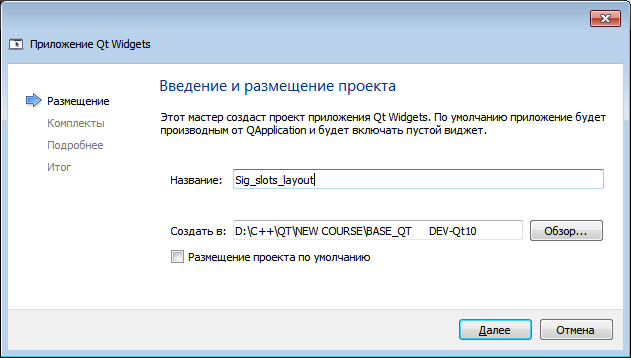
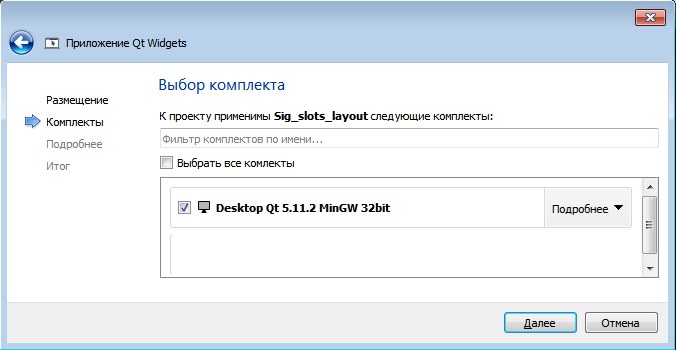
**Практика 3. Вариант B**

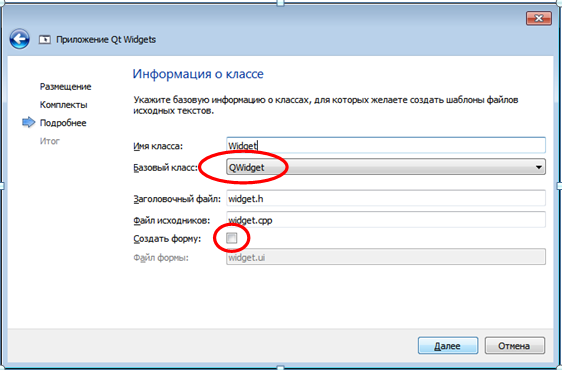
# Простейшие элементы управления (дочерние виджеты). Компоновка виджетов(Layout). Взаимодействие (синхронизация) элементов управления. Сигналы и слоты, предоставляемые классами Qt. Пользовательские сигналы/слоты. Валидаторы.

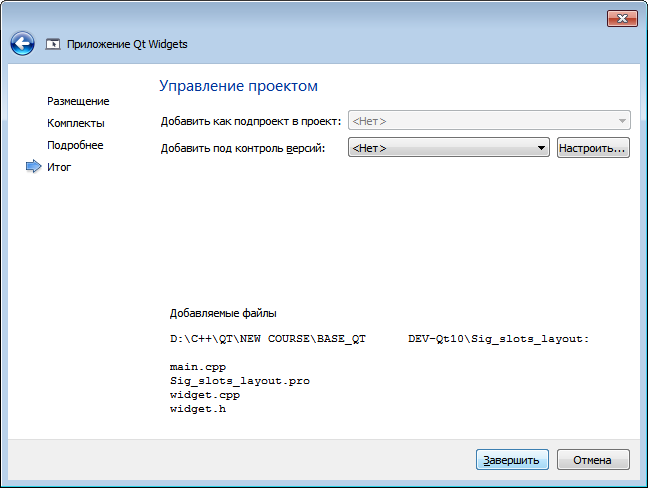
## Заготовка приложения







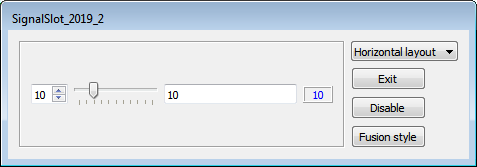


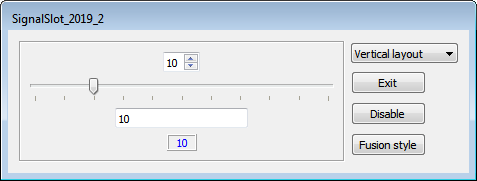


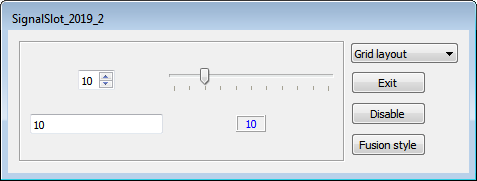
Можно протестировать заготовку.

## Задачи:

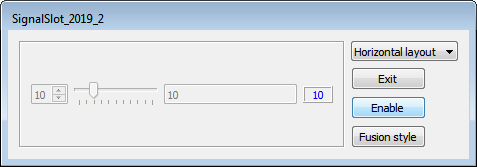
1. Создать приложение, которое позволяет пользователю разными способами (с помощью элементов управления различного типа: spinbox, lineedit, slider) задавать целое число в требуемом диапазоне (например, 1-99). Причем, элементы должны быть синхронизированы, т.е. при изменении значения посредством одного элемента, это измененное значение отображается другими элементами управления.
2. Размещением элементов управления (spinbox, lineedit, slider, label) на форме управляют менеджеры компоновки, которые можно менять с помощью comboBox.



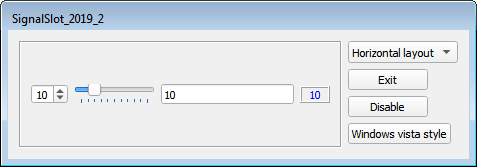




1. По нажатию кнопки “Disable” требуется сделать недоступными элементы панели



1. По нажатию кнопки “Fusion style” требуется заменить стиль отображения элементов управления на иной. (Дополнительное задание)

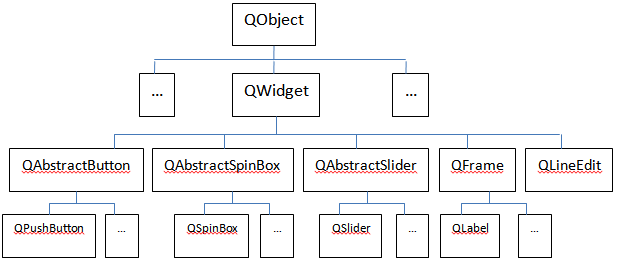


Замечание

Разработку приложения будем вести постепенно, усложняя и наращивая функциональные возможности.

## Элементы управления

Простейшими средствами взаимодействия пользователя с приложением являются: кнопка, поле редактирования, поле вывода, ползунок и наборный счетчик. Эти (и более сложные) элементы управления являются (обычно) дочерними виджетами главного окна приложения и поддерживаются соответствующими классами Qt:



**Важно!** в каждом классе поддержки функционирования элементов управления разработчики предусмотрели:

* Сигналы, которые генерируются при выполнении пользователем типичных для каждого элемента управления действий (например, для кнопки – «нажатие», для поля редактирования – изменение содержимого…)
* Слоты, которые принимают новое значение и отображают его.

### Кнопки

|  |
| --- |
| Кнопки бывают разного вида и назначения. Введем самую простую кнопку – “pushbutton”. Для создания кнопки имеется несколько конструкторов. Нам подойдет следующий: |
| [**QPushButton**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/Lab3_Qt10_задание/qpushbutton.html#QPushButton-1)(const QString &*text*, QWidget \**parent* = Q\_NULLPTR) |

При нажатии на кнопку наше приложение должно будет завершаться. Для обеспечения требуемого результата:

* В классе QPushButton определен сигнал **clicked()**, который генерируется при «нажатии» пользователем кнопки
* В классе QWidget определен слот **close()**. Если этот слот вызывается для главного окна приложения, осуществляется выход из цикла обработки событий.

Осталось только:

* создать собственно кнопку.
* «связать» сигнал со слотом посредством метода QObject::connect()

#include"widget.h"

#include<QPushButton>

Widget:: Widget(QWidget\*parent)

:QWidget(parent)

{

//Кнопка выхода

QPushButton \* buttonExit = new QPushButton(QObject::tr("Exit"),this);

//подумайте, почему можно не сохранять указатель в переменной класса?

//Кто будет освобождать память?

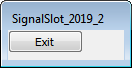
}

**//соединение создайте самостоятельно;**

Рекомендация: если Вы «новичок» в Qt, то советую начинать с простого (старинного)способа установки соединения, когда сигналы и слоты задаются в виде строк.

Протестируйте функциональность приложения.

1. Обратите внимание на расположение и размеры кнопки (они задаются по умолчанию) и размеры главного окна (выдается предупреждение «Unable to set geometry…» и размер окна «подстраивается» к размеру кнопки).
2. Имеет смысл настроить заголовок окна так, чтобы в нем отсутствовали стандартные кнопки.



### Создание других элементов управления

Задавать целое число в требуемом диапазоне разными способами можно с помощью элементов управления:

* поля ввода - **QLineEdit**
* наборного счетчика – **QSpinBox**
* ползунка–**QSlider**

Поле вывода (**QLabel**) будем использовать в учебных целях, панель (**QFrame**) - для размещения на ней ранее созданных элементов.

Для создания перечисленных элементов управления можно использовать конструкторы:

[**QLabel**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/Lab3_Qt10_задание/qlabel.html#QLabel)(QWidget \*parent = nullptr, Qt::WindowFlags f = Qt::WindowFlags());

[**QLabel**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/Lab3_Qt10_задание/qlabel.html#QLabel-1)(const QString &text, QWidget \*parent = nullptr, Qt::WindowFlags f = Qt::WindowFlags());

[**QLineEdit**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/Lab3_Qt10_задание/qlineedit.html#QLineEdit)(QWidget \**parent* = nullptr);

[**QLineEdit**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/Lab3_Qt10_задание/qlineedit.html#QLineEdit-1)(const QString &*contents*, QWidget \**parent* = nullptr);

[**QSlider**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/Lab3_Qt10_задание/qslider.html#QSlider)(QWidget \**parent* = nullptr);

[**QSlider**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/Lab3_Qt10_задание/qslider.html#QSlider-1)(Qt::Orientation *orientation*, QWidget \**parent* = nullptr);

[**QSpinBox**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/Lab3_Qt10_задание/qspinbox.html#QSpinBox)(QWidget \**parent* = nullptr);

[**Q**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/Lab3_Qt10_задание/qspinbox.html#QSpinBox)**Frame**(QWidget \**parent* = nullptr, Qt::WindowFlags f = Qt::WindowFlags());

**Замечания:**

1. **Напоминаем**: в конструкторе элементы управления требуется создавать **динамически**, иначе они прекратят свое существование после завершения работы конструктора главного окна.
2. Имеет смысл указывать родительский виджет при создании элемента, чтобы при разрушении родителя уничтожение дочерних окон происходило автоматически.
3. В качестве «родителя» панели указываем главное окно, а в качестве «родителя» других элементов управления – панель.

При конструировании элементов управления мы можем задать практически только «родителя», настройка специфических свойств осуществляется с помощью методов соответствующих классов.

На вопрос, **где сохранять указатели динамически созданных элементов** можно ответить так: если элемент управления потребуется в каком-либо еще методе, кроме конструктора, то надо соответствующий указатель сохранять в объявлении класса. В противном случае можно сохранять указатель в конструкторе, но включать элемент в объектную иерархию, чтобы он был уничтожен при разрушении родительского виджета.

Если мы хотим во время работы программы компоновать элементы ввода различным образом, то можно воспользоваться различными вариантами. Далее будет предложен один из вариантов решения.

Создается панель, на которой размещаются элементы управления. На панели устанавливается определенный компоновщик (горизонтальный, вертикальный, табличный), в который включаются элементы управления. В этом варианте сменными будут компоновки на единственной панели.

**Замечание**:

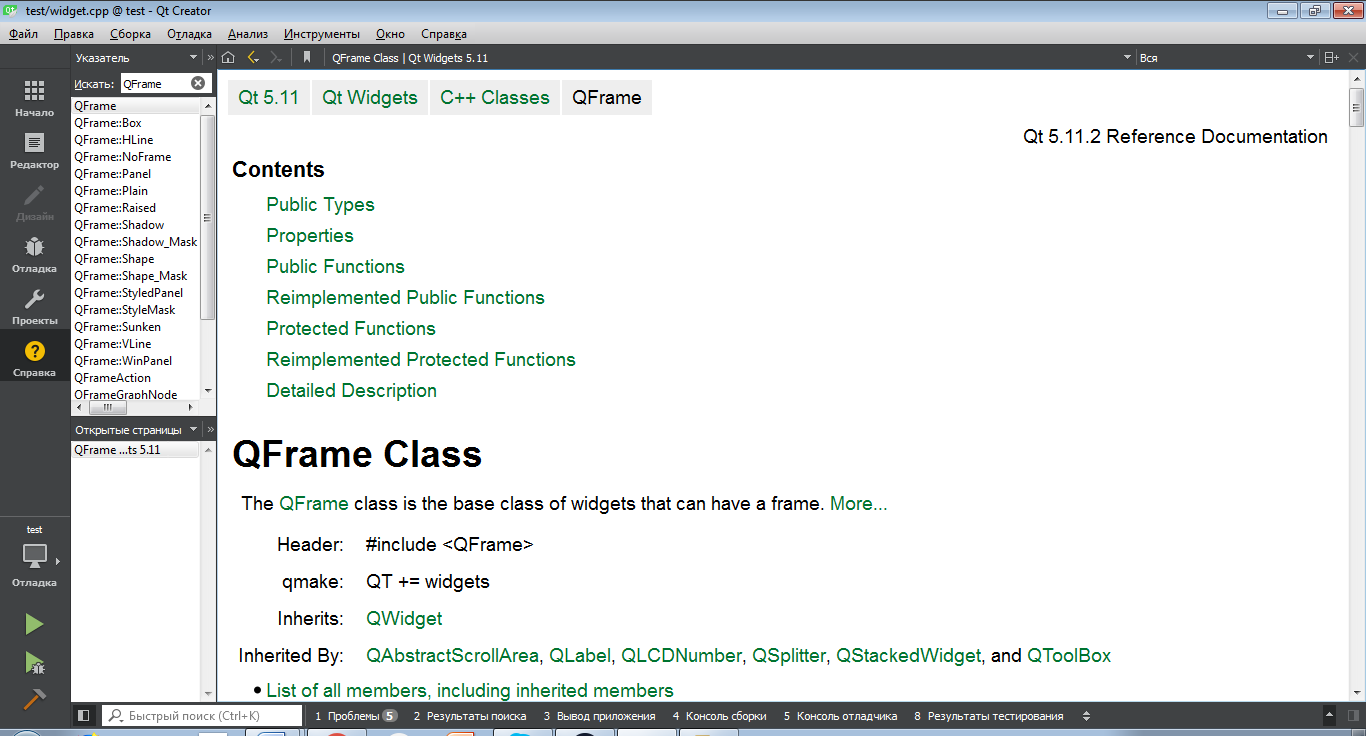
При создании элементов управления можно сразу указать для них родительский виджет (панель), а можно этого не делать. При добавлении элементов в компоновщик панели, они будут автоматически добавлены в виджет-панель.

Последовательность действий:

1. В конструкторе главного окна Widget создаем панель **QFrame .**
2. По умолчанию граница панели не отображается, поэтому следует явно задать стиль отображения. Это можно сделать с помощью метода:

**void** QFrame::**setFrameStyle**(**int** *style*).

Замечание: для лучшего ознакомления с классами библиотеки Qt пользуйтесь справкой, предоставляемой QtAssistant.



1. В конструкторе главного окна Widget создайте требуемые элементы управления (используя классы QLabel, QLineEdit, QSpinBox, QSlider).
2. Для того, чтобы код был более красивым, можно в классе Widget создать методы, в которых будет производиться создание и настройка элементов ввода. Например,

QLineEdit\* createLineEdit ();

QSpinBox\* createSpinBox();

//…

При этом будет «разгружен» конструктор и работать с кодом будет гораздо удобнее.

Пример кода:

#include<QPushButton>

#include<QSpinBox>

…

Widget::Widget (QWidget\*parent)

:QWidget(parent)

{

//Кнопка выхода

QPushButton\*buttonExit=new…

connect(…);

//Создание и настройка элементов управления

frame =new QFrame(this);

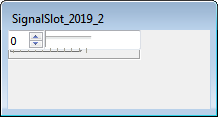
**edit = new QLineEdit(**frame**); //или без указания родителя**

//edit = createLineEdit(frame); // если заготовили в классе окна метод для //создания и настройки элемента

**}**

**Остальные виджеты(QSlider, QLabel, QSpinBox) создайте самостоятельно**

Для элементов управления по умолчанию устанавливается позиция (0,0) в координатах родительского окна. Поэтому, при запуске приложения Вы, скорее всего, увидите нечто подобное



Для того, чтобы сформировать более дружественный интерфейс, нам надо как-то иначе скомпоновать и настроить виджеты.

## Компоновки виджетов (layouts)

При создании пользовательского интерфейса программист должен решить:

* Какие виджеты потребуются для взаимодействия с пользователем
* Как их разместить на форме
* А уже в последнюю очередь, каким образом эти виджеты должны взаимодействовать между собой

Одним из способов формирования внешнего вида окна является компоновка виджетов (layout). Менеджер компоновки автоматически формирует для виджетов, включенных в компоновку:

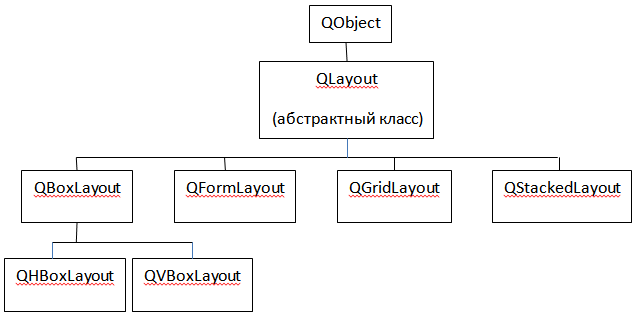
* размер
* расположение

При изменении размера окна менеджер компоновки может:

* изменять расположение виджетов относительно левого верхнего угла окна-родителя
* изменять размеры виджетов

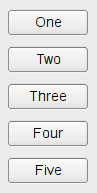
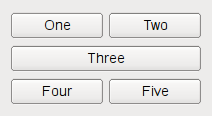
**подстраивая расположение и размеры к новому размеру окна, но сохраняя взаимное расположение виджетов**.

Qt предоставляет несколько классов для поддержки различных способов компоновки:



Подробную информацию об использовании менеджеров компоновки можно получить с помощью QtAssistent в разделе Layout Management.

Начнем изучение компоновок с наиболее распространенных:

* ****QHBoxLayout – размещает виджеты по горизонтали
* ****
* ****QVBoxLayout – размещает виджеты повертикали
* QGridLayout – размещает виджеты в ячейках сетки
* QFormLayout – специализирован для расположения в две колонки:
  + Справа – виджет, с которым будет взаимодействовать пользователь
  + Слева – ассоциированная с этим виджетом надпись (label)  
    добавляются виджетыв такую компановку специализированным методом QFormLayout::addRow()
* QStackedLayout – поддерживает такую компоновку, при которой только один виджет является видимым (это альтернатива «закладкам»).

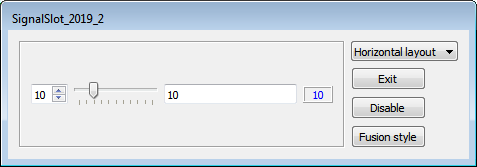
**Важно!**

* Виджет, на котором установлена компоновка, становится **владельцем** (родителем) компоновки
  + в процессе вызова для него setLayout()
  + или посредством параметра конструктора при создании объекта-компоновки
* Все виджеты, добавляемые в компоновку посредством addWidget(), автоматически становятся дочерними виджетами **окна-владельца компоновки**
* Компоновки могут быть вложенными (таким образом можно создавать сколь угодно сложные совокупности виджетов). Добавить одну компоновку в другую можно посредством методов   
  QLayout::addItem()  
  или

QBoxLayout::addLayout() и QGridLayout::addLayout()

### Проектирование интерфейса приложения

Нам в конечном итоге хотелось бы видеть такое:

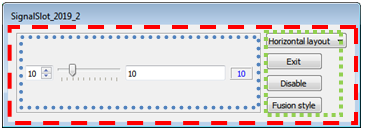


Потребуются дополнительно:

* comboBox для управления переключением компоновок ;
* Кнопка, управляющая состоянием Enabled/Disabled панели.
* Кнопка, управляющая переключением стиля отображения (Дополнительное задание).

Кроме того, нам потребуются следующие компоновки:

1. Вертикальная (QVBoxLayout) для размещения кнопок «Exit», «Disable», «Style».
2. Сменные горизонтальная (QНBoxLayout), вертикальная (QVBoxLayout), табличная (QGridLayout) для размещения на панели элементов управления (spinbox, lineedit, slider, label). На панели одновременно можно установить только одну компоновку.
3. Горизонтальная (QНBoxLayout), которая будет включать панель и вертикальную компоновку с кнопками. Эта компоновка устанавливается на главном окне.



### QHBoxLayout и QVBoxLayout

Вертикальную и горизонтальную компоновки поддерживают объекты **разных** классов: QHBoxLayout и QVBoxLayout.

Попытаемся скомпоновать виджеты нужным образом. Важно помнить, что на виджете можно установить только одну дочернюю компоновку, остальные (вложенные) компоновки становятся дочерними по отношению к главной компоновке виджета.

|  |  |
| --- | --- |
| **Виджет/компоновка** | **Состав** |
| Главное окно (Widget) | Главная компоновка окна (горизонтальная) |
| Панель (QFrame) | Главная компоновка панели (горизонтальная) |
| Главная компоновка окна | * Панель * Компоновка с кнопками «exit»,.. (вертикальная) |
| Главная компоновка панели | Компоновка с элементами ввода (Сменная)   * Строка редактирования * Метка * Наборный счетчик * Слайдер |
| Компоновка с кнопками «exit», … | * Кнопка «exit» * Кнопка «Disable» * Кнопка «… style» (Дополнительное задание) |

Последовательность действий:

1. Сначала подготовим объект главной компоновки окна. В конструкторе Widget динамически создайте объект **главной** компоновки **окна**. Для того, чтобы эта компоновка стала **главной** компоновкой виджета, ей можно задать родителя посредством параметра конструктора

**QHBoxLayout**::**QHBoxLayout**(QWidget \**parent*);

или сделать то же самое позже с помощью метода setLayout():

**void** QWidget::**setLayout**(**QLayout** \**layout*);

1. Далее надо правильно скомпоновать элементы управления на панели. В конструкторе Widget динамически создайте объект **главной** компоновки **панели**. Компоновка панели в процессе работы программы будет заменяться другой. Для определенности пусть первоначально на панели будет установлена горизонтальная компоновка.

Так как эта компоновка будет **главной компоновкой панели**, то имеет смысл указать панель в качестве родителя этой компоновки.

Рекомендация: т.к. компоновка на панели будет заменяться неоднократно, то удобно для ее создания использовать соответствующий метод, например:

QLayout\* createHBoxLayout();

в котором производить настройку компоновки

1. Добавьте элементы ввода/вывода (spinbox, lineedit, slider, label) в горизонтальную компоновку (главную компоновку панели)

Для этого можно использовать метод:

**void** QBoxLayout::**addWidget**(QWidget \* widget, int stretch = 0, Qt::Alignment alignment = Qt::Alignment()).

Выравнивание по умолчанию равно 0, что означает, что виджет заполняет всю ячейку.

1. В конструкторе Widget динамически создайте объект вложенной вертикальной компоновки с кнопками. При создании объекта компоновки указывать родителя можно только для главных компоновок виджета, остальные (вложенные) компоновки создаются БЕЗ родителя.
2. Добавьте в вертикальную (кнопочную) компоновку кнопки «Exit» и «Disable», «… style». Для этого можно использовать метод:

**void** QBoxLayout::**addWidget**(QWidget \* widget, int stretch = 0, Qt::Alignment alignment = Qt::Alignment ()).

Очень важно! Кнопочная компоновка будет вложенной для главной компоновки окна

1. Добавьте в главную (горизонтальную) компоновку окна приложения:

* Панель
* Вертикальную компоновку с кнопками «Exit» и «Disable» , «… style».

Замечания:

1. Элементы в компоновке располагаются в том порядке, в каком добавлялись.
2. (существенное замечание):

для вложенной компоновки не следует задавать в качестве владельца виджет, иначе при запуске программы будет выдано сообщение о попытке проассоциировать дочернюю компоновку с виджетом, у которого уже есть главная компоновка:  
QLayout: Attempting to add QLayout "" to Widget "", which already has a layout

QLayout::addChildLayout: layout "" already has a parent

1. Для того, чтобы отслеживать как формируется иерархия объектов, можно использовать метод класса void QObject::**dumpObjectTree**() const.
2. А чтобы сделать более понятным вывод этого метода, можно объектам присвоить имена с помощью метода

**void** QObject::**setObjectName**(QString);

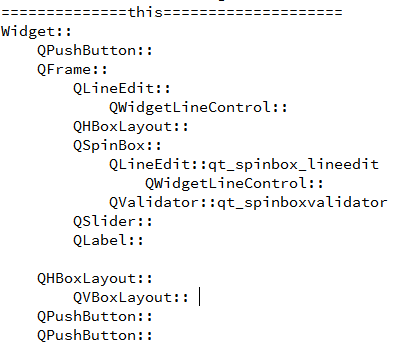
Например, buttonExit->setObjectName(“ButtonExit”);

Запускаем приложение и любуемся на достигнутый резульнтат.

Graphical user interface, application

Description automatically generated

При этом метод **QObject::dumpObjectTree()** выдаст следующую информацию об объектной иерархии:



Замечание:

Оказывается, что если используются менеджеры компоновки, то необязательно указывать родителя при создании элемента управления. При добавлении внутренней компоновки в главную компоновку:

* Внутренняя компоновка становится дочерней для главной компоновки
* А все виджеты, добавленные во внутреннюю компоновку, автоматически становятся дочерними для виджета, на котором установлена главная компоновка.

## Настройка внешнего вида элементов управления

При создании элементов управления мы не указывали никаких специфических свойств, поэтому значения этих свойств были сформированы по умолчанию. Поэтому слайдер мог быть расположен вертикально (если вы не указали иное при конструировании), а метка вовсе не отображается.

В частности, виджету задаются наиболее «подходящие» (разумные) размеры и «стратегии растяжения», которые используются менеджером компоновки. (Размеры можно узнать с помощью метода **sizeHint()**, а «стратегии растяжения» - помощью метода **sizePolicy()**). При первоначальном знакомстве с компоновками мы будем полагаться на значения sizeHint и sizePolicy, заданные по умолчанию.

Однако, другие умолчания подходят далеко не для каждой задачи =>нужно уметь задавать **специфические** свойства элементов управления.

### Наборный счетчик – QSpinBox

* Задание диапазона. По умолчанию диапазон 0-99, а для нашей задачи требуется 0-50  
  void QSpinBox::setMaximum ( int max ) //если устраивает нижняя граница == 0  
  или void QSpinBox::setRange ( int minimum, int maximum )
* начальное значение – setValue(int)

Замечание:

Для того, чтобы spinbox не «растягивался», а размеры его оставались наиболее «разумными», можно задать ему фиксированный размер следующим образом:

spin->setFixedSize(spin->sizeHint());

// метод sizeHint() как раз и предоставляет наиболее «подходящие» размеры виджета

Задайте диапазон наборного счетчика и при желании – начальное значение

### Ползунок –QSlider

Для ползунка можно задать:

* свойство горизонтальный/вертикальный - Qt::Horizontal или Qt::Vertical

в конструкторе QSlider QSlider::**QSlider**([Qt::Orientation](qthelp://org.qt-project.qtwidgets.502/qtcore/qt.html#Orientation-enum)*orientation*, [QWidget](qthelp://org.qt-project.qtwidgets.502/qtwidgets/qwidget.html) \**parent* = 0)  
или в методе void QAbstractSlider::**setOrientation** (Qt::Orientation).

* диапазон значений, в которые будет пересчитываться позиция ползунка  
  QAbstractSlider::**setRange(**), QAbstractSlider::**setMaximum(),** QAbstractSlider::**setMinimum()**
* начальную позицию  
  QAbstractSlider::**setSliderPosition()**
* вид «тиков» ползунка (по умолчанию тики **не** рисуются)  
  QSlider::**setTickPosition()**
* величину деления QSlider::**setTickInterval()**
* величину шага (при единичном нажатии стрелки на полосе прокрутки или с помощью клавиатуры) - QAbstractSlider::**setSingleStep()**
* минимально возможную ширину QWidget::**setMinimumWidth**();
* …

Задайте диапазон, внешний вид и свойства ползунка

### Поле редактирования - QLineEdit

Qt предоставляет несколько классов для приема пользовательского текста. Самым простым средством является элемент управления **QLineEdit**, который «умеет» принимать пользовательский ввод и позволяет редактировать текст, но только одну строку (для многострочных - **QTextEdit**). По умолчанию в поле редактирования можно ввести любую строку. Но для нашего примера нужно **ограничить** пользовательский ввод не только цифрами, но и диапазоном значений. Для поля редактирования можно задать:

* множество разрешенных для ввода символов  
  QLineEdit::**setValidator()**,
* максимальную длину вводимого текста  
  QLineEdit::**setMaxLength()**
* …

Рассмотрим поподробнее метод  
**void** QLineEdit::**setValidator**(constQValidator\*);  
Перед вызовом метода требуется создать объект, производного от **QValidator** класса, в нашем случае – **QIntValidator** (так как задача требует ввода только целых значений). В классе QIntValidator есть удобный конструктор, который принимает допустимый диапазон вводимого значения (что означает: если при вводе очередной цифры результирующее значение выходит за заданный диапазон, эта цифра будет) проигнорирована. В качестве «родителя» логично указать поле редактирования, для которого создается валидатор.

QIntValidator::**QIntValidator** (int*minimum*, int*maximum*, [QObject](qthelp://org.qt-project.qtgui.502/qtcore/qobject.html) \**parent* = 0);

Задайте диапазон и свойства поля редактирования

### Поле вывода – QLabel

Это самый простой элемент управления, который напрямую не предназначен для взаимодействия с пользователем. В большинстве случаев он используется для того, чтобы вывести надпись, которая не будет изменяться на всем протяжении жизни окна. Но в нашем случае (для того, чтобы посмотреть возможности этого элемента управления) поле вывода должно отображать текущее значение тоже.

Для QLabel можно задать:

* В конструкторе QLabel строчку (например «Not set»),
* Рамку (иначе рамки просто не будет)  
  QFrame::**setFrameStyle()**
* размер, так как «без содержимого» ширина QLabel будет нулевой  
  QWidget::**setMinimumWidth()**
* выравнивать текст (по умолчанию текст «сдвинут» влево)  
  QLabel:: **setAlignment()**
* Можно сопоставить картинку  
  QLabel::**setPixmap()**

Картинка пока не нужна, а рамку и минимальный размер установить полезно. Также логично центрировать выводимый текст в поле вывода как по горизонтали, так и по вертикали

## Синхронизация элементов управления (сигналы/слоты)

Во всех элементах управления разными способами должно отображаться **одно и то же значение** =>при изменении значения с помощью любого из редактируемых элементов управления, в остальных элементах управления оно должно тоже изменяться.

Для большинства элементов управления в соответствующих классах Qt предусмотрены сигналы, которые генерируются при воздействии пользователя, и слоты, которые принимают новое значение и отображают его.

|  |  |  |
| --- | --- | --- |
|  | Сигналы | Слоты |
| QSpinBox | void valueChanged (**int**)  void textChanged (const **QString**&)  (void valueChanged (**QString**))  ( начиная с Qt14 – obsolete) | void setValue (**int**) |
| QSlider | void QAbstractSlider::valueChanged (**int**) | void QAbstractSlider::setValue (**int**) |
| QLineEdit | void textChanged (**QString**) | void setText (**QString**) |
| QLabel |  | void setNum (**int**)  void setText (**QString**) |

### Соединяем наборный счетчик с ползунком

В качестве примера обеспечим коррекцию позиции ползунка при изменении значения посредством наборного счетчика:

QObject::connect(spin,SIGNAL(valueChanged(**int**)),slider,SLOT(setValue(**int**)));

И, наоборот, при изменении позиции ползунка посылаем сигнал наборному счетчику.

QObject::connect(slider,SIGNAL(valueChanged(**int**)), spin,SLOT(setValue(**int**)));

Важно: два элемента управления теперь «завязаны» друг на друга. Почему же не происходит «зацикливания» при изменении значения посредством наборного счетчика:

* наборный счетчик генерирует сигнал valueChanged(**int**)
* в ответ на этот сигнал вызывается слот ползунка setValue(**int**)
* В слоте выполняется проверка того, что полученное значение отличается от текущего и если это так, то

а) значение ползунка при этом **изменяется**,

б) ползунок свою очередь тоже генерирует сигнал valueChanged (**int**)

* в ответ вызывается слот наборного счетчика setValue(**int**), но так как значение наборного счетчика соответствует принятому параметру, дальнейшей генерации сигнала не происходит!

Соедините самостоятельно наборный счетчик с полем редактирования и полем вывода. Замечания:

1. следует учесть, что при выполнении слота (при условии, что значение меняется) в свою очередь генерируется сигнал об изменении значения

2. поле вывода (QLabel) не генерирует сигнал при изменении значения!

3. поле редактирования (QLineEdit)генерирует сигнал textChanged(**QString**), который формирует параметр типа QString, а слоты наборного счетчика и ползунка setValue(**int**)принимают в качестве параметра целое значение!

### Пользовательские сигнал/слот

Как связать сигнал QLineEdit::textChanged(**QString**) со слотом наборного счетчика или ползунка setValue(**int**)? Вариантов решения такой задачи несколько =>рассмотрим один из них в учебных целях:

* в классе Widget создадим пользовательский сигнал, например,   
  void mySignal (**int**)
* в качестве «переводчика» из QString в int используем наше главное окно. В классе главного окна создадим слот, который:
  + принимает в качестве параметра **QString**,
  + преобразует его в int
  + эмитирует сигнал, который в качестве параметра принимает int, - mySignal(**int**)
* соединим сигнал QLineEdit::textChanged(**QString**)со слотом Widget:: mySlot(**QString**)
* соединим сигнал Widget::mySignal(**int**)) со слотом QSpinBox:: setValue(**int**)

Проверьте функциональность приложения

Замечание:

В слоте mySlot(**QString**) в качестве параметра получаем строку. Так как для QLineEdit установлен QIntValidator, то он не будет допускать ввода символов, которые не являются цифрами. Но если попытаться ввести значение 55, то валидатор позволит это сделать. Значение 99 также возможно будет ввести. В чем же дело?

## Вертикальная компоновка - QVBoxLayout

В конструкторе Widget мы динамически создали объект главной компоновки панели. Это была горизонтальная компоновка.

1. Так как использовать будем одну панель, а менять будем компоновки на ней, то удобно для создания вертикальной компоновки использовать соответствующий метод, например:

QLayout\* createVBoxLayout();

в котором производить настройку компоновки

1. Добавьте элементы ввода/вывода (spinbox, lineedit, slider, label) в вертикальную компоновку (главную компоновку панели)
2. Возможно потребуется дополнительная настройка.

Предупреждение:

Одновременно создавать в конструкторе Widget несколько разных компоновок на панели не следует, т.к. эти компоновки содержат один набор элементов ввода.

### Центровка

Приложение смотрится красивее, если элементы управления «отцентрированы». Для расположения элементов внутри компоновки используется третий параметр метода addWidget() - Qt::Alignment.

Выравнивание по умолчанию равно 0, что означает, что виджет заполняет всю ячейку

### Настройка вертикальной компоновки

Специфика: по умолчанию менеджер вертикальной компоновки разделяет клиентскую область

окна поровну между всеми компонентами => все компоненты имеют одинаковую ширину (и высоту). Возможно, расположение и размеры элементов управления Вас устроят. Но скорее всего, захочется кнопку, наборный счетчик, а может быть и поле редактирования сделать поменьше.

### Свойство QSizePolicy элементов управления

|  |  |
| --- | --- |
| **QSizePolicy::Fixed** | виджет всегда имеет фиксированные размеры -sizeHint().Не может ни растягиваться, ни сжиматься. |
| **QSizePolicy::Minimum** | виджет может расширяться, но уменьшаться может только до *sizeHint()* |
| **QSizePolicy::Maximum** | виджет может расширяться, но только до *sizeHint()*, может и уменьшаться,  но до *minimumSizeHint()* |
| **QSizePolicy::Preferred** | виджет может и расширяться и уменьшаться, но до *minimumSizeHint()* |
| **QSizePolicy::Expanding** | виджет может и расширяться и уменьшаться, но до *minimumSizeHint(), но*  *предпочитает расширяться* |
| **QSizePolicy::MinimumExpanding** | возможно расширение, может уменьшаться до *sizeHint()* |
| **QSizePolicy::Ignored** | игнорируется рекомендуемый размер виджета (*sizeHint()*) |

Для элементов управления установлены следующие значения QSizePolicy

|  |  |  |
| --- | --- | --- |
|  | Horizontal | Vertical |
| QPushButton | Minimum | Fixed |
| QSlider | Expanding | Fixed |
| QLabel | Prefered | Preferred |
| QLineEdit | Expanding | Fixed |
| QSpinBox | Minimum | Fixed |

Замечание:

Если применить центровку для слайдера, то он перестанет растягиваться при изменении размеров окна, а зато кнопки, находящиеся справа, будут растягиваться.

Graphical user interface, text, application

Description automatically generated

В том случае, если при добавлении слайдера в вертикальную компоновку, мы не будем устанавливать центровку, то для выравнивания будет использоваться значение по-умолчанию (0) и при этом элемент будет занимать все доступное пространство.

Graphical user interface, application, table

Description automatically generated

Заметим, что кнопки правой вертикальной компоновки перестанут растягиваться. Это происходит потому, что у слайдера по умолчанию установлена горизонтальная политика **QSizePolicy::Expanding** т.е. он пытается расширяться и занимать как можно большее место.

Если для кнопок “Disable” и “Fusion style” установить политику Qt::Fixed по горизонтали, то получим иной эффект. Размер кнопок будет установлен из значения свойства sizeHint().

Graphical user interface, application

Description automatically generated

## Компоновка с помощью таблицы - QGridLayout

Выполните создание табличной компоновки аналогично.

Специфика:

* Добавить виджет в такую таблицу можно с помощью метода   
  QGridLayout::**addWidget(),** указывая в качестве параметров индекс строки и столбца.

**void** QGridLayout::**addWidget**([**QWidget**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/Lab3_Qt10_задание/qwidget.html) \**widget*, **int** *row*, **int** *column*, [**Qt::Alignment**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/qtcore/qt.html#AlignmentFlag-enum) *alignment* = ...);

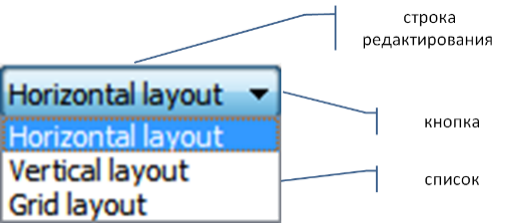
* Перегруженный метод QGridLayout::addWidget() позволяет указать дополнительные параметры в том случае, если хочется, чтобы виджет занимал несколько ячеек таблицы

**void** QGridLayout::**addWidget**([**QWidget**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/Lab3_Qt10_задание/qwidget.html) \**widget*, **int** *fromRow*, **int** *fromColumn*, **int** *rowSpan*, **int** *columnSpan*, [**Qt::Alignment**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/qtcore/qt.html#AlignmentFlag-enum) *alignment* = ...);

## Переключение компоновок

Добавляем в наше приложение возможность изменять вид компоновки по желанию пользователя. В нашем варианте сменными являются компоновки панели. Для переключения компоновок будем использовать элемент управления comboBox.

### Выпадающий список – QComboBox



Это элемент управления с большими функциональными возможностями. В нашей задаче он потребуется для выбора нужного вида компоновки. Выпадающий список содержит строки.

Каждой строке в списке соответствует индекс. Индексы начинаются с 0.

Конструктор: QComboBox (QWidget \**parent* = 0);

При выборе пользователем пункта из comboBox генерируется сигнал **activated**(int)

Для «заполнения» comboBox строками-подсказками можно использовать методы  
addItem():

**void** QComboBox::**addItem**(const [**QString**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/qtcore/qstring.html) &*text*, const [**QVariant**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/qtcore/qvariant.html) &*userData* = QVariant());

**void** QComboBox::**addItem**(const [**QIcon**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/qtgui/qicon.html) &*icon*, const [**QString**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/qtcore/qstring.html) &*text*,

const [**QVariant**](file:///D:/C++/QT/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LAB%203/qtcore/qvariant.html) &*userData* = QVariant());

Подсказки:

1. Добавьте combobox в вертикальную компоновку с кнопками.
2. В классе Widget для «переключения» компоновок предусмотрите соответствующий слот, например:

void **slotChangeLayout**(int index);

1. В этом слоте необходимо будет в зависимости от параметра подключать нужную компоновку к панели. Но прежде, чем на панели установить новую компоновку, необходимо **удалить с панели объект прежней компоновки с помощью delete**. Для того, чтобы узнать, какая на панели установлена компоновка (какую надо удалять), можно воспользоваться методом

**QLayout\* QWidget::layout()**

1. Чтобы при выборе пользователем пункта в combobox происходило «переключение», следует соединить combobox и Widget с помощью соответствующих сигнала и слота.

Замечание

Замена компоновки на панели происходит во время выполнения, поэтому следует учитывать некоторые особенности. Если при создании менеджера компоновки не был указан виджет, то связь между виджетом и компоновщиком можно установить с помощью метода QWidget::setLayout(). После установки нового компоновщика на виджете необходимо вызвать метод QLayout::activate().

Graphical user interface, text, application

Description automatically generated

Graphical user interface, application, table

Description automatically generated Graphical user interface, text, application

Description automatically generated

## Синхронизация отображения данных при запуске приложения

Было бы неплохо синхронизировать отображение данных при старте программы, а именно, отображать одинаковые значения на всех элементах, отображающих данные (метке, слайдере, наборном счетчике, строке редактирования). Для этого можно:

* в конструкторе окна сгенерировать сигнал от любого элемента с заданным значением.
* Или для любого элемента вызвать слот, изменяющий значение. В стандартной реализации слота помимо изменения значения генерируется соответствующий сигнал.

## Установка для элементов управления состояния Disabled

Иногда бывает полезно заблокировать некоторые группы элементов. Это можно сделать « в лоб», т.е. у каждого элемента установить enabled в false с помощью метода void **setEnabled(bool)**. Но так как все элементы у нас включены в объектную иерархию, то достаточно заблокировать «родителя».

1. При нажатии на кнопку с текстом «Disabled»:

* элементы управления, расположенные в сменной компоновке, а также кнопки переключения компоновок делаются недоступными
* а текст на кнопке меняется на текст «Enabled».

При нажатии на кнопку с текстом «Enabled»:

* Элементы становятся доступными
* а текс т на кнопке меняется на текст «Disabled».

Graphical user interface, text, application

Description automatically generated

## Дополнительные задания

## Изменение стиля окна

В лабораторной работе 1 мы изменяли цвет фона главного окна, используя объект класса QPalette. Этот класс используется классом QStyle и его потомками для того, чтобы единообразно отображать окна и элементы управления на текущей платформе. В этом задании мы попробуем настроить элементы окна на иной стиль. Для управления будем использовать кнопку QPushButton и сигналы/слоты.

При нажатии на кнопку с текстом «Fusion style»:

* окно отображается в стиле «Fusion»,
* а текс т на кнопке меняется на текст «Windows style», для того, чтобы можно было вернуться к традиционному отображению, принятому в Windows.

При нажатии на кнопку с текстом «Windows style»:

* окно отображается в стиле «Windows»,
* а текс т на кнопке меняется на текст «Fusion style», для того, чтобы можно было отобразить окно в стиле «Fusion», .

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Подсказки:

qDebug()<<QStyleFactory::keys();

QApplication::setStyle(QStyleFactory::create("Fusion"));

Обратите внимание на размер кнопки, переключающей стиль. Он изменился т.к. размер текста изменился.

## Изменение стиля кнопок

Если хочется как-то изменить внешний вид элементов управления, то можно вместо имеющихся стилей создать свой, унаследовав от класса QStyle. Но это весьма трудоемкая и неочевидная работа. В Qt существует более простой способ создания стиля путем использования таблиц стилей, которые основаны на языке CSS (Cascading Style Sheets,), который используется в HTML.

Каскадный стиль есть не что иное, как **текстовое** описание стиля. Существуют определенные правила для формирования строки, описывающей стиль.

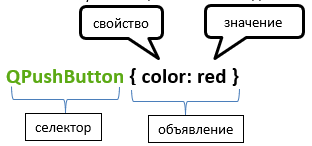
Стиль можно установить в приложении, используя метод **QApplication:: setStyleSheet ()**, или в отдельном виджете при помощи аналогичного метода **QWidget: : setStyleSheet ().** Оба этих метода имеют одинаковую сигнатуру:

void QApplication :: setStyleSheet (const **QString**& sheet); //для всего приложения

void QWidget :: setStyleSheet (const **QString**& sheet); для определенного виджета

Qt Style Sheet терминология и синтаксис полностью соответствуют правилам HTML CSS.

Таблицы стилей состоят из последовательности **правил** стиля. Правило стиля состоит из селектора и объявления. Селектор указывает, на какие виджеты распространяется действие правила; объявление указывает, какие свойства должны быть установлены для виджета.



Для каждого виджета имеется свой список поддерживаемых свойств. Объявление состоит из свойств и значений, а если их несколько, то они отделяются друг от друга точкой с запятой.

Каскадный стиль не чувствителен к регистру, например: COLOR = color = CoLoR и т. д. Исключение составляют имена классов.

Полная информация имеется в документации.

[**QPushButton**](file:///D:/C++/Qt/NEW%20COURSE/BASE_QT%20%20%20%20%20%20DEV-Qt10/LABS/ЗАДАНИЯ%20Qt10/ЗАДАНИЯ/2022/Lab3_Qt10_задание_2022/qpushbutton.html) **{color: green; background-color: yellow}**

Так как описание стиля представляет собой строку, то цвет должен быть сформирован в виде строки. Для формирования цвета можно использовать:

* формат “#RRGGBB”, где R, G, B заданы в 16 системе счисления,
* rgb(r,g,b), где r,g,b- целые значения 0-255
* строка с именем цвета (SVG 1.1)

Table

Description automatically generated

Например, можно установить цвет для строки редактироанияследующая строка устанавливает красный цвет текста для виджета однострочного текстового ввода:

QString stylesheet="QLineEdit {COLOR: red}";

QString stylesheet="QLineEdit {color: #FF0000}";

QString stylesheet="QLineEdit { color: rgb(255, 0, 0)}";

qApp->setStyleSheet(stylesheet);

При этом на всех QLineEdit приложения будет установлен этот стиль.

Обычно описание стиля помещают в текстовый файл, имеющий расширение **qss**.

QFile file(“simple.qss");

QPushButton {

background-color: yellow;

color: green}

}

file.*open*(QFile::ReadOnly);

QString stylesheet = file.readAll();

qApp->setStyleSheet (stylesheet);

**Задание 2:** Задайте стиль для кнопок нашего приложения

## Новые формы метода connect, введенные в Qt5

1. Попробуйте использовать новые формы метода connect, введенные в Qt5.

QMetaObject::Connection  QObject::**connect**(const QObject \**sender*,

**PointerToMemberFunction** *signal***,**

const QObject \**receiver*,

**PointerToMemberFunction***method*) ; [static]

QMetaObject::Connection  QObject::**connect**(const QObject \**sender*,

**PointerToMemberFunction** *signal***,**

**Functor*func*tor**) ; [static]

* Для начала попробуйте связать сигнал textChanged(QString) объекта edit со слотом mySlot(QString) виджета окна.
* Попробуйте выполнить ту же задачу, но в качестве обработчика сигнал используйте не слот mySlot(QString), а функтор.
* Аналогично можно связать сигнал valueChanged(int) объекта slider со слотом setValue(int) объекта spin.
* Потом попробуйте выполнить аналогично «обратную» операцию, т.е. связать сигнал valueChanged(int) объекта spin со слотом void setValue(int) объекта slider.

Скорее всего Вас ожидает неудача (компилятор Вас не поймет), т.к. класс QSpinBox обладает двумя перегруженными сигналами valueChanged(int) и valueChanged(QString). Поэтому необходимо с помощью преобразования уточнить компилятору какой именно сигнал Вы намерены использовать.

Подсказка:

Преобразование можно задать самостоятельно, а можно воспользоваться готовыми средствами, которые предлагает Qt. Для получения справки поищите в QtAssitent информацию о **qOverload**.

## Генерация и обработка пользовательских событий

* Прежде чем генерировать событие, необходимо создать объект-событие. Для этого существует два подхода.
* Для генерации события можно использовать два разных метода.
* Для обработки пользовательского события также можно применять разные методы.

### 1 способ создания объекта события:

* Задать значение для типа события.

Можно задать целочисленное значение от QEvent::User (1000) до QEvent::MaxUser (65535). Но для защиты от случайного дублирования значений лучше получить уникальный идентификатор для пользовательского события с помощью статического метода:

int QEvent::**registerEventType**( int hint = -1 )

* Создать объект типа QЕvent и в конструктор передать значение для своего типа события.

Однако, конструктор QEvent ожидает данное типа QEvent::Type, а в нашем распоряжении значение типа int. Поэтому приходится выполнять явное приведение типа.

### 2 способ создания объекта события:

* Создать **класс** пользовательского события, унаследовав его от класса QEvent. Этот способ в отличие от первого позволяет «запаковать» в событие некоторые данные, которые хочется передать.
* Создать объект этого класса.

### Генерация пользовательского события

Генерацию пользовательского события можно выполнить как с помощью статического метода QCoreApplication::sendEvent(), так и с помощью статического метода QCoreApplication::postEvent() в зависимости от задачи.

bool QCoreApplication::sendEvent(QObject\* receiver, QEvent\* event);

void QCoreApplication::postEvent(QObject\* receiver, QEvent\* event,

int priority = Qt::NormalEventPriority) ;

Замечание: надо помнить, что

1. Вызов sendEvent() не является потоко-безопасным.
2. Существует специфика создания объекта события при использовании разных методов генерации события

### Обработка пользовательского события

1 способ обработки пользовательского события.

Использование перегрузки метода

bool QObject::**event**(QEvent \* e).

Метод event() вызывается прежде, чем специализированный обработчик конкретного события, поэтому перегрузка метода event() позволяет обрабатывать любые события, в том числе пользовательские события.

Замечания:

* Анализ типа события и приведение QEvent\* к нужному типу программист должен сделать сам.
* Важно! Метод event() должен возвращать true, если событие было «принято» и обработано, и false если надо передать его на обработку другим объектам (в иерархии объектов).
* С большой аккуратностью следует перегружать QWidget::event(). Если Вы в своем классе перегружаете метод event(), о не забудьте вызвать метод event() базового класса для того, чтобы была выполнена штатная обработка.

2 способ обработки пользовательского события.

Использование перегрузки метода

bool QObject::**customEvent**(QEvent \* e).

Метод customEvent() вызывается внутри метода QObject::**event**(QEvent \* e) и как раз предназначен для обработки нестандартных событий.

Замечание:

* Анализ типа события и приведение QEvent\* к нужному типу программист должен сделать сам.

**Задание 3.1:**

1. Картинка появляется в произвольном месте клиентской области окна. Координаты позиции картинки генерируются в событии таймера.
2. Стартовать таймер можно любым способом. У меня используется для этого обработчик нажатия на клавишу клавиатуры.
3. Цель –«поймать» картинку, т.е. заключить ее в прямоугольник, нарисованный с помощью мыши.
4. При условии, что картинка находится внутри прямоугольника генерируется пользовательское событие
5. В нашем однопоточном приложении это событие посылается главному окну
6. Обработка события выполняется в главном окне. При этом таймер останавливается и выводится сообщение в диалоговом окне сообщений QMessageBox.

Замечания:

1. Предложенную задачу, конечно, можно решить и без использования пользовательских событий. Здесь этот инструмент используется исключительно в учебных целях.
2. Можно, вообще опустив «игровой момент», по любому событию генерировать пользовательское событие и обрабатывать его.

Graphical user interface, application, Teams

Description automatically generatedGraphical user interface, application

Description automatically generated