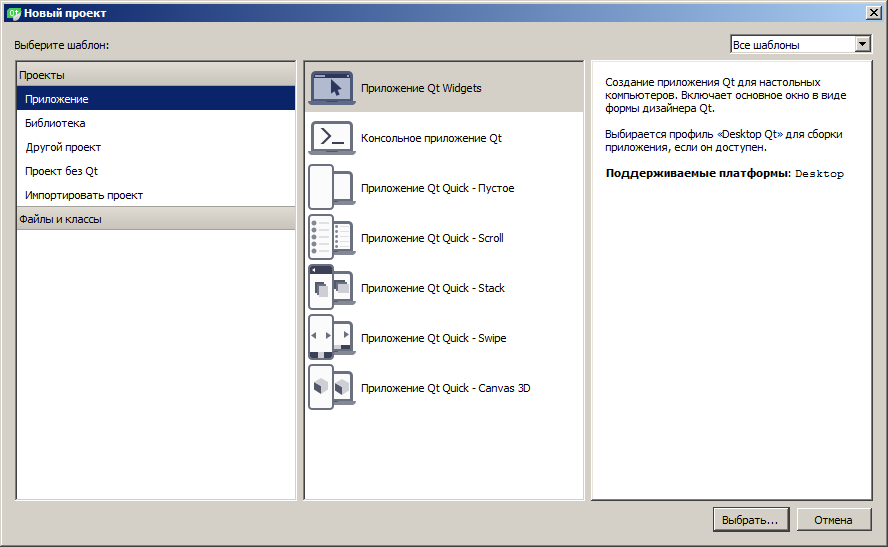
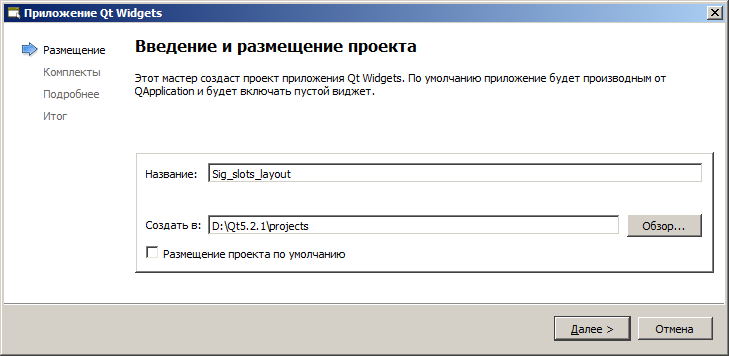
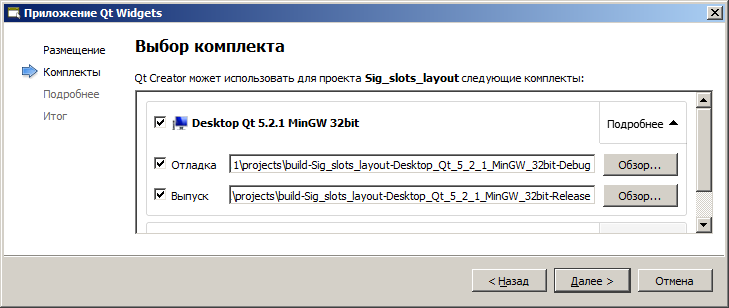
**Практика 3.**

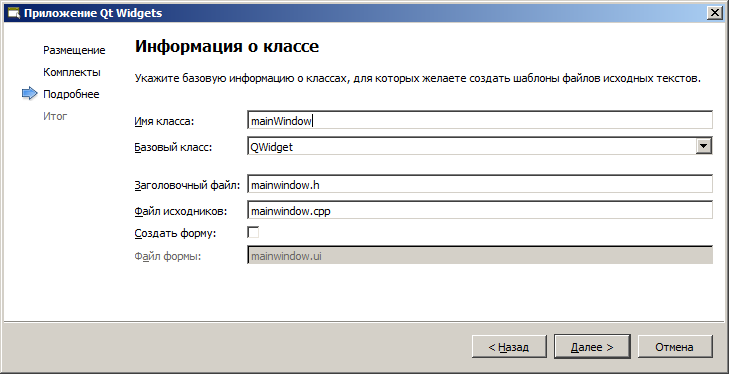
# Простейшие элементы управления (дочерние виджеты). Компоновка виджетов(Layout). Взаимодействие (синхронизация) элементов управления. Сигналы и слоты, предоставляемые классами Qt. Пользовательские сигналы/слоты. Валидаторы. QSignalMapper

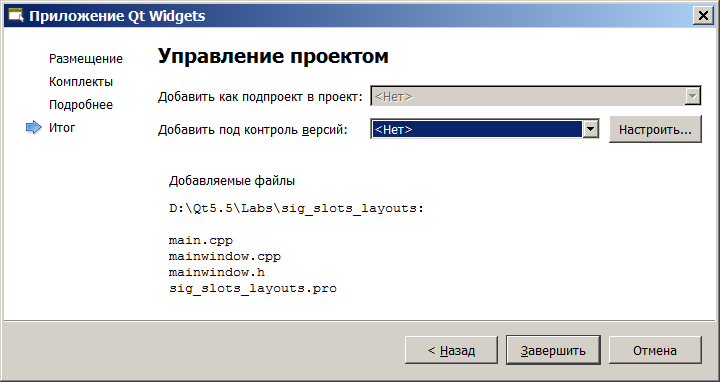
## Заготовка приложения











Можно протестировать заготовку.

## 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:

QObject

QWidget

QAbstractButton

…

…

QAbstractSpinBox

QPushButton

…

QSpinBox

QAbstractSlider

QSlider

QFrame

QLabel

QLineEdit

…

…

…

**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**](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"mainwindow.h"

#include<QPushButton>

mainWindow::mainWindow(QWidget\*parent)

:QWidget(parent)

{

//Кнопка выхода

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

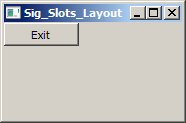
//подумайте, почему можно не сохранять указатель в переменной класса?

//Кто будет освобождать память?

**//соединение создайте самостоятельно;**

}

Должно получится что-то, похожее на:



Протестируйте функциональность приложения.

Обратите внимание на:

* расположение и размеры кнопки (они задаются по умолчанию)
* размеры главного окна (выдается предупреждение «Unable to set geometry…» и размер окна «подстраивается» к размеру кнопки)

## 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 объединенных им виджетов:

* 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:

QLayout

(абстрактный класс)

QObject

QBoxLayout

QFormLayout

QGridLayout

QStackedLayout

QHBoxLayout

QVBoxLayout

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»).

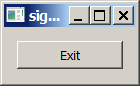
Важно!

* Окно становится владельцем 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()

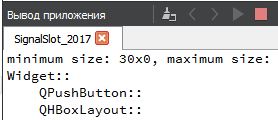
### QHBoxLayout

Проассоциируем с главным окном горизонтальную компоновку. Для начала единственный столбец этой компоновки будет занимать кнопка “Exit”



Последовательность действий:

* В класс mainWindow добавьте указатель   
  QHBoxLayout\* mainLayout;
* В конструкторе mainWindow динамически создайте объект горизонтальной компоновки (QHBoxLayout). При создании объекта можно сразу же задать владельца компоновки (в нашем случае главное окно приложения) посредством параметра конструктора или сделать то же самое позже с помощью метода setLayout()
* Добавьте в компоновку кнопку “Exit” посредством   
  void QBoxLayout::addWidget(QWidget \* widget, int stretch = 0, Qt::Alignment alignment = 0)
* Для того, чтобы отслеживать как формируется иерархия объектов, можно использовать метод класса QObject::dumpObjectTree(). Можно вызвать в конце конструктора этот метод для Вашего окна. В окне «Вывод приложения» можно увидеть результат работы этого метода:



* Это подтверждает, что наше главное окно имеет два дочерних объекта: главную компоновку и кнопку.

### Вложенные компоновки

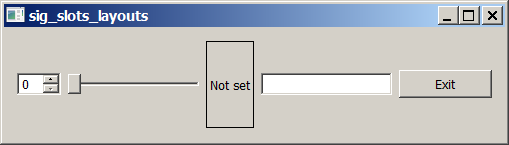
Создаем приложение, которое позволяет пользователю разными способами задавать целое число в требуемом диапазоне (например, 1-99):

* с помощью поля ввода - **QLineEdit**
* с помощью наборного счетчика - **QSpinBox**
* с помощью ползунка–**QSlider**
* Добавим еще поле вывода (**QLabel**) в учебных целях

При этом: каким бы образом пользователь не сформировал значение, все элементы управления должны это значение отображать (то есть все элементы управления должны быть синхронизированы).

В главной компоновке теперь должны быть два столбца:

* первый столбец будет занимать вложенная горизонтальная компоновка (а в ней в свою очередь будут располагаться элементы управления)
* второй – кнопка «Exit»



Последовательность действий:

1. В класс mainWindow добавьте указатели на элементы управления (QLineEdit\* edit; …)
2. В конструкторе mainWindow создайте требуемые элементы управления (классы QLabel, QLineEdit, QSpinBox, QSlider).

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 = Q\_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* = Q\_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* = Q\_NULLPTR);

[**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 = Q\_NULLPTR, Qt::WindowFlags f = Qt::WindowFlags());

[**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* = Q\_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* = Q\_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* = Q\_NULLPTR);

**Очень удобно создавать и настраивать элементы не прямо в конструкторе главного окна, а в отдельных методах, например,таких:**

QLineEdit \***createLineEdit**();

QSlider \* **createSlider**();

QSpinBox \* **createSpinBox**();

QLabel \* **createLabel**();

1. В класс mainWindow добавьте указатель (для вложенной компоновки)  
   QHBoxLayout\* hLayout;
2. В конструкторе mainWindow динамически создайте объект горизонтальной компоновки (QHBoxLayout).
3. Сгруппируйте все виджеты (которые хочется скомпоновать горизонтально) посредством объекта компоновки и метода addWidget()
4. Добавьте вложенную компоновку в главную компоновку окна

**Замечания :**

1. **Напоминем**: дочерние окна требуется создавать **динамическ**и, чтобы при разрушении родителя уничтожение дочерних окон происходило тоже автоматически
2. **(существенное замечание):**

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

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

Пример кода:

#include<QPushButton>

**#include<QSpinBox>**

**…**

**#include <QHBoxLayout>**

mainWindow::mainWindow(QWidget\*parent)

:QWidget(parent)

{

//Кнопка выхода

QPushButton\*buttonExit=new…

connect(…);

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Создание и настройка элементов управления

**Edit = new QLineEdit(this); //или без указания родителя**

//Edit = createLineEdit();

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

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**//Вложенная горизонтальная компоновка**

hLayout**=new QHBoxLayout();//без владельца!!!**

**//Добавить элементы управления в компоновку**

hLayout**->addWidget(edit);**

**остальные виджеты добавьте в компоновку самостоятельно**

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

//Формирование главной компоновки

mainLayout=new QHBoxLayout(this);

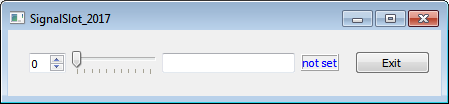
mainLayout->addLayout(hLayout);

mainLayout->addWidget(buttonExit);

}

Мы получили «нулевое» приближение компоновки.

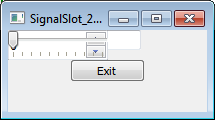
Запускаем приложение и любуемся на достигнутый результат.



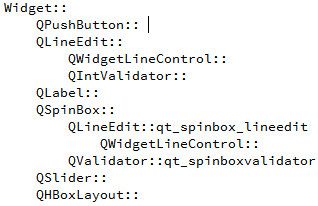
Замечание (о создании элементов управления и компоновке):

Если при создании элементов управления в качестве родителяВыуказали **главное окно** приложения, то для элементов управления по умолчанию устанавливается позиция (0,0) в координатах рабочей области окна.

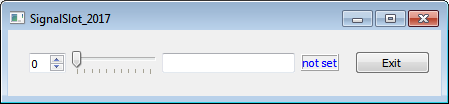
При запуске приложения (до того, как виджеты были добавлены в компоновку) Вы увидите приблизительно следующее:



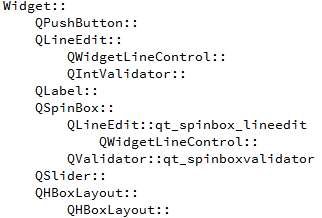
При этом метод QObject::dumpObjectTree() выдаст следующую информацию об объектной иерархии:



После того, как виджеты будут добавлены в компоновку, а сама компоновка добавлена в главную компоновку, картина изменится:

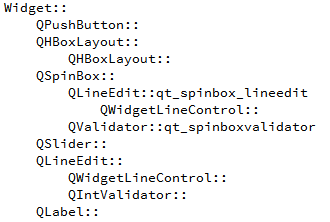


При этом метод QObject::dumpObjectTree() выдаст следующую информацию об объектной иерархии:



**НО!** Оказывается, что необязательно указывать родителя при создании элемента управления, если используются менеджеры компоновки. 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. 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)

Задайте диапазон наборного счетчика и при желании – начальное значение

Наборный счетчик при изменении значения генерирует сигналы:

* void [**valueChanged**](file:///D:\Marina\Qt\Labs\qspinbox.html#valueChanged) ( int *i* )
* void [**valueChanged**](file:///D:\Marina\Qt\Labs\qspinbox.html#valueChanged) (constQString& *text* )

### Ползунок –QSlider

Для ползунка можно задать:

* горизонтальный/вертикальный - Qt::Horizontal или Qt::Vertical вконструктореQSlider  
  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 void QAbstractSlider::**setOrientation** (Qt::Orientation)
* диапазон значений, вкоторые 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()
* …

Задайте диапазон внешний вид и свойства ползунка

### Поле редактирования - QLineEdit

Qt предоставляет несколько классов для приема пользовательского текста. Самым простым средством является элемент управленияQLineEdit, 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 «Notset»),
* 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  
  setPixmap()
* QLabel:: setFrameStyle() – рамка
* …

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

В результате Ваших экспериментов должно получиться что-то вроде



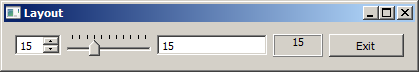
## Синхронизация элементов управления (сигналы/слоты)

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 Qtпредусмотрены сигналы, которые генерируются при воздействии пользователя, и слоты, которые принимают новое значение и отображают его.

|  |  |  |
| --- | --- | --- |
|  | Сигналы | Слоты |
| QSpinBox | void valueChanged(int)  void valueChanged(QString) | 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) |

В результате должно получиться что-то вроде:



### 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:

* он генерирует сигнал valueChanged(int)
* в ответ на этот сигнал вызывается слот ползунка setValue(int)
* так как значение ползунка при этом изменяется, он в 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:

LineEdit

сигнал - textChanged(QString)

Slider

слот - setValue(int)

mainWindow

void mainWindow::mySlot(QString s){

int n;

//перевод s в целое

emit (mySignal(n));

}

mySignal(int)

* в классе mainWindow 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 mainWindow:: mySlot(QString)
* соединим сигнал mainWindow::mySignal(int)) со слотом QSpinBox:: setValue(int)

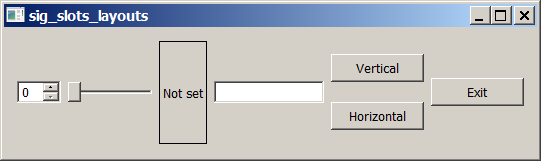
Проверьте функциональность приложения

# Переключение компоновок

Добавляем в наше приложение возможность изменять вид компоновки по желанию пользователя (например, при нажатии соответствующей кнопки).

Замечание: кнопки можно тоже сгруппировать посредством компоновки.

Например, имеем горизонтальную компоновку:



Самостоятельно:

* добавьте кнопки «Vertical» и «Horizontal»
* и объедините их вложенной вертикальной компоновкой QVBoxLayout \* buttonsLayout
* добавьте эту вертикальную компоновку в главную компоновку окна с помощью метода addLayout().

Замечание: добавление производим ДО добавления buttonExit.

//Формирование главной компоновки

mainLayout=new QHBoxLayout(this);

mainLayout->addLayout(hLayout);

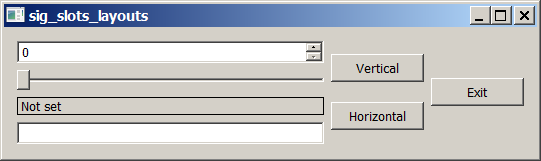
mainLayout->addLayout(buttonsLayout);

mainLayout->addWidget(buttonExit);

При нажатии на соответствующую кнопку, компоновка элементов управления должна изменяться.

# Вертикальная компоновка - QVBoxLayout

Вертикальная компоновка:



Вертикальную и горизонтальную компоновки поддерживают объекты **разных** классов: QHBoxLayout и QVBoxLayout =>нужно:

* создать и подготовить к использованию **все** возможные вложенные компоновки, предназначенные для размещения элементов управления
* проассоциировать с главной компоновкой только одну компоновку - по умолчанию

Последовательность действий:

* добавить в класс mainWindow переменную   
  QVBoxLayout\*vLayout;
* в конструкторе mainWindow создать объект вертикальной компоновки и посредством addWidget() добавить в компоновку уже существующие элементы управления
* Важно! добавление элементов управления в вертикальную компоновку необходимо выполнить **до** вызова mainLayout->addLayout(hLayout);  
  то есть:

Создание элементов управления

Создание объекта горизонтальной компоновки

Заполнение горизонтальной компоновки элементами управления-AddWidget()

Создание объекта вертикальной компоновки

Заполнение вертикальной компоновки элементами управления-AddWidget()

Формирование главной компоновки

mainLayout->addLayout(hLayout);

mainLayout->addLayout(vLayout);

# QSignalMapper

### Переключаем компоновки

Замену текущей компоновки на новую можно выполнить в общем виде посредством слота mainWindow, если в качестве параметра слот будет получать **указатель** на новую компоновку =>унифицируем переключение посредством QSignalMapper.

Объект класса QSignalMapper будет служить «ретранслятором» сигнала clicked(bool) (который в нашем случае будет генерироваться при нажатии кнопок «Vertical» и «Horizontal»)и получателем (слотом mainWindow)

QSignalMapper

Horizontal

Vertical

void slotChange(QObject\*newLayout)

{…}

mainWindow

clicked(bool)

clicked(bool)

clicked(bool)

clicked(bool)

mapped(QObject\*)

mapped(QObject\*)

Последовательность действий:

* В классе mainWindow реализовать слот void ChangeLayout(QObject\*pNewLayout);  
  В слоте:
  + получить указатель на текущую вложенную компоновку посредством   
    QLayoutItem \* QLayout::itemAt(int index) const главной компоновки
  + исключить текущую вложенную компоновку из главной компоновки посредством   
    void QLayout::removeItem(QLayoutItem \* item)
  + вставить новую вложенную компоновку в главную компоновку посредством   
    void QBoxLayout::insertLayout(int index, QLayout \* layout, int stretch = 0)

void mainWindow::ChangeLayout(QObject \*pNewLayout)

{

//получить указатель на текущую вложенную компоновку

//0-индекс вложенной компоновки в mainLayout.

QLayoutItem \* pItem= mainLayout->*itemAt*(0);

//исключить текущую вложенную компоновку из главной компоновки

mainLayout->removeItem(pItem);

//вставить новую вложенную компоновку в главную компоновку

mainLayout->insertLayout(0,static\_cast<QLayout \*>(pNewLayout));

}

* в классе mainWindow создать переменную - QSignalMapper\*mapper;
* в конструкторе mainWindow создать объект QSignalMapper.   
  Замечание: при создании удобно «родителем» сделать главное окно
* соединить (connect())сигнал каждой кнопки clicked(bool) со слотом объекта mapper - QSignalMapper::map()
* Так как слот ChangeLayout должен получать в качестве параметра **указатель** на новую компоновку =>после создания объекта mapper требуется сопоставить каждому отправителю отправляемое значение посредством   
  void QSignalMapper::setMapping(QObject \* sender, QObject \* object)  
  где sender–указатель на нажатую кнопку, а object – в нашем случае указатель на требуемую компоновку
* осталось соединить сигнал QSignalMapper::mapped(QObject\*)со слотом   
  mainWindow:: ChangeLayout(QObject\*)

### 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.

### Фиксация размера виджетов в компоновке - QSizePolicy

Варианты задания размера виджетов в layout-е можно сформировать с помощью свойства   
QSizePolicyQWidget::sizePolicy. При создании виджетов работают следующие умолчания:

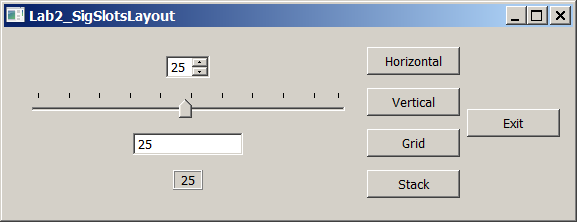
* Таким элементам управления, как QButton, QLineEdit, QSpinBox позволено изменять горизонтальный размер, а высота у них фиксирована
* Вертикальным слайдерам позволено изменять только высоту, горизонтальным – только ширину (это справедливо для всех полос прокрутки)
* Для остальных (большинства) виджетов - Preferred/Preferred, что означает, что оба размера виджета могут изменяться.
* …

Если умолчания не устраивают, то можно задать политику изменения размеров с помощью:  
QWidget::setSizePolicy()

### Центровка

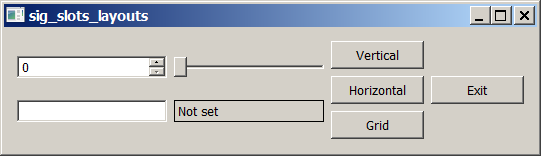
Приложение смотрится красивее, если элементы управления «отцентрированы». Для расположения элементов внутри компоновки используется третий параметр метода addWidget() - Qt::Alignment.

Расположите элементы управления примерно так:



## Компоновка с помощью таблицы - QGridLayout

Самостоятельно подключите табличное представление элементов управления. Таблица состоит из строк и столбцов:

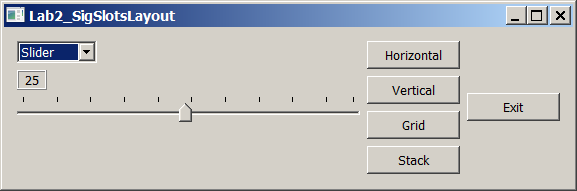


Специфика:

* Добавить виджет в такую таблицу можно с помощью метода   
  QGridLayout::addWidget(), указывая в качестве параметров индекс строки и столбца.
* Перегруженный метод QGridLayout::addWidget() позволяет указать дополнительные параметры в том случае, если хочется, чтобы виджет занимал несколько ячеек таблицы
* …

## Стековая компоновка - QStackedLayout

Стековая компоновка используется для таких задач, в которых требуется, чтобы в каждый момент времени видимым был только один из виджетов, включенных в такую компоновку (это альтернатива «закладкам»).



Для демонстрации работы стековой компоновки в главную компоновку вставляем вертикальную компоновку (синий пунктир) - demoStackLayout.

В этой вложенной вертикальной компоновке располагаем:

* combobox для переключения элементов управления в стековой компоновке,
* label и
* собственно стековую компоновку (зеленый пунктир), которая будет содержать счетчик (spin), слайдер (slider) и строку редактирования (edit).

Подсказки:

1. Для «заполнения» combobox строками-подсказками пользуйтесь методом  
   addItem().
2. При выборе пользователем пункта из combobox генерируется сигнал **activated**(int)
3. В классе QStackedLayout для «переключения» компонент (это могут быть виджеты, а могут быть в свою очередь тоже компоновки) предусмотрен слот - **setCurrentIndex**(int)
4. Чтобы при выборе пользователем пункта в combobox в стековой компоновке происходило «переключение», эти два объекта следует соединить с помощью соответствующих сигнала и слота.
5. При создании горизонтальной компоновки, в которую входит combobox, высота компоновки определяется размером combobox с учетом выпадающего списка. => Это стоит учесть при формировании горизонтальной компоновки и «прижать» combobox и label к верхнему краю посредством третьего параметра метода QHBoxLayout::addWidget(…,…,Qt::Alignment);

Последовательность действий:

* В класс mainWindow добавить указатель  
  QStackedLayout\*stackLayout;  
  и в конструкторе mainWindow создать соответствующий объект
* Заполнить stackLayout элементами управления ( )
* В класс mainWindow добавить указатель QComboBox\* combo;
* В конструкторе mainWindow создать объект QComboBox и посредством QComboBox::addItem() заполнить comboBox строками-подсказками (какой элемент должен быть видимым). Порядок строк должен соответствовать порядку следования элементов в stackLayout.
* В класс mainWindow добавить указатель  
  QVBoxLayout\*demoStackLayout;   
  и в конструкторе mainWindow создать соответствующий объект
* Важно! Необходимо сделать combobox невидимым – QWidget::hide(), так как этого элемента не в других компоновках
* Необходимо соединить сигнал QComboBox::activated( int) со слотом стековой компоновки QStackedLayout::(setCurrentIndex(int).
* Важно! для стековой компоновки нужно установить режим – отображать все виджеты, иначе мы не увидим их в других компоновках  
  stackLayout->setStackingMode(QStackedLayout::StackAll);

Придется также «поставить заплатку» в слоте ChangeLayout():

void mainWindow::ChangeLayout(QObject\*pNewLayout)

{

if(pNewLayout==demoStackLayout)

{

combo->*setVisible*(true);

//скрываем все элементы управления

spin->hide();

edit->hide();

slider->hide();

//отображаем только текущий

stackLayout->currentWidget()->show();

stackLayout->setStackingMode(QStackedLayout::StackOne);

}

else

{

combo->*setVisible*(false);

stackLayout->setStackingMode(QStackedLayout::StackAll);

}

}

## Очистка ресурсов при завершении приложения

Если компоновка «отключена» от виджета, то при разрушении виджета программист сам должен позаботиться об уничтожении динамически созданных компоновок. Следовательно, в деструкторе mainWindow можно отключить текущую компоновку от виджета и для всех динамически созданных компоновок вызвать delete.

## «Пружинка»

Если требуется «подпереть» элемент, чтобы при увеличении размера окна виджет не увеличивался пропорционально растяжению, можно в компоновку добавить «пружинку» (объект QSpacerItem).

* Создаем объект QSpacerItem
* Добавляем его в компоновку с помощью метода addItem()