# Programacion QT4 facil

Luis Tomas Wayar

October 31, 2010

A mi esposa Elisa, mi compañera de toda la vida.

# Sobre este documento

## Motivación

Para aprender a programar en C++ usando las librerías QT recurrí a varios libros que me resultaron de gran utilidad, lamentablemente todos ellos en ingles, todos muy completos y abundantes en explicaciones lo que resulta un poco tedioso de leer cuando uno lo que quiere es obtener los conocimientos y los resultados rápido. Por eso he decidido escribir este documento, donde voy a intentar escribirlo pensando en como me gustaría a mi encontrar un documento que me introduzca en un tema de programación en particular, yendo a los temas directamente y poniéndolos en practica en un ejemplo sin entrar en muchos detalles que confundan y dilaten el proceso de aprendizaje. Es raro encontrar documentación donde se muestre y explique la metodología para el uso, por eso este documento va a proveer una guía incremental donde en ejemplos simples y sencillos vamos a ver como usar esta fantástica librería de programación de interfaces gráficas de usuario.

Como es sabido la mejor forma de aprender algo es haciéndolo, por eso, luego de explicado el ejemplo para cada tema, se proponen al final de cada capitulo unos ejercicios para que el lector los desarrolle y adquiera las destrezas necesarias para aplicar lo aprendido.

Mi idea es leer todo lo que encuentre sobre cada tema, mas la experiencia propia y escribir lo mas simple y conciso posible.

## Audiencia

Este documento esta destinado a todos aquellos que quieran aprender a usar y sacar el máximo provecho a la programación de interfaces gráficas de usuario usando el lenguaje de programación C++ y las librerías QT4. Se requieren conocimientos previos de programación C++ y de la terminología y conceptos propios de la programación de GUIs.

Se usa como plataforma de desarrollo GNU/Linux pero también se pueden aplicar a otras plataformas con los cambios mínimos propios de cada sistema operativo.

## Herramientas

Para la realización de las practicas se usara un editor o ide de programador y el correspondiente compilador de C++, se recomienda como editores VIM o Kdevelop que son mis favoritos, sin embargo se puede optar por otros editores o ides según sea su elección. Para la compilación y pruebas se necesita acceso a un emulador de terminal.

# Bibliografía

[1] C++ GUI Programming with Qt 4, Second Edition

[2] The Art of Building Qt Applications

[3] Introduction to Design Patterns in C++ with Qt 4

# Índice general

# Índice de figuras

# Parte I

# Introducción a QT

# Capítulo 1

# Comenzando

> "Hay sólo dos clases de lenguajes de programación: aquellos de los que la gente está siempre quejándose y aquellos que nadie usa."
>
> — Bjarne Stroustrup

En este capitulo vamos aprender los siguientes temas:

- Nuestro primer programa
- Como crear un proyecto
- Como organizar los widgets en la ventana

## 1.1. Algunas convenciones iniciales

Vamos a ponernos de acuerdo en algunas convenciones básicas sobre como escribir

- Los nombres de clase comienza con una letra mayúscula
- Los nombres de las funciones comienzan con una letra minúscula.
- Aunque son permitidos por el compilador los puntos, guiones, subrayado y otros caracteres especiales deben evitarse siempre que sea posible (excepto donde se indica a continuación).
- En los nombres compuestos por varias palabras se escriben la iniciales con mayúsculas.
- Las constantes se escriben con la inicial en mayúscula, constantes globales y macros todo con mayúsculas.
- Cada nombre de clase debe ser un sustantivo.
- Cada nombre de la función debe ser un verbo.
- Cada nombre de variable bool debe producir una aproximación razonable a su función.

## 1.2. Nuestro primer programa

### 1.2.1. Hola mundo

Vamos a comenzar haciendo un primer programa, se trata como es de esperar del clásico "Hola mundo", esta vez pensado para desplegarse en una simple ventana sin ningún tipo de control. Cree un directorio llamado ejmeplo1 y dentro de el escriba un archivo con el siguiente código, nombrelo ejemplo1.cpp

```
1  #include <QApplication>
2  #include <QLabel>
3  int main(int argc, char *argv[])
4  {
5      QApplication app(argc, argv);
6      QLabel *label = new QLabel("Hola Mundo");
7      label->show();
8      return app.exec();
9  }
```

### 1.2.2. Crear el proyecto

Ahora vamos crear el proyecto para nuestro primer ejemplo, para ello vamos a utilizar la herramienta qmake[1], se trata de una herramienta desarrollada por los creadores de las librerías QT[2] para la simplificación del proceso de creación de proyectos, básicamente su funcionalidad consiste en la automatización de la creación de los archivos Makefile para la compilación.

✍La empresa Nokia compro a la empresa Trolltech creadora de la librería QT

Desde una terminal ejecute el siguiente comando para crear un nuevo proyecto basado en nuestro código fuente.

```
$ qmake -project
```

Observe que ahora tiene un nuevo archivo llamado ejemplo1.pro, este es su archivo de configuración del nuevo proyecto.

Ahora vamos a crear nuestro archivo Makefile a partir de ejemplo1.pro, para ello ejecute el siguiente comando en una terminal.

```
$ qmake ejemplo1.pro
```

### 1.2.3. Compilar y ejecutar el programa

Tenemos ahora un nuevo archivo en nuestro directorio llamado Makefile, este es el que se encargara de automatizar la compilación y enlazado de nuestro código fuente, para ello ejecutamos el siguiente comando en una terminal.

```
$ make
```

Tenemos ya un archivo ejecutable al que podemos invocar simplemente llamándolo desde linea de comando.

```
$ ./ejemplo1
```

Con lo que podremos ver en la Figura 1.1 en la página siguiente

### 1.2.4. Entendiendo el programa

Analicemos este breve programa para comprender la funcionalidad de cada una de sus partes.

En esta linea (1) incluimos la clase QApplication. QApplication es la clase

✍Cada clase existente en QT tiene un include con el mismo nombre que la clase

[1]Podemos encontrar el manual de qmake en http://doc.trolltech.com/4.5/qmake-manual.html

[2]La pagina oficial de la librería QT es http://qt.nokia.com/

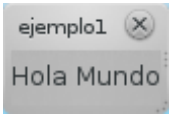


Figura 1.1: Ventana de ejemplo1

que contiene a toda aplicación que use esta librería, recibe argc que contiene el número de argumentos recibidos por el programa, debemos considerar que siempre será el número de argumentos pasados más 1, ya que el primer argumento se reserva para contener el nombre del programa. El segundo es un puntero a un array de chars que contiene los parámetros pasados en el mismo orden en que fueron escritos.

```
1 #include <QApplication>
```

En la linea 2 incluimos la clase QLabel la que sirve para mostrar un texto o una imagen, no provee ningún tipo de interacción con el usuario.

```
1 #include <QLabel>
```

En la linea 6 creamos un objeto de la clase QApplication llamado app que recibe como argumentos los provistos desde linea de comandos, en este caso no vamos a proveerle ninguno cuando lo ejecutemos.

```
1 QApplication app(argc, argv);
```

En la linea 7 creamos un puntero de tipo QLabel llamado etiqueta que apunta a un nuevo objeto que recibe como parámetro una cadena “Hola mundo”.

```
1 QLabel *label = new QLabel("Hola Mundo");
```

En la linea 8 llamamos al método show del objeto label que tiene como función hacer visible el objeto, por defecto los widgets al ser instanciados no se despliegan en pantalla.

```
1  label->show();
```

Finalmente en la linea 9 llamamos a nuestro objeto principal (app) el cual devolverá el valor de retorno del programa.

```
1  return app.exec();
```

## 1.3. Disposición de widgets

### 1.3.1. Planificando la apariencia

La librería QT cuenta con una serie de contenedores que permiten ordenar la disposición de los widgets en nuestras ventanas, se conocen como Layouts, y todos derivan de una clase padre llamada QLayout, mediante estos elementos es posible controla la geometría de los elementos gráficos a ser desplegados. Para entenderlo mejor pensemos en ellos como si se tratara de celdas donde meter los widgets o incluso otros Layouts.

✍Los widgets son los elementos que conforman las interfaces gráficas de usuarios

Según nuestras necesidades disponemos de los siguientes, QBoxLayout, QGridLayout, QFormLayout, y QStackedLayout. A su vez estos heredan en nuevas clases que agregan funcionalidades o características a los manejadores de disposición, por ejemplo, QBoxLayout tiene dos clases heredadas, QVBoxLayout que organiza la disposición verticalmente y QHBoxLayout que lo hace de manera horizontal. En el ejemplo siguiente aplicaremos ambos para demostrar y comprender su funcionamiento.

En un nuevo archivo llamado ejemplo2.cpp escriba el siguiente código.

```
1  #include <QApplication>
2  #include <QVBoxLayout>
3  #include <QLabel>
4  int main(int argc, char *argv[]) {
5      QApplication app(argc, argv);
6      QWidget ventana;
7      QVBoxLayout* principalLayout = new QVBoxLayout(&ventana);
8      QLabel* label1 = new QLabel("Hola");
9      QLabel* label2 = new QLabel("Mundo");
10     principalLayout->addWidget(label1);
11     principalLayout->addWidget(label2);
12     ventana.show();
13     return app.exec();
14 }
```

Siga los pasos descriptos en el ejemplo1 para crear el proyecto y compilar el programa. Como resultado de la ejecución del programa obtendremos la ventan de la Figura 1.2

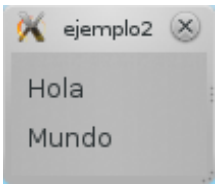


Figura 1.2: Ventana de ejemplo2

### 1.3.2. Entendiendo el programa

Analicemos ahora el código. Como pueden ver se agrego una nueva inclusión, esta incorpora la clase QVBoxLayout, es un contenedor que ordena sus elementos contenidos de manera vertical, uno debajo de otro.

```
1 #include <QVBoxLayout>
```

Luego, para utilizarlo, instanciamos un nuevo objeto llamado principalLayout, observe que recibe como argumento el objeto padre, de esta manera se convierte en el contenedor de nivel superior del widget ventana. Otra cosa importante a tener en cuenta en esta linea es la forma en que instanciamos, teniendo en cuenta que C++ no provee mecanismos de manejo automático de memoria es el programador quien tiene que tomar los recaudos necesarios para gestionar inteligentemente la misma, esa es la razón por la cual creamos el objeto mediante "new" ya que de esta manera nos referimos al objeto mediante un puntero que lo ubica en el "heap", de no hacerlo así el objeto y todos sus elementos derivados se crearían en el "stack" y permanecerían en memoria durante todo el tiempo de vida del proceso, haciéndolo con "new" una vez que se llama a su destructor se libera del "heap" la memoria utilizada.

✍El Stack o Pila es el área de memoria estática donde se almacenan por ejemplo las variables locales

✍El Heap o Montón es una área de memoria dinámica que se usa por ejemplo para ubicar objetos creados con new

```
1 udaQVBoxLayout* principalLayout = new QVBoxLayout(&ventana);
```

La manera incorrecta de instanciarlo seria.

```
1 QVBoxLayout principalLayout(&ventana);
```

De igual manera instanciamos los dos objetos QLabel, recurriendo a punteros.

```
1  QLabel* label1 = new QLabel("Hola");
2  QLabel* label2 = new QLabel("Mundo");
```

Ahora debemos embeber los dos nuevos QLabel (label1 y label2) dentro del contenedor QVBoxLayout (principalLayout), para ello usamos el método addWidget de la clase QVBoxLayout que tiene como función insertar el widget al final del contenedor.

```
1  principalLayout->addWidget(label1);
2  principalLayout->addWidget(label2);
```

En este ejemplo se muestra la forma básica en que se disponen los elementos dentro de una ventana, lo importante a aprender es la forma en que se organiza la disposición de elementos, en resumen se trata de usar contenedores (Layouts) para insertar elementos (widgets) dentro de sus celdas.

## 1.4. Disposiciones mas complejas

### 1.4.1. Anidando Layouts

En el ejemplo anterior vimos como usar los Layouts para disponer de manera ordenada los widgets dentro de la ventana, este método nos ofrece un sinfín de opciones al momento de diseñar nuestras interfaces de usuario, es importante considerara que se pueden anidar y tienen varios atributos y métodos que permiten controlar su posicionamiento y comportamiento.

En el siguiente ejemplo se introducen varios nuevos widgets, QLineEdit, QRadioButton, QCheckBox, QTextEdit, QPushButton, es fácil deducir la función y el funcionamiento de cada uno de ellos, no hace falta entrar en explicaciones al respecto. También introducimos en este ejemplo el nuevo Layout QHBoxLayout cuyo comportamiento es similar al QVBoxLayout con la sola diferencia que dispone de los widgets de manera horizontal.

Veamos el ejemplo.

```
1  #include <QtGui/QApplication>
2  #include <QtGui/QCheckBox>
3  #include <QtGui/QFrame>
4  #include <QtGui/QHBoxLayout>
5  #include <QtGui/QLabel>
6  #include <QtGui/QLineEdit>
7  #include <QtGui/QPushButton>
8  #include <QtGui/QRadioButton>
9  #include <QtGui/QSpacerItem>
10 #include <QtGui/QTextEdit>
11 #include <QtGui/QVBoxLayout>
12
```

```
13  int main(int argc, char *argv[])
14  {
15      QApplication app(argc, argv);
16  // Creamos el widget principal
17      QWidget ventana;
18  // Creamos el Layout principal que sera de tipo Horizontal
19      QHBoxLayout* principalLayout = new QHBoxLayout(&ventana);
20
21  // Creamos dos layouts verticales para ponerlos en el principal
22      QVBoxLayout* izquierdoVLayout = new QVBoxLayout();
23      QVBoxLayout* derechoVLayout = new QVBoxLayout();
24
25  // Creamos un layouts que se agregara a layout izquierdoVLayout
26      QVBoxLayout* izqsupVLayout = new QVBoxLayout();
27
28  // Creamos los layouts que se agregaran en izqsupVLayout
29      QHBoxLayout* ltHLayout = new QHBoxLayout();
30      QLabel* label = new QLabel("Etiqueta");
31      QLineEdit* lineEdit = new QLineEdit("Ingrese un texto");
32      ltHLayout->addWidget(label);
33      ltHLayout->addWidget(lineEdit);
34
35      QHBoxLayout* rbHLayout = new QHBoxLayout();
36      QRadioButton* radioButton1 = new QRadioButton("RB1");
37      QRadioButton* radioButton2 = new QRadioButton("RB2");
38      rbHLayout->addWidget(radioButton1);
39      rbHLayout->addWidget(radioButton2);
40
41      QHBoxLayout* cbHLayout = new QHBoxLayout();
42      QCheckBox* checkBox1 = new QCheckBox("CB1");
43      QCheckBox* checkBox2 = new QCheckBox("CB2");
44      cbHLayout->addWidget(checkBox1);
45      cbHLayout->addWidget(checkBox2);
46
47  // Agregamos los Layouts creados a izqsupVLayout
48      izqsupVLayout->addLayout(ltHLayout);
49      izqsupVLayout->addLayout(rbHLayout);
50      izqsupVLayout->addLayout(cbHLayout);
51
52  // Creamos un nuevo widget de edicion de texto.
53      QTextEdit* textEdit = new QTextEdit("Editor de texto simple");
54
55  // Agregamos el layout izqsupVLayout y el widget de edicion de
        texto a izquierdoLayout
56      izquierdoVLayout->addLayout(izqsupVLayout);
57      izquierdoVLayout->addWidget(textEdit);
58
59  // Creamos un Layout para los botones del tipo QVBoxLayout
60      QVBoxLayout* botonesVLayout = new QVBoxLayout();
61      QPushButton* pushButton1 = new QPushButton("Boton1");
62      QPushButton* pushButton2 = new QPushButton("Boton2");
63      QPushButton* pushButton3 = new QPushButton("Boton2");
64
65  // Agregamos los botones al layout
66      botonesVLayout->addWidget(pushButton1);
67      botonesVLayout->addWidget(pushButton2);
68      botonesVLayout->addWidget(pushButton3);
69
70  // Creamos un Spacer para mejor disposicion
71      QSpacerItem* verticalSpacer = new QSpacerItem(20, 188,
            QSizePolicy::Minimum, QSizePolicy::Expanding);
72
73  // Agregamos a derechoVLayout el layout de botones y el spacer
74      derechoVLayout->addLayout(botonesVLayout);
75      derechoVLayout->addItem(verticalSpacer);
76
77  //Finalmente agregamos al layout principal los layouts izquierdo y
        derecho
```

```
78        principalLayout->addLayout(izquierdoVLayout);
79        principalLayout->addLayout(derechoVLayout);
80
81        ventana.show();
82        return app.exec();
83    }
```

El diagrama que usamos corresponde a una ventana que contiene varios widgets organizados a su vez en varios layouts, en la Figura 1.3 se puede ver el modelo de diagrama.

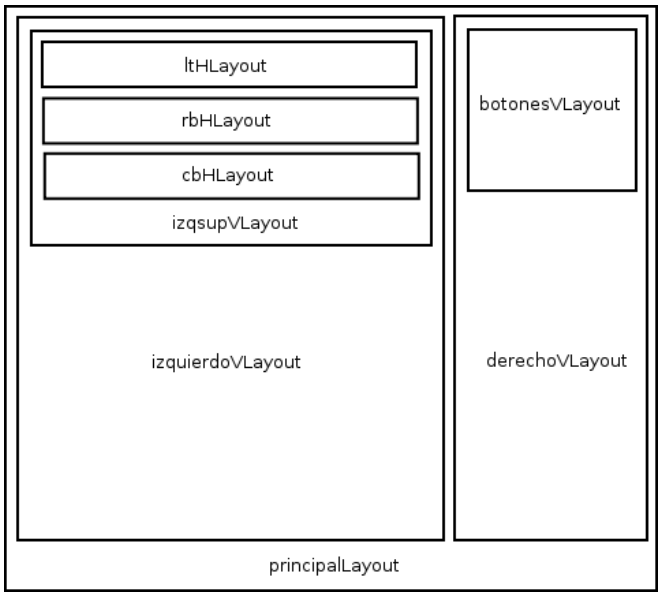


Figura 1.3: Diagrama de ejemplo 3

Una vez compilado el ejemplo siguiendo el mismo procedimiento de los ejemplos anteriores obtenemos la ventana de la Figura 1.4

### 1.4.2. Entendiendo el código

En este nuevo ejemplo incorporamos nuevos layouts y los organizamos anidándolos, podemos ver que además de agregar widgets a un layout como lo hicimos en el ejemplo anterior también podemos agregarles nuevos layouts, en las lineas de abajo vemos una combinación de ambos que se agregan a izquierdoVLayout tanto un layout "izqsupVLayout" como un widget "textEdit", a su vez izqsupVLayout es un layout que anida a otros tres layouts horizontales.

```
1   izquierdoVLayout->addLayout(izqsupVLayout);
2   izquierdoVLayout->addWidget(textEdit);
```

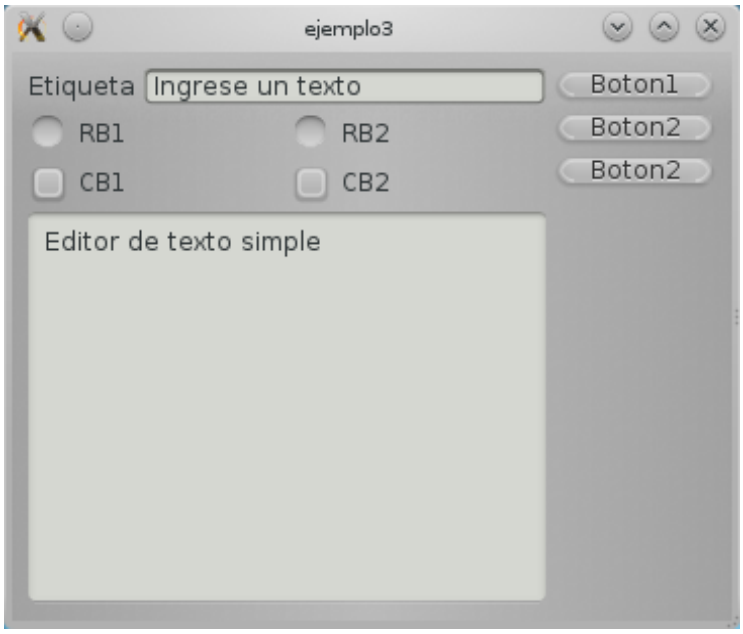


Figura 1.4: Ventana de ejemplo3

## 1.5. Conectando eventos con acciones

### 1.5.1. Señales y slots

Hasta ahora los ejemplos que vimos no tienen ningún tipo de interacción con el usuario, solo despliegan ventanas y en ellas los widgets, en este nuevo ejemplo usaremos un botón el cual sera capaz de responder a un evento.

La mayoría de las librerías de diseño de interfaces de usuario usan callbacks o simplemente entran en un bucle a la escucha de eventos para disparar métodos internos, en el caso de QT la cosa es un poco diferente, entramos acá al concepto de Señal-SLOT, la ventaja principal de esta implementación es que una señal puede conectarse a tantos slots como sea necesario, así un solo evento puede disparar varias acciones de manera mas simple para el programador. Mas adelante veremos con mayo detalle el tema de los slots, por ahora veamos un ejemplo sencillo.

```
1  #include <QApplication>
2  #include <QVBoxLayout>
3  #include <QLabel>
4  #include <QPushButton>
5
6  int main(int argc, char *argv[])
7  {
8      QApplication app(argc, argv);
```

```
9       QWidget ventana;
10      QVBoxLayout* principalLayout = new QVBoxLayout(&ventana);
11      QLabel* label = new QLabel("Hola mundo");
12      QPushButton* boton = new QPushButton("Salir");
13      principalLayout->addWidget(label);
14      principalLayout->addWidget(boton);
15      ventana.show();
16      QObject::connect(boton, SIGNAL(clicked()), &app, SLOT(quit()))
            ;
17      return app.exec();
18  }
```

### 1.5.2. Entendiendo el código

Usamos aquí como base el ejemplo 2, reemplazamos el segundo label por un botón, simplemente lo instanciamos y luego de agregarlo a nuestro layout usamos el método "connect" de la clase QObject. Como se puede observar los dos primeros argumentos que recibe son el objeto que genera la señal y la señal en si misma ante la que tiene que responder, los últimos dos argumentos corresponden al objeto que va a recibir la señal, en este caso nuestra aplicación "app" y el slot.. Esta linea de código lo que hace es enlazar el objeto "boton" con el SLOT "quit()" que actuara sobre el objeto "app" al producirse la señal "clicked()".

Un SLOT es una función normal de una clase especialmente definida para reaccionar ante una señal

```
1   QObject::connect(boton, SIGNAL(clicked()), &app, SLOT(quit()));
```

### 1.5.3. Un poco mas sobre slots y señales

Todos los widgets emiten señales y tienen slots, podemos así conectar la señal emitida de un determinado widget para que actúe sobre otro mediante el llamado a uno de sus slots. La idea es, cuando se produce un evento en un objeto, éste emite una señal. Esa señal se puede conectar con un slot (que no deja de ser un método de otro objeto) que responde a esa señal.

Además, es posible conectar una señal a varios slots. O hace que un slot responda a varias señales.

En el ejemplo siguiente vemos como tres objetos que emiten señales (QSpinBox, QDial y QSlider) y uno que no (QLabel) se interconectan usando este mecanismo para cambiar su estado de acuerdo a los eventos que disparan señales en cada uno de los objetos. Veamos el código.

```
1   #include <QAction>
2   #include <QApplication>
3   #include <QButtonGroup>
4   #include <QDial>
5   #include <QFrame>
6   #include <QLabel>
```

```
7  #include <QPushButton>
8  #include <QSlider>
9  #include <QSpinBox>
10 #include <QVBoxLayout>
11
12 int main(int argc, char *argv[])
13 {
14     QApplication app(argc, argv);
15     QWidget ventana;
16
17     QVBoxLayout* verticalLayout = new QVBoxLayout(&ventana);
18     QSpinBox* spinBox = new QSpinBox();
19     QDial* dial = new QDial();
20     dial->setObjectName(QString::fromUtf8("dial"));
21     QSlider* horizontalSlider = new QSlider();
22     horizontalSlider->setOrientation(Qt::Horizontal);
23     QLabel* label = new QLabel();
24     QPushButton* salirButton = new QPushButton("Salir");
25
26     verticalLayout->addWidget(spinBox);
27     verticalLayout->addWidget(dial);
28     verticalLayout->addWidget(horizontalSlider);
29     verticalLayout->addWidget(label);
30     verticalLayout->addWidget(salirButton);
31
32     ventana.show();
33
34     QObject::connect(spinBox, SIGNAL(valueChanged(int)), dial,
           SLOT(setValue(int)));
35     QObject::connect(spinBox, SIGNAL(valueChanged(int)),
           horizontalSlider, SLOT(setValue(int)));
36     QObject::connect(spinBox, SIGNAL(valueChanged(int)), label,
           SLOT(setNum(int)));
37     QObject::connect(dial, SIGNAL(valueChanged(int)), spinBox,
           SLOT(setValue(int)));
38     QObject::connect(dial, SIGNAL(valueChanged(int)),
           horizontalSlider, SLOT(setValue(int)));
39     QObject::connect(dial, SIGNAL(valueChanged(int)), label, SLOT(
           setNum(int)));
40     QObject::connect(horizontalSlider, SIGNAL(valueChanged(int)),
           spinBox, SLOT(setValue(int)));
41     QObject::connect(horizontalSlider, SIGNAL(valueChanged(int)),
           label, SLOT(setNum(int)));
42     QObject::connect(horizontalSlider, SIGNAL(valueChanged(int)),
           dial, SLOT(setValue(int)));
43     QObject::connect(salirButton, SIGNAL(clicked()), &app, SLOT(
           quit()));
44
45     return app.exec();
46 }
```

### 1.5.4. Entendiendo el código

Podemos ver que tenemos en este ejemplo algunos nuevos widgets que la librería QT nos ofrece, una vez instanciados y ordenados en un Layout vertical hacemos todas las confecciones necesarias para que cualquier evento que se produzca en los widgets capaces de emitir señales disparen una acción. En este caso en particular el cambio de cualquiera de los widgets con valores variables modifica a los otros, la única excepción es QLabel que como sabemos no es editable. En el diagrama de la Figura 1.5.

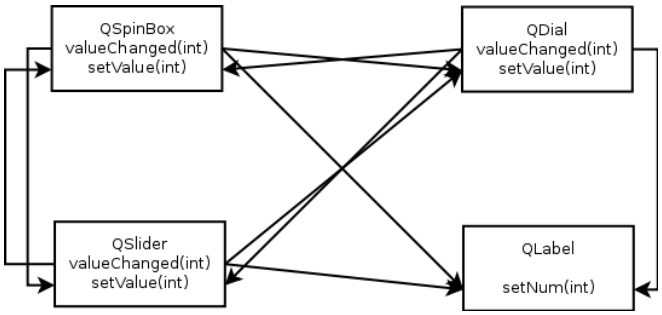


Figura 1.5: Diagrama de interconexión de señales y slots

Si ejecutamos este ejemplo obtenemos la ventana de la Figura 1.6 donde podemos ver que los valores de cada widget es coherente con los otros. Las lineas del programa que realizan estas conexiones son las siguientes.

```
1  QObject::connect(spinBox, SIGNAL(valueChanged(int)), dial, SLOT(
       setValue(int)));
2  QObject::connect(spinBox, SIGNAL(valueChanged(int)),
       horizontalSlider, SLOT(setValue(int)));
3  QObject::connect(spinBox, SIGNAL(valueChanged(int)), label, SLOT(
       setNum(int)));
4  QObject::connect(dial, SIGNAL(valueChanged(int)), spinBox, SLOT(
       setValue(int)));
5  QObject::connect(dial, SIGNAL(valueChanged(int)), horizontalSlider
       , SLOT(setValue(int)));
6  QObject::connect(dial, SIGNAL(valueChanged(int)), label, SLOT(
       setNum(int)));
7  QObject::connect(horizontalSlider, SIGNAL(valueChanged(int)),
       spinBox, SLOT(setValue(int)));
8  QObject::connect(horizontalSlider, SIGNAL(valueChanged(int)),
       label, SLOT(setNum(int)));
9  QObject::connect(horizontalSlider, SIGNAL(valueChanged(int)), dial
       , SLOT(setValue(int)));
10 QObject::connect(salirButton, SIGNAL(clicked()), &app, SLOT(quit()
       ));
```

Donde por cada widget emisor de señales se hacen tres conexiones a cada uno de los widgets capaces de recibir esas señales en sus slots.


Figura 1.6: Ventana del ejemplo 5

# Apéndice 1

## Licencia

# Index