

Introducción a Qt

<debes saber C++ por la punta dels dits !>

Estos documentos en están en una fase muy inicial y sólo son un copypaste de emergencia de lo visto y explicado en clase.

Paquetes a instalar:

En Ubuntu, conviene instalar previamente un paquete llamado build-essential

- `apt-get install build-essential`

Ahora ya podemos instalar Qt (Actualizado 20-21) los dos primeros instalan lo suficiente.

- `qttools5-dev` (`apt-get install qttools5-dev`)
- `qtdoc-qt5` o `qt5-doc`

Una vez instalados los de arriba, prueba a ejecutar los siguientes comandos

- `qmake`
- `designer`

`qmake` debería mostrar una ayuda con opciones (ya que necesita parámetros que no le pasas)

En algunos sistemas, coexisten `qt4` y `qt5`. En estos casos, al lanzar `qmake`, el sistema busca la versión `qt4` de `qmake` (nosotros hemos instalado la `qt5`). Hay que instalar (`apt-get install qt5-default`) el paquete `qt5-default` para cambiar a la versión 5 de la herramienta.

`designer` debería lanzar un programa con interfaz gráfica

Otros posibles paquetes si los anteriores no funcionan son:

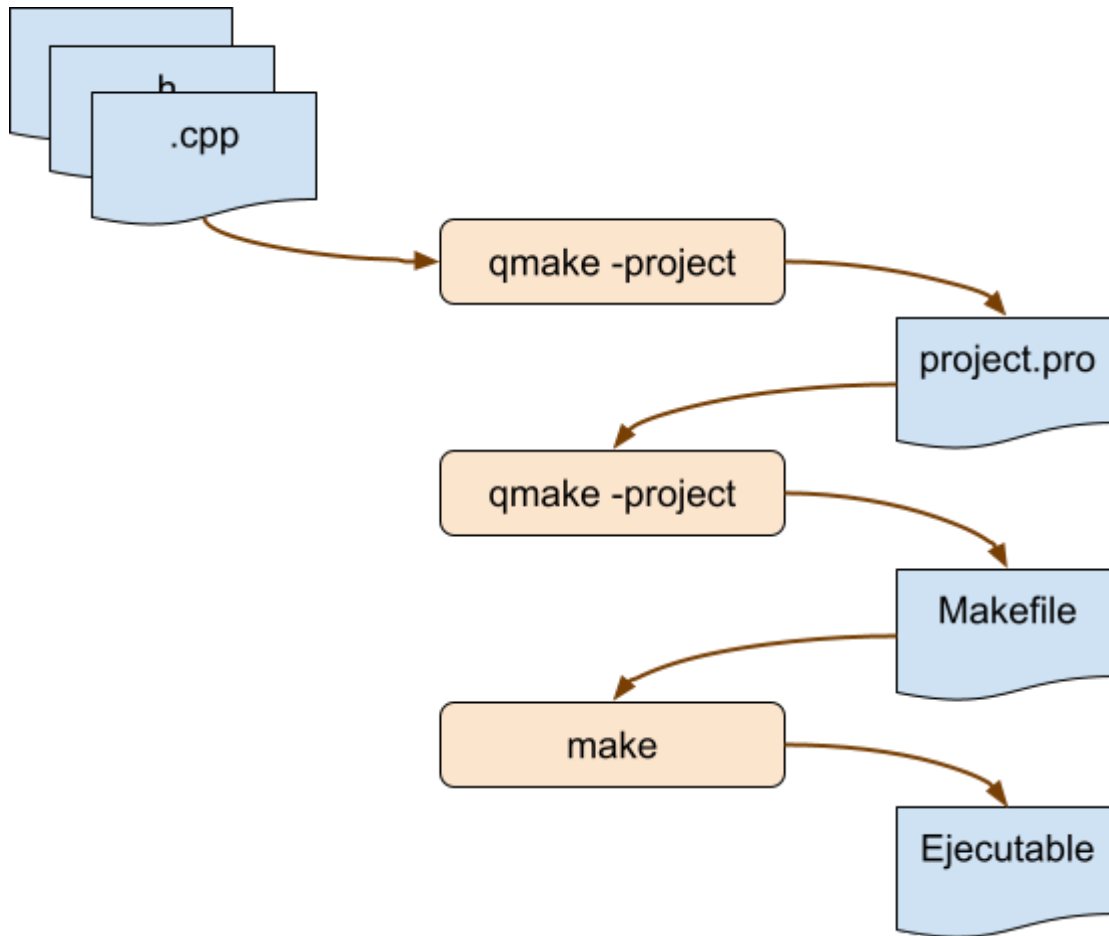
- `qttools5-dev-tools`
- `qtbases5-dev-tools`
- `qtbases5-doc`
- `qtchooser` + `qt5-default`
- `qt5-assistant`
- `qt5-qmake`

Proyectos en Qt y Hello world

Cuando se compilan programas en C compuestos de varios ficheros (lo habitual) deben crearse "makefiles". Éstos son ficheros que automatizan la compilación de los diversos ficheros y generan el ejecutable final (el programador no lanza el compilador "g++" a mano sobre cada archivo, sino que ejecuta "make" y éste automatiza toda la compilación). Crear un Makefile en C es un infierno, pero **en Qt hay una utilidad que genera**

automáticamente los makefiles necesarios. Aún así, hay que hacer algún retoque para adaptar el proyecto a nuestro uso de qt.

Observa los pasos a dar en el siguiente diagrama



Esto lo vamos a aprender directamente con el proyecto "hello world" habitual al inicio del aprendizaje de un entorno o lenguaje.

Crea un fichero "hello.cpp" con el siguiente código:

```
#include <QApplication>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel *label = new QLabel("Hello Qt!");
    label->show();
    return app.exec();
}
```

Guarda el fichero en la carpeta del proyecto, (Aquí llamada "proyecto") y sigue los siguientes pasos

Creación del proyecto, makefile y compilación

Pasos (a partir de main.cpp) para obtener el ejecutable:

1. Ejecuta `qmake -project`

y con ello se crea el proyecto

```
lliurex@01-HelloWorld$ qmake -project
lliurex@01-HelloWorld$ ls
01-HelloWorld.pro  hello.cpp  hello.cpp~
lliurex@01-HelloWorld$
```

El fichero de proyecto ".pro" que se crea tiene el nombre de la carpeta donde estás. A partir de aquí ese es el fichero ".pro" que debes manipular. el tutorial este usa "proyecto.pro" como mero ejemplo.

2. (Ahora, **esto es incómodo** pero hay que hacerlo). Hay que añadir

`"QT += widgets"`

al fichero '.pro' creado.

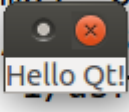
- a. Una forma de hacerlo: `echo " QT += widgets " >> proyecto.pro`
- b. Otra forma es editar con un editor el archivo y escribir esa línea.

3. Ejecutar `qmake` or `qmake hello.pro`

Esta acción generará el fichero **Makefile** que automatiza la compilación

4. Ejecutamos el comando `make` para compilar.
5. ejecutamos (el ejecutable se llama como la carpeta) y el resultado es el siguiente (sólo la ventanita que ves a la mitad de la captura pantalla):

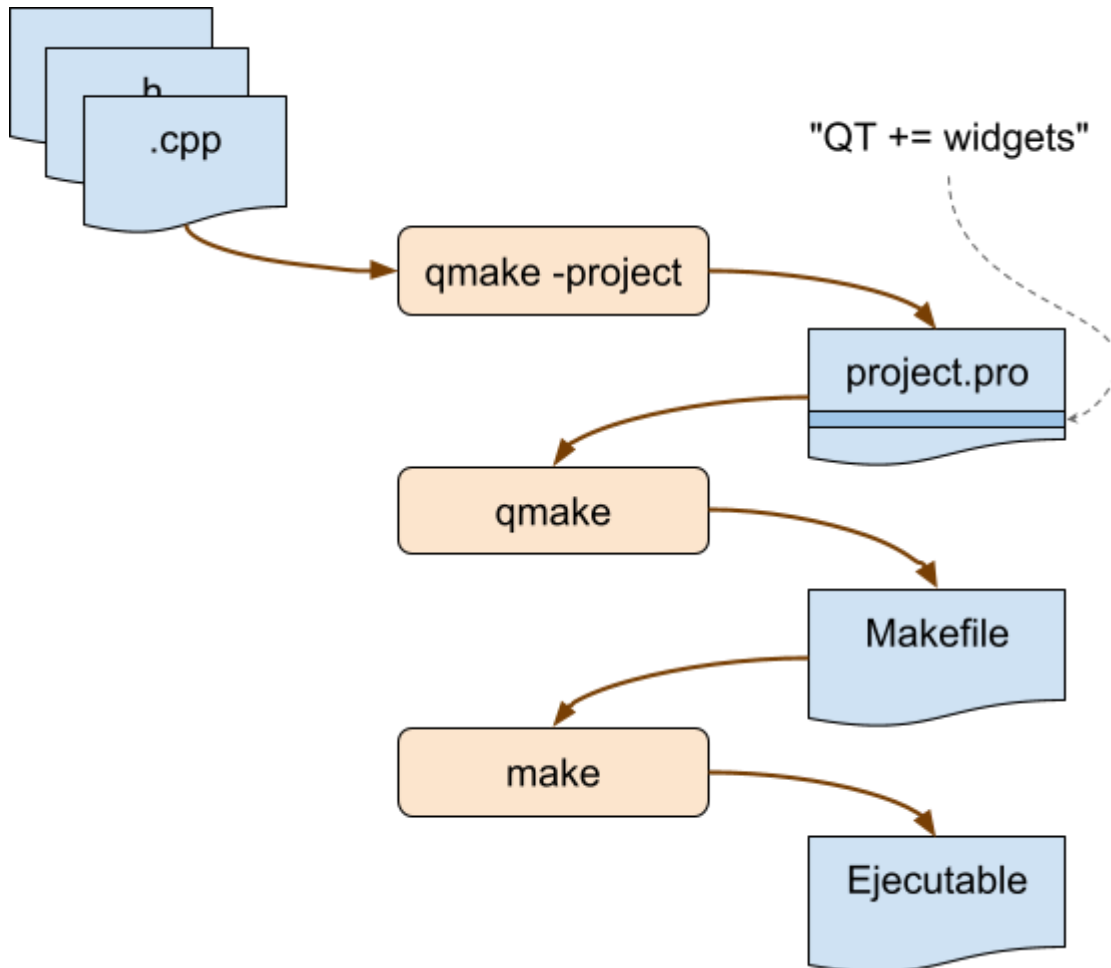
```
lliurex@HiDi-Tec:~/Documentos/Interl
g++ -c -m64 -pipe -O2 -Wall -W -D_RE
HARED -I /usr/share/qt4/mkspecs/linux
4/QtGui -I /usr/include/qt4 -I. -I. -
g++ -m64 -Wl,-O1 -o 01-HelloWorld he
lpthread
```



A partir de ahora, si cambias el código (no absolutamente siempre) pero no añades ni

eliminas un archivo, sólo has de ejecutar "make" para recompilar

El proyecto se crea y genera siempre **después** de haber creado cualquier fichero de código. Si **añades o eliminas** un archivo, tendrás que regenerar el proyecto y compilar después.



Explicación línea a línea del código

```
1 #include <QApplication>
2 #include <QLabel>

3 int main(int argc, char *argv[])
4 {
5     QApplication app(argc, argv);
6     QLabel *label = new QLabel("Hello World!");
7     label->show();
8     return app.exec();
9 }
```

Explicación línea a línea:

1. Se incluyen bibliotecas de qt
2. Se incluye la biblioteca del componente QLabel
3. Se implementa la función main() (la que inicia un programa cuando programamos en lenguaje C.
5. Se crea un objeto de tipo QApplication. Este objeto es complejo de entender e irá comprendiéndose con el paso de las lecciones. de momento puedes pensar que se trata de un supervisor de actividad que se encarga de mostrar los elementos de la interfaz de comandos y gestionar los eventos que ocurren.
6. Aquí se declara e inicializa una variable "label" que apuntará a un objeto que representa un texto dentro de una ventana de la interfaz de comandos.
7. El objeto creado anteriormente se muestra, y para ello se crea una ventana (muy pequeña) donde se podrá ver el contenido de la etiqueta de texto
8. En esta línea se pone en marcha el objeto QApplication creado antes (variable app). Este método exec() se queda en un bucle infinito y por tanto de esta función no se sale más que para terminar todo el programa.

Vamos a mejorar algo el ejemplo cambiando el texto contenido en la etiqueta

```
QLabel *label = new QLabel("<h2><i>Hello</i> "
                           "<font color=red>Qt!</font></h2>");
```

El código total queda así:

```
#include <QApplication>
#include <QLabel>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel *label = new QLabel("<h2><i>Hello</i> "
                              "<font color=red>Qt!</font></h2>");

    label->show();
    return app.exec();
}
```

¿Qué es app en ese programa?

Es la "maquinaria Qt" o "**motor Qt**" (a veces llamado "monstruo" en este documento). Es un objeto de la clase `QApplication` que está hecha por los creadores de QT.

La llamada `app.exec()` pone en marcha este motor Qt y no termina nunca. Esta función tiene en su interior un bucle infinito que inspecciona todos los eventos (clics y movimientos de ratón, mensajes de red, pulsaciones de teclado, alarmas, etc). Este bucle también actúa sobre los elementos del programa (botones, ventanas, audio, etc)

En este tutorial, llamaremos **monstruo** o **motor Qt** a este elemento para darle importancia y tenerlo siempre presente.

Experimento para demostrar el trabajo del monstruo

Si pruebas el siguiente código verás que hasta pasados 10 segundos (fíjate en el "sleep(10)) verás que no se ve la etiqueta. Esto es porque el "monstruo" no está activo hasta ese momento (sleep retrasa su activación)

```
#include <QApplication>
#include <QLabel>
#include <unistd.h>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QLabel *label = new QLabel("Hello Qt!");
    label->show();
    sleep(10);
    return app.exec(); //Arrancamos el monstruo
}
```

Recuerda:

- ¿**Modifico** código en un fichero existente? -> `make`
- ¿**Añado** o elimino un fichero .cpp o .h? -> `qmake -project; qmake...`

Señales y slots

Las señales y slots son un mecanismo *exclusivo* de Qt para que el programador indique qué reacción debe tener el programa ante eventos concretos.

Probemos ahora este código:

```

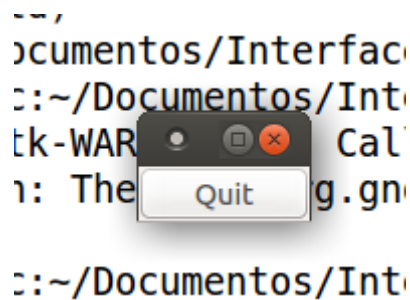
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QPushButton *button = new QPushButton("Quit");
    QObject::connect(button, SIGNAL(clicked()),
                     &app, SLOT(quit()));
    button->show();
    return app.exec();
}

```

Recuerda: " qmake -project + qmake + make "

Ejecutamos y aparece este botón minúsculo en alguna parte :



Pulsamos el botón y oh! el programa se cierra. En este ejemplo, hemos dado un **comportamiento** al botón y lo hemos conseguido mediante la línea

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

Este método connect recibe dos elementos (cada uno de los cuales lleva dos sub-elementos, de ahí que veas 4 argumentos).

- Los dos primeros elementos corresponden al componente en el que el motor de Qt debe vigilar para captar eventos ocurridos.
- Los dos segundos elementos indican qué objeto debe activarse como consecuencia del evento indicado anteriormente.

¿ Por qué hay dos argumentos en cada parte?

Hemos de indicar un **evento**, y esto viene determinado por dos elementos:

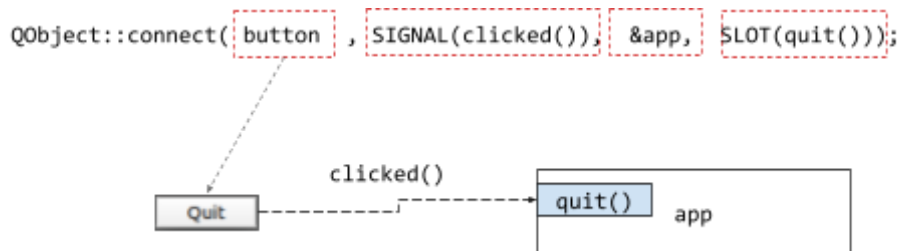
1. **Objeto** que desencadena el evento

2. Tipo de evento desencadenado

En el ejemplo, es el botón, el objeto sobre el que nos interesa detectar algo. Y aquello que deseamos detectar (o ante lo cual reaccionar) es el clic.

Por ello, el primer argumento es el botón (o un puntero a él, concretamente) y el segundo es una especificación del evento concreto: "el clic"

Para que ocurra algo como consecuencia del clic del ratón es por lo que ponemos el tercer y cuarto argumento. Básicamente ahí se está indicando un método de un objeto, y esas dos cosas son las que se pasan como tercer y cuarto argumento respectivamente



Por tanto, la instrucción anterior se interpreta como sigue:

Por favor, señor motor de QT, fíjese en el botón, y fíjese si durante el funcionamiento del programa, algo hace clic en él. Si ocurre eso, entonces por favor, invoque el método quit() del objeto app.

El objeto app es la aplicación entera, es el propio motor de Qt y el método quit() se invoca para terminar la ejecución del mismo. Por ello se logra simplemente que "al pulsar el botón se cierre el programa"

En otros lenguajes y entornos de creación de interfaces gráficas tenemos los mismos objetivos que aquí se tratan de solucionar. En concreto, en **javascript** para reaccionar a la pulsación de un botón podemos hacer

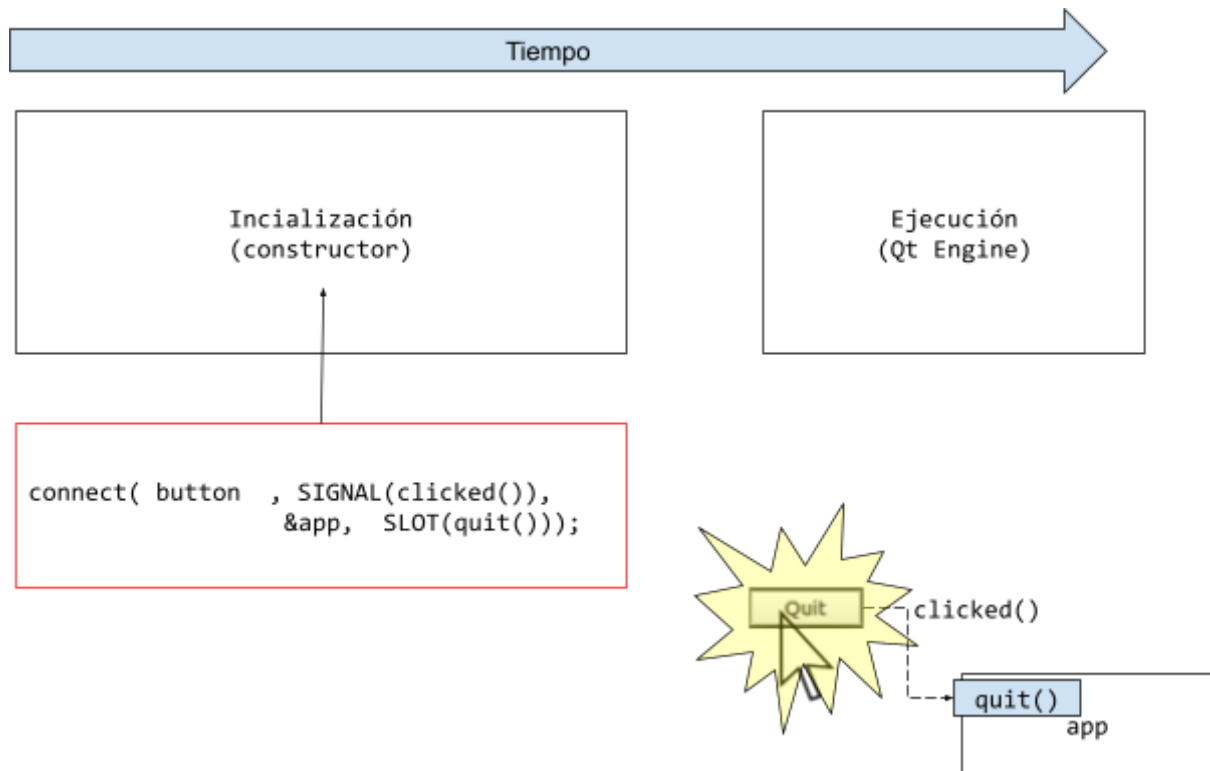
```
button.addEventListener("clic", quit);
```

o en el propio elemento html

```
<input type="button" onClick="quit">
```

Siendo quit una función javascript de la página

Es conveniente saber un hecho que nos da una visión global de lo que está pasando: La conexión de la señal con el slot se prepara o configura cuando se ejecuta constructor, pero su efecto ocurre durante la posterior ejecución de la aplicación (tiempo después de que el constructor haya finalizado). Y el efecto, en este caso, ocurre sólo si se pulsa el botón.

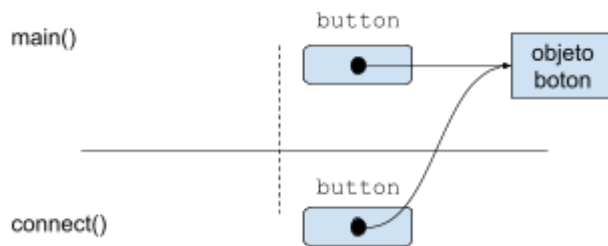


Por otra parte, esta instrucción `connect` **no es c++**. Es una construcción que durante la compilación es traducida a C++ y finalmente compilada. No debe esto preocuparte ahora, puedes seguir viviendo con ello, igual que cuando descubriste que los reyes magos son los padres.

Como ejercicio para revisar y practicar punteros, puedes cambiar la versión anterior y declarar el botón como objeto. Así, deberías pasar un puntero a un objeto en "connect" de forma diferente

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    // QPushButton *button = new QPushButton("Quit");
    QPushButton button("Quit");
    QObject::connect(&button, SIGNAL(clicked()),
                   &app, SLOT(quit()));
    button.show();
    return app.exec();
}
```



Disponiendo widgets

Después de lograr mostrar un botón, nos planteamos confeccionar diálogos más complejos que muestren más elementos. Probemos directamente el siguiente ejemplo

```
#include <QApplication>
#include <QHBoxLayout>
#include <QSlider>
#include <QSpinBox>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QWidget *window = new QWidget;
    window->setWindowTitle("Enter Your Age");

    QSpinBox *spinBox = new QSpinBox;
    QSlider *slider = new QSlider(Qt::Horizontal);
    spinBox->setRange(0, 130);
    slider->setRange(0, 130);

    QObject::connect(spinBox, SIGNAL(valueChanged(int)),
                     slider, SLOT(setValue(int)));
    QObject::connect(slider, SIGNAL(valueChanged(int)),
                     spinBox, SLOT(setValue(int)));
    spinBox->setValue(35);

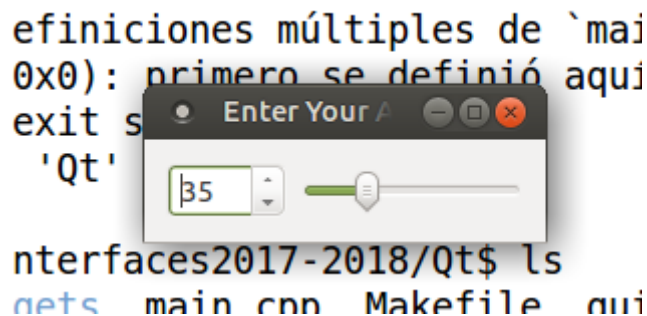
    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(spinBox);
    layout->addWidget(slider);
    window->setLayout(layout);

    window->show();

    return app.exec();
}
```

```
}
```

La salida es algo así (fíjate sólo en la ventana)



En este ejemplo aparecen dos áreas de conceptos nuevos: el comportamiento de los widgets y la disposición de los mismos en la ventana.

Disposición de ventanas

Veámos la forma en la que podemos organizar los componentes gráficos (slider y spinbox) en la pantalla.

Empezamos creando un objeto widget, que es el contenedor que incluirá a los otros (al slider y spinbox)

```
QWidget *window = new QWidget;  
window->setWindowTitle("Enter Your Age");  
...  
window->show();
```

La clase QWidget cumple varias funciones.

- Es la clase de la heredan todos los componentes gráficos (botones, QLabel, QSpinBox, QSlider, etc). De esta forma se aglutina todo el comportamiento común en una clase.
- Es un contenedor genérico que sirve para disponer dentro otros elementos gráficos. Aquí se usará así:

Para colocar el spinbox y el slider dentro del objeto window, hemos de utilizar un objeto intermedio llamado layout.

```
window->setLayout(layout);
```

De esta forma establecemos este "layout" como layout del qwidget "window"

¿Qué es un layout ?

Un Layout es un objeto que representa un espacio en la ventana dentro del cual se colocarán componentes visibles (u otros layouts internos). La característica especial es :

- Que pueden ponerse varios componentes
- Que los componentes se ubican en una disposición geométrica concreta establecida por el tipo de layout

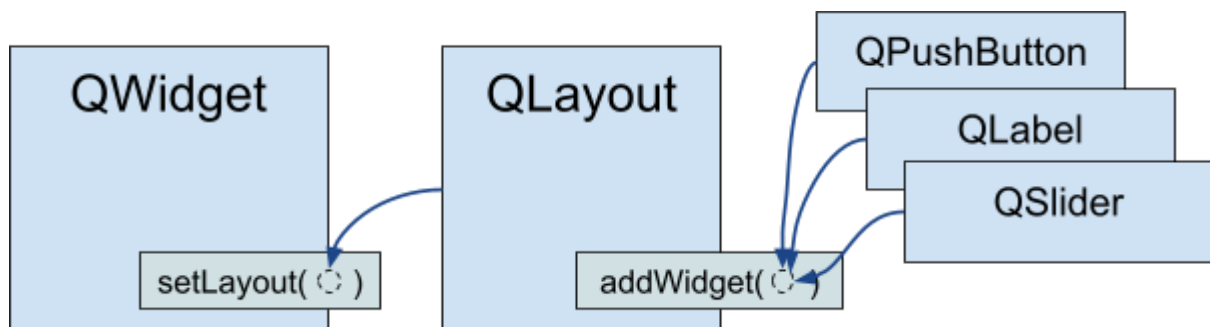
Así nuestro Layout inicializado en la siguiente línea

```
QHBoxLayout *layout = new QHBoxLayout;
```

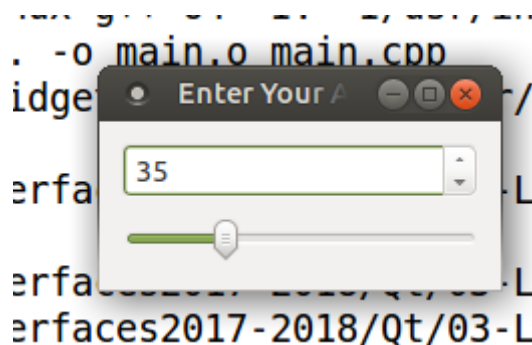
Hace que todo lo que se meta dentro de él se disponga horizontalmente (QHBoxLayout : "H" de 'horizontal'). La forma de "meter" cosas visibles (o widgets) dentro del layout es así:

```
layout->addWidget(spinBox);  
layout->addWidget(slider);
```

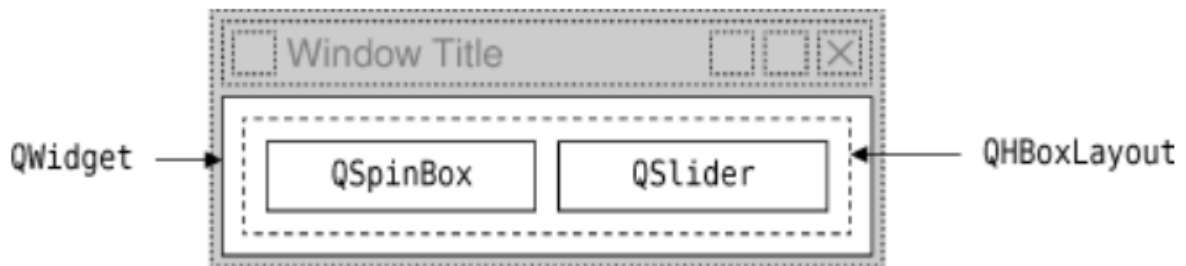
Los layouts disponen del método addWidget(QWidget *) para meter dentro de ellos componentes



Prueba en vertical con QVBoxLayout



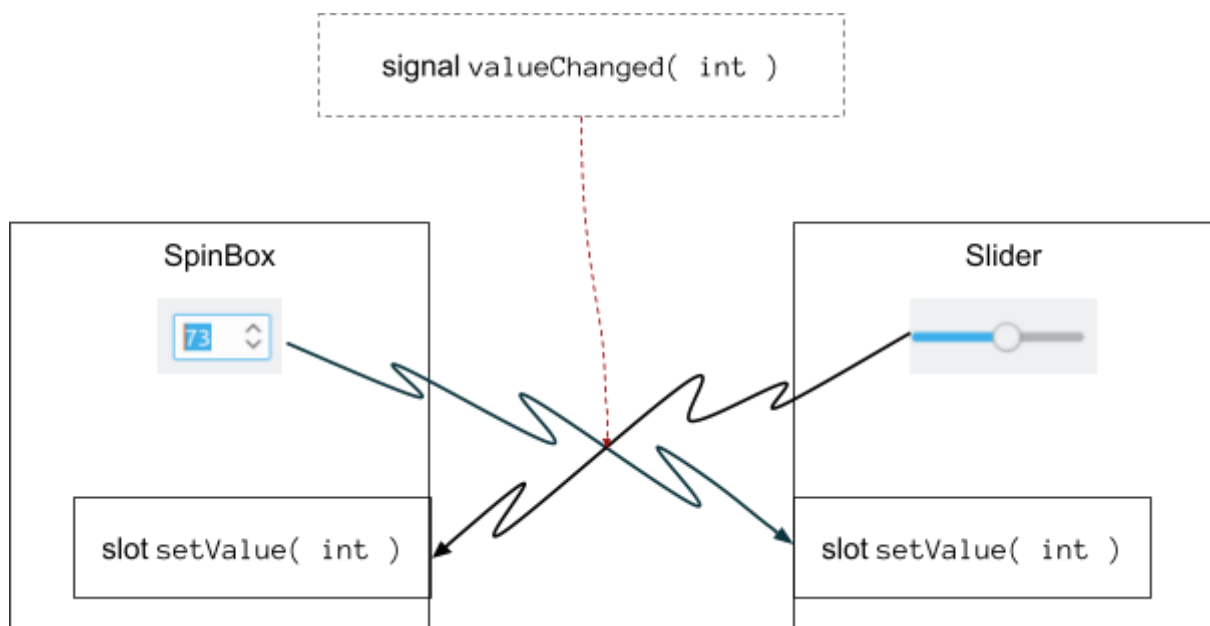
Conexión señales y slots



LA conexión de señales y slots es más compleja, pero simplemente porque estamos enlazando uno con el otro en los dos sentidos (cuando cambia un componente, se actúa sobre el otro para mantener siempre el mismo valor

```
QObject::connect(spinBox, SIGNAL(valueChanged(int)),  
                 slider, SLOT(setValue(int)));
```

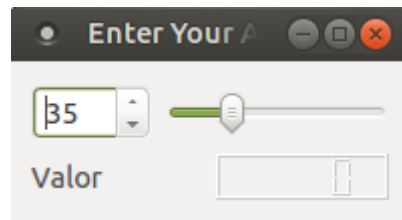
```
QObject::connect(slider, SIGNAL(valueChanged(int)),  
                 spinBox, SLOT(setValue(int)));
```



Lo que se logra es cambiar el valor de un componente como respuesta cambiar desde el interfaz gráfico . Pero si se actúa directamente con un `setValue()`, la señal no se emite, de forma que no entramos en un bucle de señales-slots infinito.

Ejercicio Resuelto

Amplía la ventan anterior para que tenga los siguientes componentes con el siguiente aspecto:



Solución:

```
QLCDNumber *qLCDNumber= new QLCDNumber();

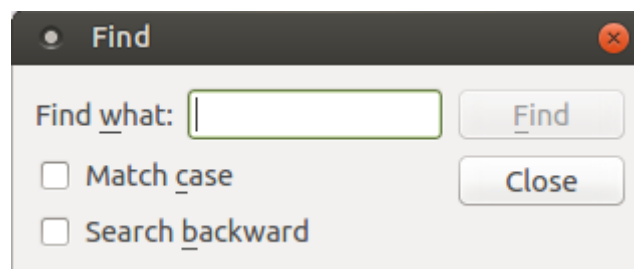
QObject::connect(spinBox, SIGNAL(valueChanged(int)),
                 qLCDNumber, SLOT(display(int)));
QObject::connect(slider, SIGNAL(valueChanged(int)),
                 qLCDNumber, SLOT(display(int)));

QVBoxLayout *layoutPrincipal = new QVBoxLayout;
QHBoxLayout *layoutSuperior = new QHBoxLayout;
QHBoxLayout *layoutInferior = new QHBoxLayout;
layoutSuperior->addWidget(spinBox);
layoutSuperior->addWidget(slider);
layoutInferior->addWidget(new QLabel("Valor"));
layoutInferior->addWidget(qLCDNumber);
layoutPrincipal->addLayout(layoutSuperior);
layoutPrincipal->addLayout(layoutInferior);

window->setLayout(layoutPrincipal);
```

Diálogos

Vamos a crear un diálogo con el siguiente aspecto



El cual tendrá un comportamiento (reaccionará a la escritura en el recuadro de texto). Hay varias ideas filosóficas que conviene tener en mente. Piensa que este es el diálogo que tú necesitas, pero nadie lo había previsto antes y te toca a tí hacerlo. Sin embargo, este diálogo no es tan diferente de otros muchos diálogos. Hay un elemento aquí que juega a nuestro favor: alguien ha pensado en que los diálogos tienen semejanzas y por ello ha implementado una clase de tipo `QDialog` (un diálogo) pero como no sabía qué elementos y comportamiento poner, lo ha dejado bastante vacío.

Sin embargo, ahora podemos **continuar** su trabajo y terminar ese diálogo. Hemos de mentalizarnos que estamos terminando el trabajo de otros que no supieron en su momento qué tipo concreto de diálogo íbamos a necesitar ahora.

Por ello crearemos **una nueva clase y heredaremos** de la clase `QDialog`. Esto se reflejará en la línea siguiente :

```
class FindDialog : public QDialog
```

Observa la parte que define que estamos heredando (`": public QDialog"`).

Heredar, tiene implicaciones muy importantes:

- Al heredar, estamos implementando un `QDialog`, nuestra clase será igualmente un `QDialog`. Todo lo que se pudiese hacer con un `QDialog` va a poderse hacer con la clase que estamos haciendo, especialmente ser mostrada por pantalla.

Esta idea se puede entender de la siguiente forma: Puedes tener un vehículo, y este vehículo puedes elegir que sea un coche, por ejemplo, pero seguirá siendo totalmente un vehículo. Todo lo que caracteriza a un vehículo está incluido en el coche, pero los coches añaden características adicionales que no todos los vehículos tienen (por ejemplo, no todos tienen motor, o maletero)

Aquí, teníamos ya la clase `QDialog` disponible, pero ahora queremos tener **nuestro `QDialog` particular**. De ahí que tengamos que ampliar o heredar `QDialog`

- Igualmente desde nuestros métodos podemos utilizar y aprovecharnos de métodos ya implementados en la clase `QDialog`. Algunos de estos métodos se han hecho con la sola intención de facilitarnos la tarea
- Todos los atributos y métodos que añadamos ahora serán una ampliación de lo que ya teníamos disponible

En definitiva, con la herencia "es como si el `qdialog` no estuviese terminado y lo tuviese que terminar yo mismo a mi gusto"

Ficheros de cabecera y ficheros de código

Dado que vamos a hacer una clase y esta clase va a ser usada por otros trozos del programa, surge un problema: Si hacemos un cambio en alguna parte del código, se recompilará todo el programa, lo cual conlleva un tiempo de compilación muy grande para cambios que pueden ser muy pequeños. La solución a esto (que también es solución a muchas otras cosas) consiste en dividir las clases que hagamos en dos ficheros:

- Ficheros de cabecera:
- Ficheros de código:

Ficheros de cabecera

En los ficheros de cabecera se definen las clases, pero no completamente. En concreto se define principalmente lo siguiente:

- Atributos
- Prototipo (o declaración) de los métodos.

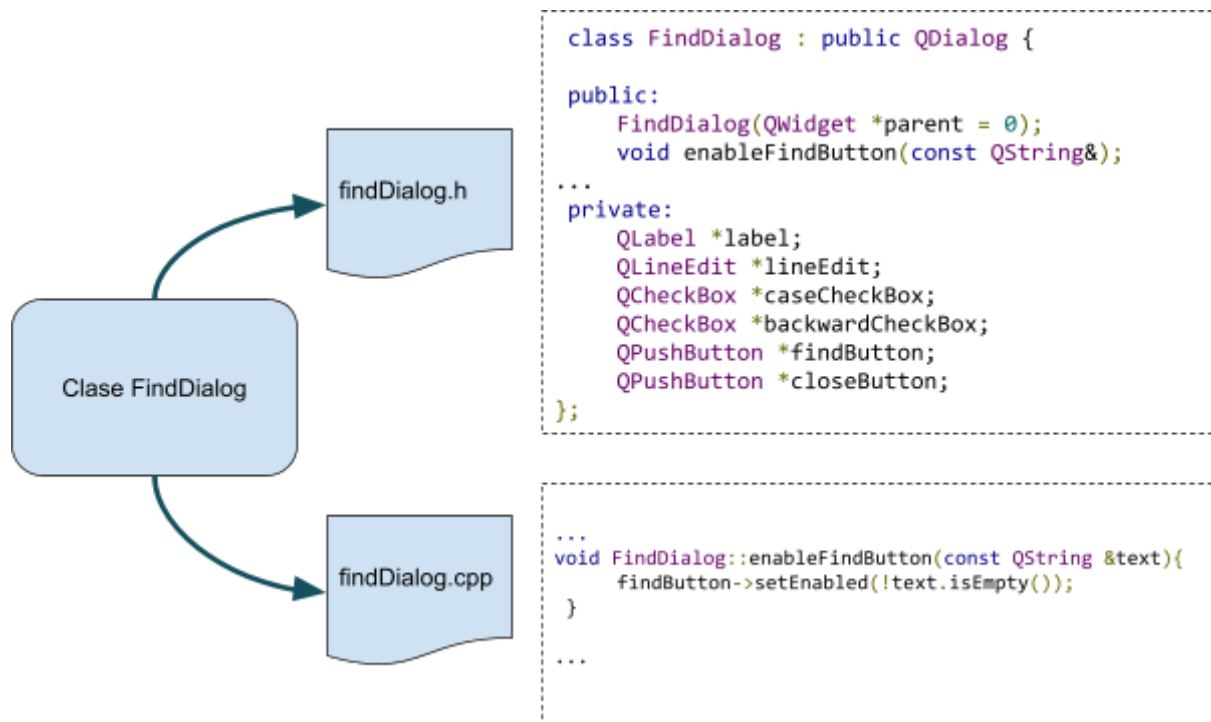
Los ficheros de cabecera tienen la extensión ".h" y son los que se #include desde otros ficheros que los utilicen

La idea de los ficheros de cabecera es sencilla: Ponlo todo menos cualquier instrucción o código ejecutable.

Ficheros de código

Los ficheros de código, llevan principalmente la implementación de los métodos de la clase. A diferencia de Java, en C++, las clases pueden estar declaradas y definidas en diferentes ficheros y lugares.

Observa el siguiente diagrama centrado en el método **enableFindButton()**



En el fichero superior (findDialog.h), aparece este método en la declaración de la clase, dentro de la palabra o sección class{ ... }. Sin embargo, allí no existe ningún código asociado a ese método (no tiene "{" con instrucciones dentro). Está vacío.

En el fichero inferior (findDialog.cpp) aparece ese método, otra vez (aunque con el prefijo "FindDialog::". Esta vez sí que se abren y cierran unas llaves "{" y aparece código en su

interior. Aquí es donde estamos **implementando** finalmente el método, mientras que antes sólo lo **declarábamos**. El motivo de dividir esto así cae fuera de nuestros objetivos ahora mismo.

A partir de ahora siempre declararemos la clase junto con sus métodos y atributos en un fichero de cabecera (.h) sin poner código (salvo contadas excepciones); e implementaremos los métodos en otro fichero de código ".cpp"

Observa Declaració clase en finddialog.h. Assegurança per no passar dues vegades :

```
#ifndef PALABRA
#define PALABRA

... código ...

#endif
```

findDialog.h

```
#ifndef FINDDIALOG_H
#define FINDDIALOG_H

#include <QDialog>

class QCheckBox;
class QLabel;
class QLineEdit;
class QPushButton;

class FindDialog : public QDialog
{
    Q_OBJECT

public:
    FindDialog(QWidget *parent = 0);

signals:
    void findNext(const QString &str, Qt::CaseSensitivity cs);
    void findPrevious(const QString &str, Qt::CaseSensitivity cs);

private slots:
    void findClicked();
    void enableFindButton(const QString &text);

private:
    QLabel *label;
    QLineEdit *lineEdit;
    QCheckBox *caseCheckBox;
    QCheckBox *backwardCheckBox;
    QPushButton *findButton;
    QPushButton *closeButton;
};
#endif
```

findDialog.cpp

```

#include <QtGui>
#include <QCheckBox>
#include <QLabel>
#include <QLineEdit>
#include <QPushButton>
#include <QDialog>
#include <QHBoxLayout>
#include "findDialog.h"

FindDialog::FindDialog(QWidget *parent)
    : QDialog(parent)
{
    label = new QLabel(tr("Find &what:"));
    lineEdit = new QLineEdit;
    label->setBuddy(lineEdit);

    caseCheckBox = new QCheckBox(tr("Match &case"));
    backwardCheckBox = new QCheckBox(tr("Search &backward"));

    findButton = new QPushButton(tr("&Find"));
    findButton->setDefault(true);
    findButton->setEnabled(false);

    closeButton = new QPushButton(tr("Close"));

    connect(lineEdit, SIGNAL(textChanged(const QString &)),
            this, SLOT(enableFindButton(const QString &)));
    connect(findButton, SIGNAL(clicked()),
            this, SLOT(findClicked()));
    connect(closeButton, SIGNAL(clicked()),
            this, SLOT(close()));

    QHBoxLayout *topLeftLayout = new QHBoxLayout;
    topLeftLayout->addWidget(label);
    topLeftLayout->addWidget(lineEdit);

    QVBoxLayout *leftLayout = new QVBoxLayout;
    leftLayout->addLayout(topLeftLayout);
    leftLayout->addWidget(caseCheckBox);
    leftLayout->addWidget(backwardCheckBox);

    QVBoxLayout *rightLayout = new QVBoxLayout;
    rightLayout->addWidget(findButton);
    rightLayout->addWidget(closeButton);
    rightLayout->addStretch();

    QHBoxLayout *mainLayout = new QHBoxLayout;
    mainLayout->addLayout(leftLayout);
    mainLayout->addLayout(rightLayout);
    setLayout(mainLayout);
    setWindowTitle(tr("Find"));
    setFixedHeight(sizeHint().height());
}

void FindDialog::findClicked()

```

```

{
    QString text = lineEdit->text();
    Qt::CaseSensitivity cs =
        caseCheckBox->isChecked() ? Qt::CaseSensitive
                                   : Qt::CaseInsensitive;

    if (backwardCheckBox->isChecked()) {
        emit findPrevious(text, cs);
    } else {
        emit findNext(text, cs);
    }
}

void FindDialog::enableFindButton(const QString &text){
    findButton->setEnabled(!text.isEmpty());
}

```

main.cpp

```

#include <QApplication>

#include "findDialog.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    FindDialog *dialog = new FindDialog;
    dialog->show();
    return app.exec();
}

```

<diagrama de clases i fitxers >

Primera. Vull un diàlego que tenga un comportament propi, no puc utilitzar coses hechas como antes, he de hacer mi propio diàlego. El programa principal, tan sólo es la creación de un objeto de este tipo de diàlego propio + su show().

```

FindDialog *dialog = new FindDialog;
dialog->show();

```

Por ello el proyecto se divide en tres ficheros:

- main.cpp,
- finddialog.h
- finddialog.cpp

contenido genérico de finddialog.h (clase, constructor y dos métodos), finddialog.cpp (implementación de los métodos y constructores).

En el constructor creas objetos (widgets) y los pones en Layouts, al final mediante

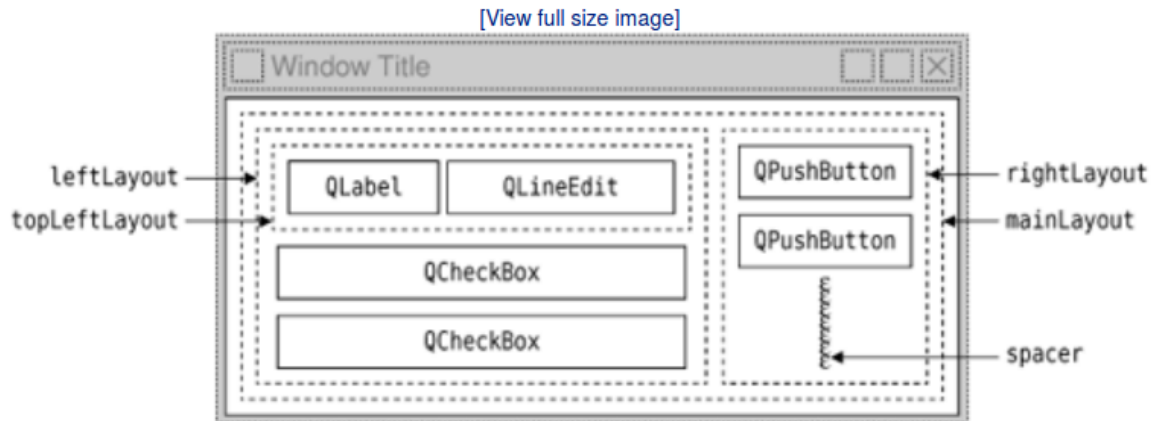
```

setLayout(mainLayout);

```

Logras mostrar todos los componentes en el diàlego, que es la propia clase en la que se está haciendo el constructor, no hay que hacer como antes (window->setLayout(layout);)

porque la propia clase en la que estoy es YA un QDialog y se inserta a ella misma el layout con los componentes.



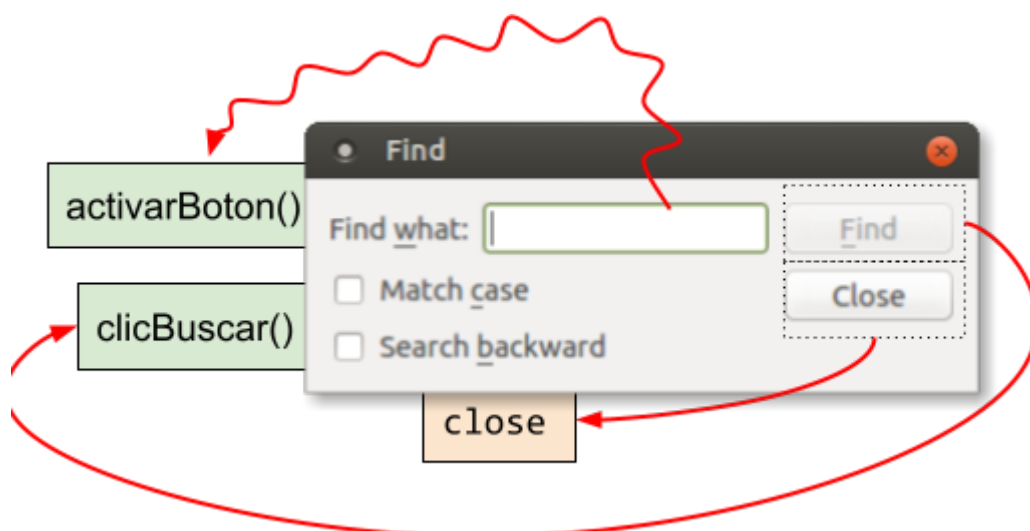
Señales y slots

Observa las siguientes conexiones y deduce cómo estamos determinando la reacción del diálogo.

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SLOT(enableFindButton(const QString &)));

connect(findButton, SIGNAL(clicked()),
        this, SLOT(findClicked()));

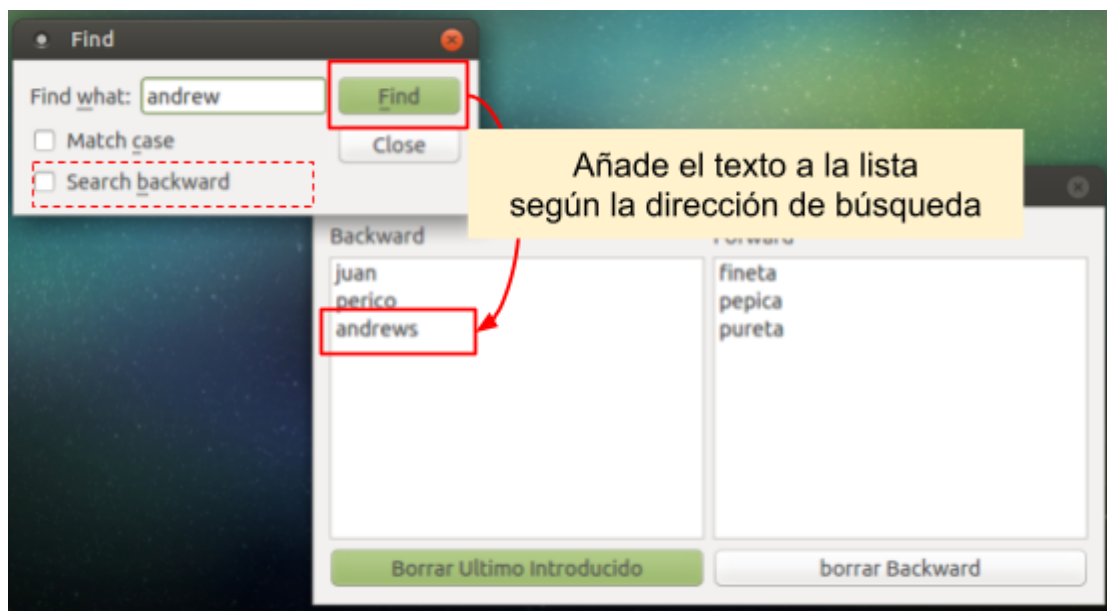
connect(closeButton, SIGNAL(clicked()),
        this, SLOT(close()));
```



Conexión entre diferentes ventanas

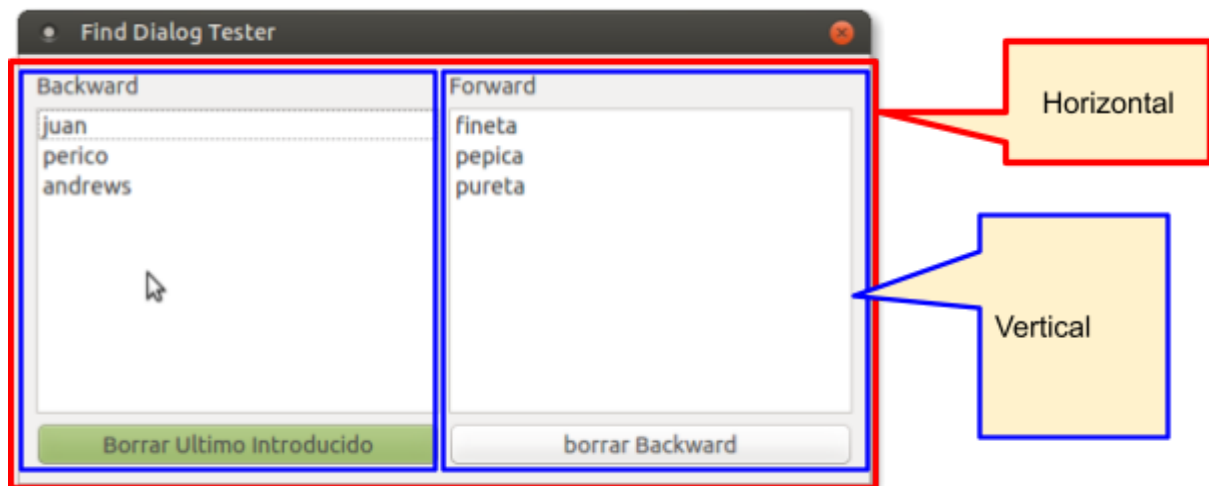
El diálogo anterior Find Dialog no puede ser probado porque las señales que emite no están conectadas a nada. Vamos a crear otro diálogo dentro del proyecto (no modifiques el proyecto, hazte una copia y sigue desarrollando en la copia), en el cual habrá otro diálogo que nos ayudará a probar el diálogo Finddialog.

Este otro diálogo tendrá el siguiente aspecto



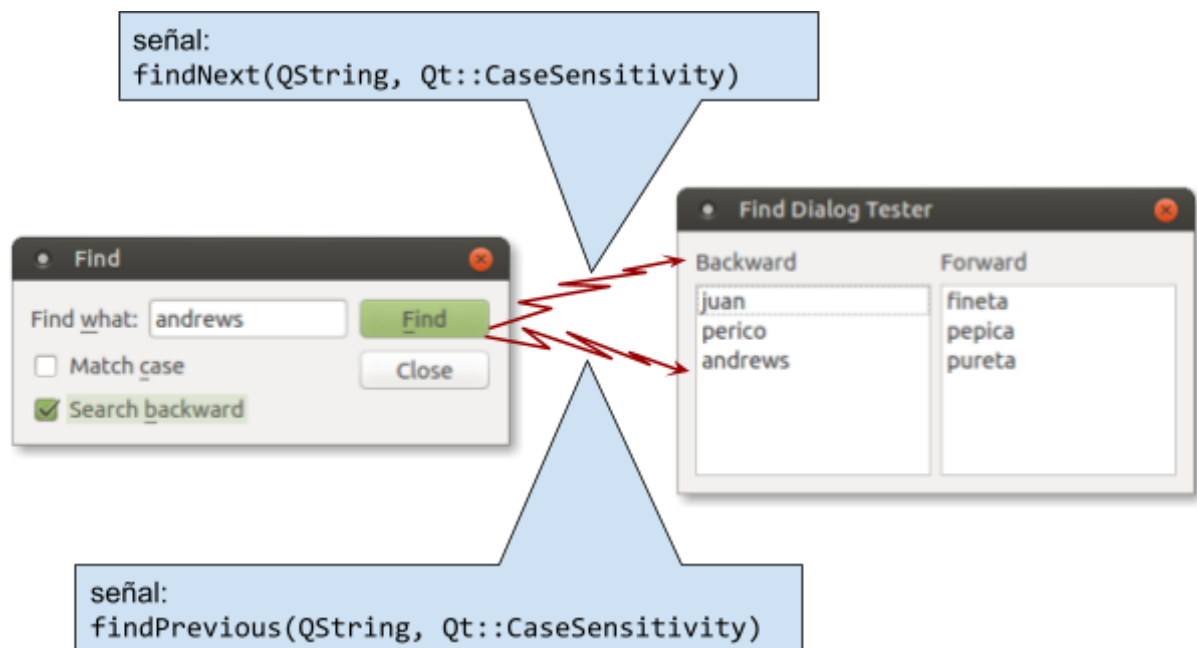
(los dos botones "Borrar Ultimo Introducido" y "Borrar Backward" son para una ampliación posterior del ejercicio.

El diseño visual se basa en la siguiente disposición de layouts



Hay que hacer una clase nueva FindDialogTester que se implementará en dos ficheros.

En la declaración hay que hacer dos slots para las dos señales



Declaración de los slots en findDialogTester.h

public slots:

```
void addForward(const QString &, Qt::CaseSensitivity cs);
```

```
void addBackward(const QString &, Qt::CaseSensitivity cs);
```

La implementación es muy sencilla (findDialogTester.cpp)

```
void FindDialogTester::addForward(const QString &palabra,
Qt::CaseSensitivity cs){
    listaForward->addItem(palabra);
}

void FindDialogTester::addBackward(const QString &palabra,
Qt::CaseSensitivity cs){
    listaBackward->addItem(palabra);
}
```

Para que funcione hay que realizar la conexión de las señales y slots reflejada en el dibujo de antes. Aquí se pega el contenido de main.cpp

```
#include <QApplication>
#include "DialogoBuscar.h"
#include "DialogoProbador.h"

int main(int argc, char *argv[] ) {
    QApplication app(argc, argv);

    DialogoBuscar dialogo;
    dialogo.show();

    DialogoProbador dialogoProbador;
    dialogoProbador.show();

    QObject::connect(
        &dialogo,
        SIGNAL(findNext(const QString &, Qt::CaseSensitivity)),
        &dialogoProbador,
        SLOT(slotForward(const QString &, Qt::CaseSensitivity)));

    QObject::connect(
        &dialogo,
        SIGNAL(findPrevious(const QString &, Qt::CaseSensitivity)),
        &dialogoProbador,
        SLOT(slotBackward(const QString &, Qt::CaseSensitivity)));

    return app.exec();
}
```

```
}
```

Ampliación

- Borrar la última entrada de la lista backward mediante un botón.
- Borrar la última entrada introducida en cualquiera de las dos listas

Implementación de borrar la última entrada:

...sé que voy a borrar, lo que ahora no me importa es de qué lista voy a borrar, supongo que hay una variable que ya lo sabe:

```
void FindDialogTester::slotBorrarUltimo( ){  
    listaABorrar->takeItem(listaABorrar->count()-1);  
}
```

Ahora hay que pensar cómo mantener el valor de la variable correcta:

```
void FindDialogTester::addForward(const QString &palabra,  
Qt::CaseSensitivity cs){  
    listaForward->addItem(palabra);  
    listaABorrar = listaForward;  
}
```

```
void FindDialogTester::addBackward(const QString &palabra,  
Qt::CaseSensitivity cs){  
    listaBackward->addItem(palabra);  
    listaABorrar = listaBackward;  
}
```

No olvidemos que el usuario podría pulsar el botón antes de meter nada en las listas y disparar la ejecución del slot `slotBorrarUltimo` sin que existe una dirección de memoria correcta: Inicializamos en el constructor para "tirar p'alante"

```
FindDialogTester::FindDialogTester(QWidget *parent) : QDialog(parent){  
    ...
```

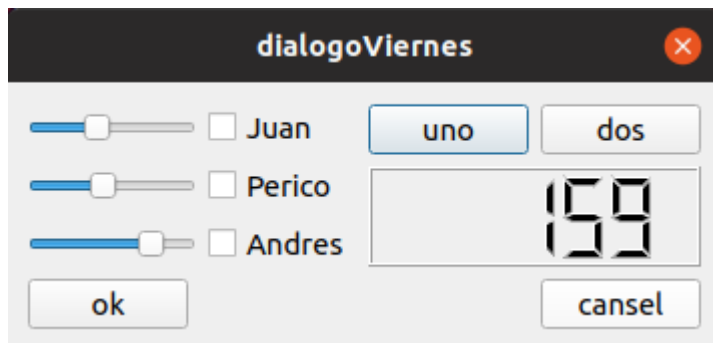


```
listaABorrar = listaForward;
```

Anexo.

Ejercicio voluntario

Ejercicio diálogo complejo (llamado aquí DialogoViernes).



Se pide:

1. Componer el diálogo con el aspecto de la captura
2. El Display LCD Mostrará la suma de los valores de los sliders.
3. Los botones uno y dos no tienen ninguna funcionalidad, pero hay que incluirlos en el diálogo.